HAL open science

Interact : un modèle général de contrat pour la garantie des assemblages de composants et services

Alain Ozanne

To cite this version:

Alain Ozanne. Interact : un modèle général de contrat pour la garantie des assemblages de composants et services. Génie logiciel [cs.SE]. Université Pierre et Marie Curie - Paris VI, 2007. Français. NNT : . tel-00292148v2

HAL Id: tel-00292148

https://theses.hal.science/tel-00292148v2

Submitted on 24 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

présentée à

L'Université Pierre & Marie Curie - Paris VI

en vue de l'obtention du titre de

Docteur de l'Université Pierre & Marie Curie

Spécialité

Informatique

par

Alain OZANNE

Sujet de la thèse :

Interact : un modèle général de contrat pour la garantie des assemblages de composants et services

Soutenue le 30 novembre 2007 devant un jury composé de :

M. Antoine Beugnard	Rapporteur	Enseignant Chercheur HDR (ENST)
M. Pierre Bazex	Rapporteur	Professeur des Universités (Toulouse III, IRIT)
M. Jacques Malenfant	Directeur	Professeur des Universités (Paris VI, LIP6)
M. Nicolas Rivierre	Co-encadrant	Chercheur (FT R&D)
M. Philippe Collet	Co-encadrant	Maître de Conférences (UNSA, I3S)
Mme Marie-Pierre Gervais	Examinatrice	Professeur des Universités (Paris X, LIP6)
M. Noël Plouzeau	Examinateur	Maître de Conférences (Rennes 1, IRISA)

Remerciements

Je tiens à remercier très chaleureusement mes encadrants, Nicolas Rivierre, Philippe Collet et Jacques Malenfant pour leur disponibilité, leur engagement et leur patience. Je remercie aussi Thierry Coupaye qui m'a suggéré la possibilité de faire une thèse alors que j'étais en stage.

Résumé

Pour satisfaire aux nouveaux besoins de flexibilité, modularité, d'adaptabilité et de distribution des applications, les paradigmes composants et services ont été déclinés dans des frameworks reconnus comme J2EE, OSGI, SCA ou encore Fractal. Néanmoins, ceux-ci offrent peu d'outils permettant de garantir la fiabilité des applications en raisonnant de manière générique sur leur configuration architecturale et les spécifications des participants. Dans cette thèse, j'envisage l'organisation de la vérification des assemblages, et le diagnostic des défaillances, sous l'angle de l'approche par contrat dirigée par les responsabilités. Pour cela, j'analyse d'abord sous quelles hypothèses intégrer différents formalismes à cette approche, puis comment appliquer cette approche à différentes architectures. J'étudie par ailleurs comment les intervenants de la mise en oeuvre des systèmes pourraient en bénéficier. Cela m'amène à présenter un modèle de contrat, qui intègre et organise différentes propriétés, analysées comme requises pour la validité de l'assemblage, conjointement et uniformément sur différentes architectures. J'en définis le modèle objet qui réifie la logique contractuelle, ainsi que son implémentation sous forme d'un framework. Ce dernier est validé sur l'architecture Fractal et deux formalismes contractuels, l'un à base d'assertions et l'autre de contraintes sur les séquences d'interactions valides entre participants. Une validation plus avancée est montrée sur l'exemple d'une application de communautés instantanées.

Mots clés :

contrat, composant, service, composition, responsabilité, conformité, compatibilité

Erratum

Etant donné qu'on m'a posé la question plus d'une fois, je crois qu'il est utile que je positionne Interact par rapport à ConFract. Tout d'abord, Interact n'est pas une extension de ConFract. Interact accepte différents formalismes et s'applique à différentes architectures, il est ainsi plus général que ConFract. Il faut encore noter qu'un contrat de ConFract est bâti sur une unique spécification, qui peut porter sur plusieurs composants, et que ainsi il n'exprime et n'évalue que la conformité d'un ou plusieurs composants à cette spécification. Un contrat dans Interact considère les spécifications propres à chacun des composants de l'assemblage, il réifie, pour en organiser l'évaluation et l'analyse, la conformité de chaque composant à sa spécification, mais aussi et surtout la compatibilité de ces composants via celle de leurs spécifications.

Prototype

J'ai développé un prototype fonctionnel d'Interact et l'ai appliqué à une petite application. Il est disponible auprès de France Telecom R&D, faites moi un mail (a.ozanne@free.fr) et je vous mettrai en contact avec la personne responsable.

Title

Interact : a general contract model for the guarantee of components and services assemblies.

Abstract

To meet the new requirements of applications in terms of flexibility and modularity and upgradability as well as distribution, the components and services paradigms have been used in well known frameworks such as J2EE, OSGI, SCA or Fractal. However, these don't offer tools for reasonning in a generic way on their architectural configuration and the specifications of their parts. In this thesis, I consider the organization of the verification of assemblies, and the diagnosis of their failures, from the point of view of a contractual, responsibility driven, approach. To this end, I analyze firstly the assumptions under which integrating various formalisms in this approach, then how to apply this approach to different architectures. Moreover, I study how people, intervening in the implementation of these systems, may take advantage of it. This leads me to present a contract model, which integrates and organizes various properties (analyzed as required for assembly dependability) jointly and uniformly on different kinds of architectures. I define an object model which reifies the contractual logic, and its implementation as a framework. This latter is validated on the Fractal component model and two contractual formalisms, one assertion-based and the other constraining interactions sequences between participants. A more advanced validation is shown on the exemple of an instant communities application.

Keywords :

contract, component, service, composition, responsibility, conformity, compatiblity

Erratum

As I have been asked several time, I think it is useful that I give the main differences between Interact and ConFract. Firstly, Interact is not an extension of ConFract. Interact accepts various formalisms and applies to various models of components, so it is more general than ConFract. Moreover one should notice that a contract in ConFract is built upon one single specification, that can constrain one or many components, and thus this contract expresses and checks only the conformity of this or these component(s) to the given specification. A contract in Interact consider the specifications of each component of the assembly, then it reifies not only the conformity of each component to its specification, but also of the compatibility of these components through the one of their specifications.

Prototype

I have implemented an operational prototype and its application to an example system. They are available by France Telecom R&D, send me a mail and I will connect you to the person in charge.

Table des matières

1 Introduction **13**

2 Etat de l'art **17**
2.1 Introduction 17
2.2 Architectures logicielles 17
2.2.1 Systèmes à objets 18
2.2.2 Systèmes à composants 19
2.2.3 Architecture de services 23
2.3 Fondation de l'approche contractuelle 26
2.3.1 Définition du contrat logiciel 26
2.3.2 Etude de la "conception par contrat" 29
2.4 Formalismes de spécification 32
2.4.1 Spécifications fonctionnelles appliquées aux objets 32
2.4.2 La conception par contrat appliquée aux composants 37
2.4.3 Spécifications non-fonctionnelles appliquées aux objets et interfaces 44
2.4.4 Spécifications non-fonctionnelles appliquées aux composants . . 46
2.4.5 Spécifications non-fonctionnelles appliquées aux services 49
2.5 Le raisonnement par hypothèse garantie 52
2.5.1 Paradigme Assume-guarantee et logique de Hoare 52
2.5.2 Portées architecturales des différents types de spécifications . . . 54
2.5.3 Positionnement par rapport à la notion de compatibilité 54
2.5.4 Conclusion 56
2.6 Les modèles et métamodèles de contrat 56
2.6.1 Méta-modèles de contrat 56
2.6.2 Objets 57
2.6.3 Composants 59
2.6.4 Service 65
2.6.5 Agents 69
2.6.6 Bilan 69
2.7 Conclusion 70

3 Modèle de contrat **71**
3.1 Introduction 71
3.2 Système exemple 72
3.2.1 Description 72
3.2.2 Spécifications du système 74
3.3 Analyse et choix de conception 76
3.3.1 Propriétés du système réifiées par le contrat 76
3.3.2 Architectures acceptées par le modèle de contrat 82
3.3.3 Formalismes acceptés par le modèle de contrat 84
3.3.4 Synthèse 87

3.4 Conception du modèle générique de contrat 89
3.4.1 Les constituants du contrat 89
3.4.2 Vue d'ensemble 93
3.4.3 Le motif architectural (`ArchitecturalPatternInstance`) 94
3.4.4 Le type de contrat (`ContractType`) 99
3.4.5 Cycle de vie du contrat 100
3.4.6 Evaluation du contrat 102
3.5 Modèle hiérarchique 103
3.5.1 Exploitation des dépendances entre contrats 105
3.6 Conclusion 106

4 Mise en oeuvre 107
4.1 Introduction 107
4.2 Architectures 107
4.2.1 Modélisation de l'architecture 108
4.2.2 Modélisation des observations 111
4.3 Formalismes 117
4.3.1 Modélisation des contraintes sur un système 117
4.3.2 Modélisation de contraintes sous une forme évaluable 121
4.3.3 Interprétation des spécifications en hypothèses et garanties . . . 122
4.3.4 Mise en oeuvre dans un contrat 129
4.4 Conclusion 140

5 Validation 141
5.1 Introduction 141
5.2 Cas d'étude 141
5.2.1 Contexte de l'exemple 141
5.2.2 Motivation de l'utilisation des contrats 142
5.2.3 Le système Amui 143
5.2.4 La mise en oeuvre des contrats 148
5.2.5 Conclusion 153
5.3 Implémentation 154
5.3.1 Gestion des contrats 154
5.3.2 Système de contrat 156
5.3.3 Plugins et spécialisation du système de contrat 158
5.4 Discussion 167
5.4.1 Par rapport à l'état de l'art 167
5.4.2 Intérêts de l'approche 168
5.4.3 Limites de la solution 169
5.5 Conclusion 170

6 Conclusion 171

A Compléments à l'état de l'art des formalismes 177
A.1 Formalismes fontionnels pour les objets 177
A.1.1 Extension temporelle d'OCL : 177
A.1.2 JML 177
A.2 Formalismes fonctionnels pour les composants 181
A.2.1 Plate-forme .Net 181

A.2.2 Plate-forme J2EE 181
A.2.3 L'approche par modèle de Barnett 182
A.2.4 Wright et Darwin 183
A.2.5 Description Logic et Kind Theory 185
A.2.6 Approche assume-guarantee d'Abadi et Lamport 186
A.3 Formalismes non fonctionnels pour les objets 187
A.3.1 QuO 187
A.3.2 QML 189
A.4 Formalismes non fonctionnels pour les composants 190
A.4.1 Le modèle des KComponent 190
A.4.2 CQML 191
A.4.3 CQML+ 193
A.5 Formalismes non fonctionnels pour les services 194
A.5.1 WSLA 194

B Plugins et spécialisation du système de contrat 197
B.1 Plugin d'architecture 197
B.2 Plugin de formalisme 199

C Exemples de contrat de validation 201
C.1 Contrat basé sur les Behavior Protocol 201
C.2 Contrat basé sur des spécifications CCLJ 203

D Publications 207

CHAPITRE 1

Introduction

Alors que les applications voient augmenter leur taille, leurs adaptabilité et flexibilité, ainsi que leur durée de vie, d'utilisation et leur aspect distribué, elles nécessitent de plus en plus une architecture manipulable, modulaire et adaptable. Pour satisfaire à ces nouveaux besoins, les paradigmes composants et services ont été développés, ouvrant la voie à la construction par assemblage des applications. De nombreux *frameworks* ont ainsi été conçus pour supporter l'exécution d'architectures orientés services ou composants, tels SCA, J2EE, OSGI ou encore Fractal. Ces *frameworks* mettent en oeuvre divers services techniques comme la gestion du cycle de vie des applications ou celle des ressources disponibles. Néanmoins, ils offrent peu de moyens pour garantir la fiabilité, ou gérer les défaillances, des applications en raisonnant de manière générique sur leur configuration architecturale et les spécifications de leurs parties.

Dans ce contexte, le bon fonctionnement d'applications, constituées d'assemblages d'éléments logiciels, repose sur la bonne collaboration de ces derniers. Pourtant bien qu'une cause essentielle d'erreur d'assemblage réside dans des propriétés restées implicites, il n'a pas été développé d'outil contractuel permettant de valider une architecture en intégrant simultanément différents formalismes (et leurs outils de vérification) développés séparément. Ceci, alors que les architectes ou administrateurs d'architectures concrètes ne s'intéressent pas à une unique propriété ou un unique formalisme, mais à la validité globale de leur système.

Il convient, pour concevoir un tel outil, de considérer non seulement l'approche contractuelle de la validation des assemblages, mais aussi comment les différents intervenants du système observé sont impliqués dans celle-ci. Ainsi pour l'architecte de l'application, la mise en oeuvre de l'outil d'évaluation de la validité de l'assemblage doit être transparente. L'architecture de l'application ne doit pas être modifiée pour permettre l'application de cet outil, et l'architecte ne doit pas avoir à acquérir une expertise supplémentaire à sa spécialité. L'auteur des spécifications (assertions, automates...) doit pouvoir les rédiger et les mettre en oeuvre, sans se préoccuper de l'intégration de leur formalisme à l'architecture qu'elles contraignent. L'administrateur de l'application est responsable du maintien de son bon fonctionnement. Il a ainsi autant besoin d'un indicateur de ce dernier, que du diagnostic précis d'une défaillance pour mettre en oeuvre son rétablissement, ce que lui fournit l'évaluation de l'assemblage. Enfin, un des objectifs de cet outil est de permettre l'application conjointe de différents formalismes à des architectures diverses. A cette fin, architectures et formalismes doivent lui être interfacés. Pour ce faire, un intégrateur de formalisme doit produire l'extension correspondante à l'outil d'évaluation, ce qui permet d'appliquer ce formalisme uniformément aux architectures qui sont intégrées à l'outil. Un intégrateur d'architecture doit produire l'extension convenable à l'outil d'évaluation, ce qui permet d'appliquer uniformément à l'architecture, les formalismes intégrés à l'outil.

Objectifs et démarche. Pour répondre à cette problématique multiple, nous avons choisi l'approche par contrat. En effet, dans le cadre des paradigmes composant et service, les contrats sont devenus une part de leur définition [101] :

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only*

Ils sont utilisés pour organiser la garantie de leurs propriétés tout au long du cycle de vie logiciel. Toutefois, les contrats classiques, qu'ils soient syntaxiques, de QoS ou comportementaux (comme distingués dans [13]), sont encore utilisés plus comme une garantie de la satisfaction à une spécification, comme pour les contrats objet [71], que comme un réel outil architectural. Mais dans son principe le contrat est l'outil que nous recherchons, car il vise à organiser les propriétés d'un ensemble d'entités pour garantir leur collaboration, de sorte qu'il soit possible de désigner un responsable en cas de dysfonctionnement.

Pour guider notre démarche nous avons identifié cinq propriétés que doit posséder notre outil. Les deux premières sont issues de ce que nous comprenons comme étant la validité d'un assemblage. Nous considèrons en effet qu'un assemblage est valide si les interactions qu'il abrite se déroulent correctement, c'est-à-dire que chaque partie reçoit des autres ce qu'elle requiert et leur fournit en retour ce qu'elles en attendent. Or pour ce faire, si nous observons les spécifications qui pour chaque composant décrivent ses échanges avec son environnement, l'outil envisagé doit réifier (pour raisonner dessus) et évaluer (pour les vérifier) :

1. La conformité : la conformité des composants, pris individuellement, à leurs spécifications (en intégrant les outils, fournis, associés à leurs formalismes) ,
2. La compatibilité : la compatibilité entre les spécifications de composants sur la base de la configuration de leur assemblage (en intégrant les outils, fournis, associés à leurs formalismes),

Par ailleurs, la solution retenue pour diagnostiquer l'origine d'une défaillance dans l'assemblage est de responsabiliser les composants par rapport au bon déroulement de leurs interactions. Ainsi nous fournissons à l'utilisateur de notre outil une approche uniforme des erreurs. Le contrat de garantie de l'assemblage devra donc :

3. Réifier les responsabilités : c'est à dire la responsabilité de chaque composant par rapport à la spécification de ses échanges avec son environnement,

Pour s'assurer de la validité d'un assemblage et donc son bon fonctionnement, il faut en spécifier chacun des échanges vis à vis de dimensions variées (Qos, comportements,...). Le contrat doit donc présenter les trois propriétés précédentes indépendamment du formalisme des spécifications, c'est à dire :

4. Accepter conjointement des formalismes variés tout en permettant d'assurer les autres propriétés,

Finalement nous souhaitons que le contrat puisse être appliqué à différents systèmes de composants, services ou objets, d'où la cinquième propriété :

5. Indépendance du contrat par rapport au type d'élément contraint, du moment qu'il est possible de raisonner sur l'assemblage des éléments considérés, de manière uniforme.

Ainsi le principe de l'outil de validation d'un assemblage repose sur la compatibilité et le respect des engagements réciproques de ses participants vis à vis de leurs interactions. Or ce principe est exactement celui du contrat de la vie courante. Ce dernier nous a donc servi de métaphore pour la définition de notre contrat et de guide pour l'organisation des spécifications car il propose un ensemble de concepts facilement appréhendables. "Participant", "clause", et "accord" constituent les concepts génériques vis à vis desquels l'assemblage et sa validité son interprétés et décrits.

Notre outil va ainsi consister en un modèle de contrat et un framework associé le concrétisant. L'ensemble pourra être vu comme un bus de validation d'assemblage. En effet, ils définiront les principes et interfaces standards via lesquels ils pourront uniformément prendre en compte conjointement différents formalismes et architectures. Ils organiseront ainsi la mise en oeuvre des spécifications et de leurs outils pour l'architecture à vérifier, de sorte que soit réifiées et évaluables les notions de conformité, compatibilité et responsablité. Dans un premier temps, nous appréhenderons la conformité et la compatibilité de la manière suivante :

- *la conformité* :

  la non-contradiction entre la réalisation d'un élément et la spécification qui le décrit. Nous acceptons que cette non-contradiction soit évaluée pour une forme affaiblie de la spécification, qui au lieu d'être l'objet d'une preuve exhaustive, est vérifiée pour des observations partielles de l'élément (à l'exécution par exemple) ou encore pour un sous ensemble de ses états possibles,

- *la compatibilité* :

  la non-contradiction de spécifications qui, attachées chacune à des entités distinctes, contraignent des grandeurs communes. A nouveau nous acceptons une forme affaiblie de cette expression qui autorise son évaluation non plus par preuve, mais sur la base d'observations de l'exécution du système, ou de sous-ensemble de ses états possibles,

Pour alléger le texte, nous utiliserons, dans le reste du mémoire, ces termes sans rappeler les affaiblissements qu'ils peuvent appeler du point de vue des spécifications formelles. Nous proposerons aussi des définitions de ces propriétés, telles que dans le cas où leur preuve ou leur vérification exhaustive ne sont pas possibles, leur vérification sur la base d'observations du système soit réalisable.

Organisation du manuscrit. Dans un premier temps, afin d'appréhender son cadre d'application, nous allons étudier dans l'état de l'art les systèmes auxquels notre modèle de contrat devra s'appliquer ainsi que divers formalismes utilisés pour décrire leurs propriétés (chapitre 2). Nous détaillerons aussi à cette occassion différentes formes de

contrat existantes, afin d'identifier les problématiques qui demeurent.

Nous analyserons par la suite, dans le chapitre 3, les propriétés attendues de notre modèle. Nous présenterons les choix de conception que nous en avons déduit. Et finalement nous décrirons notre modèle de contrat. Pour ce faire nous en donnerons la constitution et les principes, de manière indépendante des architectures et formalismes auxquels il s'applique. Pour mettre en oeuvre notre modèle, il est nécessaire de l'interfacer avec une architecture et (au moins) un formalisme. Nous montrerons dans le chapitre 4 comment le modèle se représente ceux-ci de manière générique. Nous exposerons au fur et à mesure de cette partie, la mise en oeuvre du contrat pour une architecture concrète, mais de nature très générale (les composants Fractal), et deux formalismes différents (assertions et contraintes sur des traces d'interactions).

Enfin pour valider notre approche il faut l'envisager dans une situation concrète. Nous étudierons donc la contractualisation d'une application réelle ("communautés instantanées") dans le chapitre 5. Nous présenterons ensuite dans cette partie les grands traits de l'implémentation de notre framework. Puis nous finirons par une discussion sur les avantages et limites de notre solution.

CHAPITRE 2

Etat de l'art

2.1 Introduction

Comme nous l'avons vu dans le chapitre d'introduction, nous souhaitons que notre outil s'applique à des architectures variées. Pour ce faire, nous devrons étudier les différents types d'architecture couramment rencontrés. Il est de plus admis que les propriétés à prendre en compte pour vérifier un assemblage sont multiples. Or d'après notre appréhension de la validité d'un assemblage, nous ne pourrons prendre en compte que les propriétés dont l'expression autorise l'étude de la conformité et de la compatibilité des parties de l'assemblage. Il nous faudra donc distinguer les formalismes qui s'y prêtent. Par ailleurs, nous voulons permettre de raisonner en termes de responsabilité par rapport à la validité des assemblages. L'étude des outils existants qui autorisent cette approche nous sera donc utile. Cette notion de responsabilité a donné lieu à un paradigme de spécifications nommé "assume-guarantee" ([99], [50]) qu'il sera intéressant d'étudier car son étendue pourra nous permettre d'évaluer le champs d'application de notre outil. Finalement, d'autres travaux concernant la modélisation de contrat ont été menés, nous tenterons à travers leur étude comprendre comment la question de la garantie des assemblages telle que nous la posons se présente encore.

Ainsi nous allons dans un premier temps étudier les assemblages basés sur trois des types d'éléments les plus courants actuellement (partie 2.2) : les objets, les composants, les services. Puis nous envisagerons l'approche classique du contrat logiciel et l'outil à la base duquelle elle se trouve (partie 2.3). Nous ferons par la suite une revue de différents formalismes de spécifications pour étudier comment ils autorisent l'expression et la vérification de la conformité et de la compatibilité (partie 2.4). Enfin nous nous intéresserons au paradigme "assume-guarantee" (partie 2.5). Finalement nous passerons en revue différents métamodèles de contrat du point de vue des objectifs que nous nous sommes fixés (partie 2.6).

2.2 Architectures logicielles

Le contrat que nous envisageons, a pour principe de garantir une architecture par la validation de la collaboration qu'elle supporte. En effet, cette collaboration est constituée d'un ensemble d'échanges qui sont définis par les relations qui relient ses participants. Nous choisissons de nommer "**motif d'interaction**" l'ensemble des relations qui définissent les échanges de la collaboration. Ces relations identifient une partie d'architecture à laquelle un contrat peut s'appliquer.

Nous étudions dans cette partie trois types d'architectures qui reposent sur trois types d'éléments différents : les objets, les composants et les services. Pour ce faire, nous passerons en revue chacun d'entre eux du point de vue de ses entités, motifs d'interaction, de ses interactions, des propriétés de ses relations et de leur cycle de vie.

2.2.1 Systèmes à objets

A. Entités

Il existe de nombreuses définitions des objets, l'une d'entre elles est donnée par Booch [14] : "Un objet a un état, un comportement et une identité ; la structure et le comportement d'objets similaires sont définis par leur classe commune ; les termes instances (de la classe) et objet sont interchangeables." L'état de l'objet est contenu dans ses "attributs" ou "champs", des variables, tandis que son comportement est implémenté par ses "méthodes", le code qui modifie ces variables. En théorie, un objet ne peut envoyer de données à un autre qu'en invoquant une de ses méthodes mais les langages objets permettent souvent de contourner cette règle (en donnant un accès direct aux attributs). Les classes peuvent avoir des sous classes, qui héritent de leurs attributs et méthodes, mais peuvent aussi y ajouter ou y substituer les leurs.

B. Motif d'interaction

Dans les systèmes objets à base de classes, les entités impliquées sont de deux types, les classes et leurs instances. On distingue alors deux catégories de motifs d'interaction, le graphe d'héritage et le graphe des objets (instances de classes). Le graphe d'héritage existe de manière statique indépendamment de l'exécution du programme, il est composé des relations d'héritage. Le graphe d'objets repose sur les relations d'association et aggrégation, et n'apparaît qu'à l'exécution. La description des graphes d'héritage et d'objets fait partie du code source, sous forme de lien d'héritage ou de références. Par défaut, les relations ainsi définies entre objets ne comportent qu'assez peu d'information, essentiellement le type acceptable. Des motifs d'interaction remarquables sont proposés via les *design patterns* [39] qui identifient classes et objets en leur donnant un rôle dans un assemblage donné. Ils définissent ainsi des collaborations génériques.

C. Interactions

Dans les systèmes à objets les interactions basiques consistent en appels de méthode, ou des accès aux attributs. Vis à vis de ces interactions, on peut remarquer que ces deux graphes sont complémentaires, le graphe d'héritage définit les interactions entre classes parentes et enfantes alors que le graphe d'objet décrit quelles instances peuvent communiquer.

D. Propriétés du motif d'interaction

La relation d'héritage comporte des propriétés contraignant à la fois la construction du graphe d'objets et la réalisation des interactions. En effet tel que le rappelle B. Meyer [71], une utilisation efficace de l'héritage fait intervenir :

- le polymorphisme et la consistance de type qui fixent des règles de production du graphe d'objets : une référence peut être mise à jour à l'exécution et pointer vers tout objet dont le type est un sous-type du sien,
- la redéfinition et la liaison dynamique qui jouent sur la réalisation de l'interaction : la méthode appelée dépend du type de l'entité dynamiquement référencée.

Les interactions sont en général synchrones, mais peuvent être asynchrones et suivant les modèles d'objets des signaux peuvent être définis. La résolution de la méthode à exécuter peut aussi varier sur la base de son nom, de sa signature, du modèle d'héritage.

E. Cycles de vie

Pour ce qui est des cycles de vie des relations, celle d'héritage est définie au moment de la conception des classes. C'est une relation qui ne peut être modifiée une fois le système instancié, et sa durée de vie est identique à celle d'une classe. Les relations d'association et d'aggrégation, définies en conception, peuvent apparaître et/ou disparaître ainsi qu'être modifiées au cours de l'exécution de l'application.

2.2.2 Systèmes à composants

Par rapport aux objets les composants se veulent d'une granularité moins fine et font apparaître explicitement leurs dépendances. Les applications à base de composants sont assemblées et non plus développées.

A. Entités

Il existe de nombreuses définitions des composants. Une d'entre elles, communément admise, est celle de C. Szyperski [101] :

> Une unité de composition avec des interfaces spécifiées contractuellement et seulement des dépendances explicites vis à vis de son contexte. Un composant logiciel doit pouvoir être déployé indépendamment et est l'objet de la composition par des tiers.

Plus programmatiquement, les composants sont en général des structures de données et de code se présentant sous forme de boites noires. Ils n'interagissent avec leur environnement que via les interfaces qu'ils exposent, qu'ils requièrent ou fournissent.

B. Motif d'interaction

Dans les systèmes à composants, les entités impliquées sont essentiellement les composants et leurs connecteurs, organisés dans une architecture généralement explicite. Les relations architecturales mises en oeuvre sont le plus souvent de deux types, la connexion entre interfaces requises et fournies (éventuellement via un connecteur) et l'inclusion de composants dans un autre. Le motif d'interaction de composant mêle les deux types de relations. La connexion peut avoir lieu directement entre composants comme dans l'ADL[1] Darwin [47], ou entre composant et connecteur comme dans le système SOFA [34]. L'inclusion peut être réalisée par un composant en contenant d'autres, ainsi que par des connecteurs potentiellement composites comme dans le modèle SOFA [30]. Le modèle de composants Fractal [17] présente la possibilité de composition partagée, en effet dans celui-ci un composant peut appartenir à plusieurs composites qui ne sont pas imbriqués mais se chevauchent (pour modéliser par exemple des ressources

[1] Architecture Description Language

partagées). L'architecture d'un système de composants peut être décrite soit dans le code métier des composants, soit dans un fichier de description utilisé pour produire le système (ADL). Dans ce cadre les relations sont plus riches en information qu'entre objets. En particulier la connexion est au minimum déterminée par les deux interfaces qu'elle relie. Suivant les systèmes de composants, des contraintes diverses peuvent porter sur ce couple d'interface vis-à-vis de leur connexion [70]. L'utilisation d'un connecteur peut enrichir le schéma de ses caractéristiques, il permet en particulier de désigner par des rôles les entités qu'il met en relation [38].

C. Styles architecturaux

Les styles permettent de décrire l'architecture en termes de motifs remarquables, par exemple [62] :

Pipe and filter	Modèle de traitement des données par flux, les composants agissent soit comme des pipes, soit comme des filters,
Layered	Une couche ne communique qu'avec celle adjacente,
Blackboard	Les composants échangent des données par l'intermédiaire d'une structure partagée (le black-board)
Client-server	Les composants sont soit des clients, soit des serveurs
Process control	architecture fonctionnant à l'aide de sensors et actuators, mise en oeuvre de rétroaction etc.

Les caractéristiques essentielles des styles architecturaux sont définies par [93] :

- Description de la structure du système à l'aide de termes issus d'une ontologie de haut niveau d'abstraction vis-à-vis des éléments mis en oeuvre et de leurs interactions. Par exemple, utilisation des termes "clients" et "serveurs" pour désigner des types de composants donnés,
- Richesse des interactions décrites dans les styles architecturaux, elles sont plus que de simples appels de méthode, par exemple : "*pipe*" avec des conventions de circulation, "*event broadcast*", accès aux bases de données, transaction etc,
- Expression de propriétés globales : la conception de l'architecture s'occupe généralement de propriétés à l'échelle du système, par exemple la résistance d'une partie d'un système à la panne d'une autre etc...

Par comparaison avec les patterns d'objet, la différence entre les styles et les design-patterns exprimée par [93] tient en quelques points :

- les interactions présentées dans les styles sont plus riches et complexes que celles entre objets,

- la description des composants est plus riche que celle des objets : plusieurs interfaces (rôles), requises ou fournies,
- les styles se rapprochent surtout plus d'un langage de pattern que d'un simple modèle de composition, en effet les auteurs font remarquer qu'il existe des ensembles d'utilisation idiomatiques nommés micro architectures ou architectural design pattern à l'intérieur d'un style.

Néanmoins les auteurs notent aussi que certains design-patterns peuvent s'adapter à l'architecture de composants. Mettant de côté les design-pattern de modélisation de données, ou d'algorithmes, ils notent que certains tels la Façade, l'Observer, la Strategy... sont utilisables au niveau des composants.

Divers styles peuvent dans un même système se composer suivant trois voies décrites par [62] :

- locale : les styles varient d'un endroit à l'autre de la configuration sans se superposer,
- hiérarchique : des éléments appartenant à un système d'un style donné, peuvent eux même être organisés internement suivant un autre,
- simultanée : plusieurs styles peuvent servir la description d'un même système : c'est à dire que des composants peuvent avoir plusieurs rôles simultanément,

Enfin certains modèles de composants permettent de définir des styles architecturaux ou pattern de configuration, comme étudié dans [38] :

- Aesop : un style est défini par le sous-typage d'éléments architecturaux (composant, connecteur, etc),
- Wright : un style est composé d'un ensemble de types de composants, de types de connecteurs, de contraintes (prédicats) sur configuration,

D. Interactions

Elles consistent en appels de services, émissions d'événement suivant divers protocoles, passage de flux etc... ainsi elles sont en général de plus haut niveau et plus évoluées qu'entre de simples objets. Certains ADL réifient les connecteurs mais sans nécessairement en tirer tout le parti possible. En effet comme ces derniers sont les acteurs des interactions, les propriétés de la relation entre composants dépendent directement de leurs caractéristiques. Une idée de l'importance qu'ils jouent est donnée dans [30] et une intéressante taxonomie en est faite dans [78]. L'importance des connecteurs dans SOFA [30] fait de ceux-ci des outils de la distribution de l'application et de l'interaction non-anticipée lors de leur conception de composants. En effet ils sont décrits comme le lieu de l'absorption des adaptations nécessaires à la mise en relation de composants de protocoles différents (techniques, sémantiques etc...), dont l'importance est rappelée dans [8]. Il définit ainsi les rôles essentiels des connecteurs :

- contrôle du transfert et des données de transfert,
- adaptation d'interface et conversion de données,

- synchronisation et coordination des accès : protocoles dans SOFA, " *CSP glue*" dans Wright (voir chapitre 2.4.2.3 sur les "algèbres de processus" dans les formalismes),
- interception de communication (cryptage, compression de données, debug etc),

De manière plus abstraite mais plus générale, une classification des interactions qui peuvent se produire dans un connecteur a été définie [78], ce dernier pouvant en abriter une ou plusieurs. Trois critères sont utilisés :

- la catégorie de service :

Nom	Description
communication	passage de données sans plus de comportement
coordination	passage de contrôle : passage du thread de l'un à l'autre des composants : ex basique = appel de méthode, + évolué : load balancing
conversion	rend compatible : format, type, nombre, fréquence des interactions
"facilitation"	Médiation

- le type de service : la manière dont l'interaction est réalisée

Nom	Description
appel de procédure	effectuent coordination et communication
accès aux données	communication, conversion,
lien dynamique	établissement des canaux de communication utilisés par des connecteurs de plus forte abstraction, pour assurer la sémantique de l'interaction, un exemple en est le bus de communication,
Flux	communication (grandes quantités de données),
arbitrator	quand des composants distinguent la présence des autres mais ignorent leur caractéristiques : l'arbitrator effectue les choix et résout les conflits (facilitation), et passe le contrôle (coordination). ex : scheduler etc...
événement	coordination et communication, mais plus divers (écouteurs etc...) qu'appel de procédure,
adaptor	facilite l'interaction entre des composants non connus pour coopérer
distributor	prend en charge l'aspect distribution de la communication (chemin, technologie, etc...)

- les dimensions du service : Elles représentent les variations architecturales possibles dans l'instantiation des connecteurs, surtout utiles pour les con-

necteurs complexes. Elles incluent les notions de cardinalité, synchronicité, de point d'entrée, d'invocation etc...

E. Propriétés du motif d'interaction

Des contraintes et règles sont appliquées à la production des architectures de composants. Elles peuvent être de nature et de portées diverses. Par exemple, de manière générale, la connexion de deux interfaces requiert la vérification de l'hypothèse de visibilité, ou plus spécifiquement certains langages de description d'architecture permettent de limiter le nombre de connexion à une interface. De manière plus évoluée certains ADL permettent de spécifier des contraintes de natures diverses sur les relations : comportementales, qualitative, concurrentielle.

F. Cycle de vie des relations

Suivant les modèles de composants les relations de connexion et d'inclusion peuvent être dynamiquement reconfigurées ou non. Dans cette optique on distingue en général deux phases dans le cycle de vie de l'application constituée de composant :

- la phase de configuration : les relations entre composants sont créées,
- la phase d'exécution : le code métier des composants s'exécute et les échanges ont lieu via les relations établies entre les composants,

2.2.3 Architecture de services

Les architectures de services ne sont pas une évolution de celles de composants, mais une approche de plus haut niveau de la modélisation des processus métiers.

A. Entités

Tout comme pour les objets et les composants, il existe de nombreuses définitions des services. L'une d'entre elles définit le service comme étant [79] : "un mécanisme pour accéder à une ou plusieurs fonctionnalités, dans le cadre duquel l'accès est fourni par une interface prédéfinie et effectué en accord avec les contraintes et politiques spécifiées dans la description du service". De manière plus concrète, une implémentation largement répandue d'un service consiste en un webservice qui est défini par le W3C comme étant [105] : "Un système logiciel conçu pour supporter une interaction interopérable entre machines via un réseau. Il a une interface décrite dans un format interprétable par une machine (WSDL dans ce cas). D'autres systèmes interagissent avec les web services de la manière imposée par sa description en utilisant des messages SOAP, typiquement convoyées via HTTP et avec une sérialisation en XML conjuguée à d'autres standards du web."

B. Motifs d'interaction

Différents types de motifs d'interaction apparaissent dans le cadre de l'architecture orientée service. Un motif d'interaction basique est celui entre un service et son utilisateur. Son principe est privilégié par l'architecture de service, qui n'-explicite pas les dépendances du service par rapport à son environnement. Par ailleurs deux autres formes d'organisation des services, plus évoluées, existent : l'orchestration et la chorégraphie [84]. Elles décrivent toutes les deux comment

produire un processus en combinant les actions de différents services. Elles le font de deux points de vue différents :

- orchestration : le contrôle du processus est détenu par une seule entité, un autre webservice pour le W3C ([105], p. 33), qui appelle les différents services nécessaires à la réalisation du processus,
- choréographie : le processus n'est plus décrit par les requêtes d'une unique entité aux services impliqués, mais par l'ensemble des messages ([105], p.32) que sont susceptibles (ou obligés) de s'échanger tous les services participants pour le mener à bien. Du point de vue de la choréographie, un service n'est plus seulement décrit par une interface mais par les contraintes (typiquement comportementales) sur l'ensemble des messages qu'il est susceptible de recevoir et d'émettre. Il faut préciser que le définit un modèle global d'architecture de services ([105], p.29). Dans ce dernier, les participants à la choréographie sont des "agents" (entités logicielles) qui requièrent et fournissent des services, ce qui autorise la description du fonctionnement du modèle sous forme d'appels de services échangés par les agents.

De nombreux langages ont été mis en oeuvre pour décrire les orchestrations de services [1], BPEL est l'un des plus répandus dans l'industrie. Pour ce qui est des choréographies, le W3C a spécifié deux langages pour leur mise en oeuvre :

- WSCI : chargé de décrire le comportement d'un service du point de vue de ses échanges de messages avec son environnement,
- WS-CDL : chargé de décrire les échanges de messages (possibles ou imposés) entre les services, d'un point de vue indépendant de chacun des services,

Il existe différentes implémentations industrielles de BPEL, qui semblent plus abouties que celles de WS-CDL. Pour ce dernier des plugins eclipse sont référencés sur la page http ://www.w3.org/2002/ws/chor/cdl-implementations.html, d'autres implémentations existent, mais plutôt à l'état de prototypes comme récemment dans [35].

C. Interactions

Les interactions entre services sont essentiellement réalisées par la réception ou l'émission de messages, bien qu'il soit possible qu'un service réagisse à la modification d'une ressource qu'il partage [79]. Dans le cas des webservices, le format et le mode de transmission des messages sont spécifiés par le W3C (HTTP, SOAP).

D. Cycle de vie des relations

Le cycle de vie d'une composition de services a été exploré [108]. Tout dabord les auteurs distinguent trois types de composition de services :

- la composition "exploratoire" : la composition est générée à la volée, tout comme sont découverts les services qu'elle requiert,

- la composition "semi-fixée" : la composition est spécifiée statiquement mais la liaison avec les services concrets est établie à l'exécution,
- la composition "fixée" : la structure de la composition est définie statiquement, ainsi que les services qu'elle va utiliser.

Puis le cycle de vie de la composition est scindé en quatre parties :

1. la phase de Prévision : détermine l'ensemble des opérations qu'il faudra obtenir et aggréger pour réaliser le service attendu par le client, cette phase ne concerne que les compositions "exploratoires",
2. la phase de Définition : détermine de manière abstraite la composition de service à l'aide de fichiers WSDL de description de services et d'un langage d'orchestration tel que BPEL,
3. la phase de Planification : détermine les services concrets d'après leur description issue de la phase de définition. Puis détermine quand les services vont devoir s'exécuter et les prépare. Durant cette phase la composabilité des services est évaluée et à cette occasion d'autres planifications peuvent être proposées au développeur,
4. la phase de Construction : construit et fournit une composition exécutable de services,

Une relation entre un client et un service peut être réalisée de deux manières :

- en phase d'exécution dans le cas d'une composition "exploratoire" ou "semi-fixée", le code client découvre dynamiquement le service dont il a besoin (accords de courte durée), il peut y avoir négociation automatique, durant les phases de définition et planification,
- en phase de conception, dans la cas d'une composition "fixée" : le développeur du client connaît le fournisseur du service et se dispense de négociation, ou alors le développeur du client découvre par une recherche dans l'annuaire le service qui lui convient (accords de longue durée avec les fournisseurs), il peut y avoir négociation avec le fournisseur de service,

Nous constatons que les relations entre WebServices peuvent avoir des durées de vie variable, bien qu'elles reposent dans tous les cas sur la capacité de découverte des services. Les relations établies par des développeurs humains en phase de conception du système ont de longues durées de vie, car elles ne sont pas sujettes à des modifications dynamiques durant l'exécution du processus. Les relations établies dynamiquement par la machine client durant l'exécution du processus ont une durée de vie plus courte, elles permettent de changer dynamiquement de fournisseur de service. Toutefois l'établissement de ces dernières relations repose sur une confiance accordée au système de choix de services du système client (comme discuté dans la partie 3.6.4.5 de [105]).

2.2.3.1 Conclusion

Il faut noter que le fonctionnement de ces trois architectures repose sur la collaboration de leurs constituants. Il est donc possible de leur appliquer des contrats tels que définis

dans l'introduction . Toutefois, les relations qui supportent les interactions ont des cycles de vie très différents, allant du plus statique (objets) au plus dynamique (services). De plus, dans le cadre des objets et des services, les relations sont soient implicites pour les objets, soient indépendantes des services, dans ce cas décrites et mises en oeuvre dans le cadre d'une orchestration distincte. Dans les deux, cas elles ne peuvent être atteintes qu'à l'aide d'outils ad hoc, alors que l'accès à la description de l'architecture fait partie de l'approche par composant. Ainsi, si le principe des contrats s'applique à ces différentes formes de système, ses mises en oeuvre seront bien différentes.

2.3 Fondation de l'approche contractuelle

Dans cette partie nous allons rappeler les origines du contrat logiciel. Nous nous intéresserons aussi bien à son premier représentant fonctionnel, défini par Bertrand Meyer [71] (partie 2.3.1.1), qu'à ses approches non fonctionnelles (partie 2.3.1.2). Puis nous verrons comment le contrat de Meyer est concrètement mis en oeuvre (partie 2.3.2).

2.3.1 Définition du contrat logiciel

2.3.1.1 La vision fonctionnelle : Meyer et la "conception par contrat"

Les premiers contrats logiciels sont ceux de Bertrand Meyer [71] qu'il a obtenu en s'inspirant de la logique de Hoare et des ADT[2]. Il a fondé sur la base de ces contrats une approche nommée "conception par contrat" destinée à fiabiliser les applications et faciliter leur développement.
Logique de Hoare. En 1969, Hoare ([45], inspiré par les travaux de Floyd) propose un système formel d'axiomes et de règles d'inférence pour raisonner sur la correction des programmes informatiques. Dans son approche toute commande C, peut être caractérisée par le triplet {P} C {Q}, où P et Q sont deux prédicats sur l'état du programme avant (P) et après l'exécution (Q) de la commande C. On nomme P la précondition, et Q la post condition. La sémantique du triplet est que si la précondition est vraie avant l'exécution de l'instruction alors la post condition sera satisfaite après celle-ci. Hoare a défini les règles qui permettait pour chaque type de commande C de trouver la plus faible précondition P de C à partir de la post condition Q. Cette approche permet donc de trouver la précondition d'un programme dont on connait une propriété du résultat et de vérifier ainsi que sa pré et que sa post conditions sont conformes à sa spécification.

ADT. Un ADT est un type abstrait de données, c'est à dire la spécifiction d'une structure de données qui décrit :

- son nom,
- les autres types abstraits de données susceptibles d'intervenir dans sa description,
- ses opérations : comprennent les observateurs, les transformateurs et les constructeurs,

[2] Abstract Data Type [42]

- les préconditions des opérations : des prédicats sur l'état du type abstrait avant l'appel d'une opération,
- les axiomes : des prédicats contraignant les observations fournies par les observateurs sur l'état du type abstrait après application des transformateurs, constructeurs, ou sans transformation,

Lors du passage à la phase d'implémentation les préconditions des opérations deviennent les préconditions des fonctions qui les implémentent. Les axiomes portant sur les résultats des opérations deviennent les post conditions de ces fonctions. L'ADT permet ainsi d'appliquer des pré et post conditions aux opérations sur une structure de données.
Meyer applique l'approche de Hoare et des ADT aux interactions entre objets. Pour ce faire il considère le fonctionnement d'un objet du point de vue de son interaction avec un de ses utilisateurs. Traitant l'objet en structure de donnée, il associe à chaque opération qu'il expose une pré et une post conditions qui constituent son "**contrat**", et en contraignent les paramètres et la valeur de retour. Toutefois, au lieu de considérer seulement l'exécution du point de vue de l'objet, il associe la sémantique suivante aux pré et post conditions : l'utilisateur, nommé Client, et l'objet, nommé Fournisseur, sont liés par un contrat, formé par le couple de pré et post conditions (N.B. : couple qui est lié à l'objet), dans le cadre duquel les obligations et bénéfices réciproques sont les suivants :

	Obligation	Bénéfice
Client	satisfaire la précondition	satisfaction de la postcondition
Fournisseur	satisfaire la postcondition	satisfaction de la précondition

Ce contrat présente l'intérêt d'exprimer les conditions d'utilisation d'un objet en clarifiant les obligations et bénéfices des intervenants, ce qui va dans le sens de la réutilisabilité de l'objet. Comme nous verrons plus tard ce contrat peut encore servir à garantir la validité d'une hiérarchie d'objets.

2.3.1.2 La vision non fonctionnelle

La vision non fonctionnelle d'un système ou d'une application peut couvrir un large ensemble de propriétés : performances, disponibilité, réactivité, fiabilité, sécurité, adaptabilité etc. Dans ce cadre, on parle de contrat quand ces propriétés sont l'objet de clauses, ou d'un accord, et qu'il est possible de distinguer des parties. Il faut toutefois distinguer les expressions de qualité de service en tant que telles (contraintes sur des paramètres non fonctionnels), des expressions des besoins en ressource (contrainte sur une ressource de l'environnement). L'article [49] compare ainsi un éventail des formalismes de spécification des ressources nécessaires à l'application ou au système ainsi qu'une palette de formalismes décrivant des qualités du service. En dehors de ces formalismes, on peut citer deux exemples de mise en oeuvre industrielle de la qualité de service :

ATM :

ATM signifie Asynchronous Transfer Mode [15] et désigne un protocole de transmission de données sur le réseau de télécommunication. Il a la particularité de

prendre en compte certaines propriétés de qualité de service (on nomme "cellule" chaque paquet d'octets échangé) :

- le taux de cellules arrivant avec un bit erroné,
- le taux de cellules perdues,
- le taux de cellules insérées par erreur,
- délai moyen de transmission,
- variation de délai de bout en bout

Ainsi quand un utilisateur veut se connecter au réseau et utiliser le service qu'offre ce dernier, il commence par négocier avec celui-ci les valeurs de ces paramètres. Ceux-ci détermineront alors la qualité du service qu'il pourra en attendre. Suivant sa charge et la disponibilité de ses équipements etc... le réseau pourra proposer différentes valeurs, ou configuration de valeurs, à l'utilisateur. Celles-ci définissent alors le contrat passé entre le réseau et l'utilisateur. Un point intéressant est qu'on voit apparaître ici la notion d'accord entre l'utilisateur et le réseau qui sont les parties prenantes du contrat. Nous pouvons aussi remarquer que si les valeurs des propriétés de QoS sont de la qualité du service pour l'utilisateur, elles correspondent à de la réservation de ressources du côté du réseau.

Services :

De plus en plus les applications d'entreprises sont réalisées d'après le modèle SOA : Service Oriented Architecture, c'est à dire qu'elles sont baties sur la collaboration d'un ensemble de services. La qualité de leur prestation repose donc sur la qualité de celles des services qui les composent. Pour décrire et manipuler les propriétés de QoS des services divers standards ont été développés (WSLA [66], WS-Agreement [7], ...) . De manière générale, les propriétés de QoS de chaque service et les contraintes qui portent dessus sont décrites dans le cadre d'un SLA (Service Level Agreement) qui décrit le contrat passé en un service et son utilisateur. Il faut noter qu'aussi bien l'utilisateur que le fournisseur du service peut être le responsable du respect de ces contraintes (bien qu'en général il s'agisse surtout du fournisseur). Il peut s'agir aussi bien des horaires de disponibilité du service, que du débit d'informations ou requêtes qu'il doit accepter, etc... Quand les services de l'application sont implémentés à l'aide de la technologie des Web-Services, il est possible de leur appliquer des SLA écrits, par exemple, en WSLA.

Ces propriétés peuvent être l'objet d'une négociation et d'un accord entre le fournisseur du service et son utilisateur, dont la norme WS-Agreement (pour les Web-Services) spécifie le protocole d'établissement. Dans ce cadre, utilisateur et fournisseur du service s'échangent une description des propriétés de QoS qu'il fournit et requiert, qu'ils font évoluer, jusqu' à ce qu'ils arrivent à une version qu'ils acceptent tous les deux.

Nous pouvons constater que la qualité de service fait en général partie de la conception des applications et systèmes modernes. Non seulement elle est explicitée, mais elle peut être l'objet de négociation afin de s'adapter à des ressources dont la variabilité devient courante.

2.3.2 Etude de la "conception par contrat"

Dans cette partie, nous allons, dans un premier temps, présenter le langage de développement conçu par B. Meyer et qui implémente son approche par contrat. Dans un second temps, nous nous intéresserons au positionnement de l'approche par contrat par rapport aux propriétés que nous avons retenues pour objectif dans notre introduction.

2.3.2.1 Eiffel

La "conception par contrat" est intégrée dans le langage Eiffel [72] sous la forme d'assertions exprimant les obligations de chacune des parties. Les obligations du client sont placées dans des préconditions, vérifiées à l'entrée de la méthode qu'elles contraignent, les obligations du fournisseurs sont placées dans les postconditions, vérifiées à la fin à l'exécution de la méthode. Enfin des invariants contraignent tous les états observables d'une instance de classe. Ces assertions sont notées "require" (préconditions), "ensure" (postconditions), "invariant" (invariants). Depuis leur expression, il est possible d'invoquer des méthodes, l'implication logique est exprimée à l'aide du mot clé implies, le contenu d'une variable antérieur à l'exécution d'une méthode peut être utilisé après celle ci à l'aide du mot clé `old`. Les assertions peuvent être labellisées par un nom qui sera utilisé pour les désigner en cas d'erreur. Exemple de spécification :

```
class interface
        ACCOUNT

feature -- Access
        balance: INTEGER
                -- Current balance
        deposit_count: INTEGER
                -- Number of deposits made since opening

feature -- Element change
        deposit( sum: INTEGER)
                -- Add sum to account.
require
        non_negative: sum>= 0
ensure
        one_more_deposit: deposit_count = old deposit_count + 1
        updated: balance = old balance + sum

invariant
        consistent_balance: balance= all_deposit.total

end -- class interface ACCOUNT
```

Les responsabilités en cas de non vérification d'une assertion sont distribuées de la manière suivante :

- précondition : le client est responsable,
- postcondition : le fournisseur du service est responsable,

- invariant : le fournisseur de la méthode à la sortie de laquelle l'invariant n'est plus vérifié,

A l'exécution les assertions sont factorisées [23] et évaluées par contexte (méthodes et classes) et catégorie (pre/post/invariant). Si une assertion n'est pas satisfaite, sa catégorie détermine l'endroit où se produit l'exception associée. Une violation de pré-condition génère une exception dans la méthode cliente, de post-condition une exception dans la méthode fournie, un invariant dans la méthode du fournisseur ayant causé la rupture de l'invariant. Les exceptions d'une méthode sont capturées et traitées au sein de la clause rescue et le langage permet de réévaluer la méthode dans laquelle l'exception est capturée à l'aide de la commande retry. Dans ce cas les variables locales peuvent être modifiées avant la ré-exécution et ne sont pas réinitialisées au recommencement de la méthode, par exemple :

```
local
    attempts: INTEGER
do
    if attempts< Max_attempts then
    last_character:= low_level_read_function(f)
else
    failed:= True
end
rescue
    attempts:= attempts + 1
    retry
end
```

Dans cet exemple, la variable locale attempts comptabilise le nombre de tentatives d'exécuter la fonction low_level_read_function(). Chaque exception levée par cette dernière, est rattrapée par la clause rescue, qui incrémente attempts avant de relancer la méthode.
Eiffel permet une configuration d'armement paramétrée par un niveau sur une échelle d'importance et dont la portée sur le code peut être spécifiée. La vérification des assertions d'un niveau donné entraîne celle des niveaux inférieurs : no < require < ensure < invariant < check all. La portée est décrite en terme de cluster (équivalent de package java pour Eiffel), ou d'ensembles de classes prédéfinis. L'héritage de classes supporte la redéclaration des assertions [tutorial]. Une classe fille, peut ajouter des assertions à celle de sa classe parente, tout comme une classe d'implémentation à son(ses) interface(s) ou classe(s) abstraite(s). Les règles mises en oeuvre sont les suivantes :

- invariant : somme des invariants des classes(interfaces) parentes
- require : devient require-else, les préconditions ajoutées sont alternatives (" ou ")
- ensure : devient ensure then, les postconditions ajoutées sont obligatoires (" et ")

Eiffel est une solution pragmatique à certains besoins de contractualisation. Son formalisme est accessible aux développeurs, l'évaluation des assertions peut être désarmée de manière graduelle et sur un ensemble de classes donné. La relation d'héritage du modèle objet auquel s'applique le contrat est prise en compte. Par contre les contrats ne sont pas réifiés et n'offrent pas de services de réflexion, et ils sont statiquement définis directement dans le source de la classe.

2.3.2.2 Propriétés de la conception par contrat

La conception par contrat s'applique à l'approche objet de la programmation. Dans ce cadre une application est constituée d'un ensemble d'objets qui collaborent (comme peut le montrer le diagramme de collaboration d'UML) en s'échangeant des messages, les appels de méthodes et leurs retours. Ainsi la précondition d'une méthode porte sur les paramètres reçus par cette dernière, tandis que la post condition contraint la valeur de retour qu'elle émet. La propriété essentielle qu'exprime la conception par contrat est que les paramètres fournis par un objet client à la méthode de l'objet serveur doivent vérifier sa précondition, et que dans ce cas, la valeur de retour du serveur doit vérifier la post condition de la méthode. La conformité du client et du serveur à la spécification de la méthode, constituée des pré et post conditions, est ainsi la propriété de base vérifiée par la conception par contrat.
Une seconde préoccupation concernant l'héritage des classes s'est faite jour dans la conception par contrat. Dans un premier temps les règles édictées dans le cadre de Eiffel, définissant le contrat de la classe enfant en fonction de celui de la classe parente, ont été reprises. Puis [33] les a affiné pour valider formellement l'expression, sur la base des contrats d'une classe parente et enfant, de la substituabilité de l'une par l'autre. Par ailleurs dans le cadre de la collaboration des objets, un client ne peut envoyer un message à un serveur qu'en passant par une référence sur ce dernier. Or cette référence est nécessairement celle d'une classe enfant du serveur (du fait du polymorphisme). Comme cette descendance est contrainte par les règles de substituabilité du contrat, la compatibilité d'un objet client et d'un autre serveur y est soumise. Ainsi la conception par contrat permet une première approche de l'expression de la compatibilité entre deux objets. Enfin, la conception par contrat affecte les responsabilités à la classe appelante et à la classe appelée vis à vis de leur conformité aux pré et post conditions. Toutefois celles-ci ne font pas partie d'une architecture qui permettrait de les manipuler, et donc la faute retombe sur leur développeur ou fournisseur.
Comme nous l'avons fait remarquer dans l'introduction ces propriétés sont contenues dans l'ensemble de celles fondamentales nécessaires à l'expression de la validité d'un assemblage d'objets, de composants ou de services. Toutefois elles sont l'objet dans le cadre de la conception par contrat d'assez fortes limitations. Ainsi les propriétés de conformité et compatiblité :

- ne concernent qu'un seul formalisme et un seul ensemble d'observations prédéfini (paramètres, valeur de retour d'une opération), or nous avons vu que la validité d'un assemblage dépendait de propriétés et de formalismes variés,
- sont implicites et fondues dans le code de l'application, c'est à dire que l'outil d'évaluation de ces propriétés, qui est le code du programme lui même, ne permet pas de les exposer à celui qui en utilise le résultat. Elles ne sont pas réifiées en objets visibles dont l'état reflèterait leur satisfaction.
- relèvent d'un modèle de spécification plus large, l'approche hypothèse-garantie, sur lequel il est possible de s'appuyer pour valider les interactions au sens large entre des entités de types variés (composants, services...),
- ne concernent que l'interaction entre deux entités, un client et un serveur,

Par ailleurs les responsabilités ne sont pas associées à des objets visibles et manipulables par l'acteur qui met en oeuvre l'application. Dans le cadre de l'approche par

objets, ces responsabilités devraient au mieux être associées à des modules ou des librairies dynamiques qui pourraient être interchangées par l'administrateur de l'application pour répondre à une défaillance détectée par un contrat.

2.4 Formalismes de spécification

Dans cette partie nous allons passer en revue des formalismes décrivant les caractéristiques des architectures que nous avons étudiées dans la partie 2.2. Nous cherchons en effet à évaluer si les propriétés qui nous intéressent, comme l'expression de la conformité, de la compatibilité ou la notion de responsabilité, sont répandues parmi ceux-ci. Nous distinguerons à cette occasion les formalismes dotés d'outils autorisant l'évaluation de ces propriétés. Nous allons séparer les formalismes en deux ensembles, d'une part ceux fonctionnels, d'autre part ceux non-fonctionnels. Dans un premier temps, nous allons observer des formalismes fonctionnels qui s'appliquent aux objets (partie 2.4.1) et aux composants (partie 2.4.2). Puis nous nous intéresserons à des formalismes non fonctionnels portant successivement sur les objets (partie 2.4.3), les composants (partie 2.4.4), puis les services (partie 2.4.5).

2.4.1 Spécifications fonctionnelles appliquées aux objets

Les spécifications fonctionnelles appliquées aux objets reposent en général sur la mise en oeuvre d'assertions. Néanmoins celles-ci peuvent intervenir à différents moments du cycle de vie du système et décrire différentes propriétés.

2.4.1.1 Assertions exécutables

Diverses études ([89],[23],[97]) mettent en relief des traits caractéristiques des mises en oeuvre d'assertions au niveau des langages :

ajout du mécanisme dans le langage lui même :
: Le langage fournit les structures exprimant les assertions et qui sont alors traitées par le compilateur de la même manière que le reste du programme. L'intérêt de cette approche réside en l'intégration homogène dans le langage permettant la prise en compte des assertions de qualité équivalente à celle du langage de base dans les outils associés (compilateur, debugger etc...).

tranformation par un préprocesseur de commentaires ou macros :
: Un préprocesseur produit du code source correspondant et l'intègre dans le code source du programme. Cette méthode a l'avantage de séparer la logique applicative de celle des spécifications mais elle présente l'inconvénient de modifier le code source, les numéros de ligne du code source ne sont plus utilisables tant à la compilation (erreur), qu'au debuggage.

utilisation de la métaprogrammation et de la réflexivité du langage :
: Les assertions peuvent alors se présenter sous une forme exécutable distincte du programme. Leur mise oeuvre repose sur l'étude réflexive, en cours d'exécution, des objets utilisés par le programme. Cette approche évite l'utilisation d'outils de compilation ou précompilation particuliers, ainsi que la modification du code

source, mais suppose de l'environnement d'exécution la prise en compte des assertions.

Java En Java tout un ensemble de travaux mettent en oeuvre l'approche de conception par contrat. Dans le cadre des extensions de langage on retient principalement la Java Assertion Facility et le Biscotti. La Java Assertion Facility [100] consiste dans le mot clé assert fourni par le JDK java depuis la version 1.4. Son évaluation est paramétrable jusqu'à suppression complète de l'exécution. Biscotti [27] modifie les classes du JDK pour vérifier pre/post conditions et invariants au niveau des interfaces Java RMI. La réflexion est utilisée par Handshake [29] et jContractor [52]. Dans les deux cas les assertions sont insérées dans le bytecode de la classe à contractualiser au moment de son chargement. Handshake est un peu plus bas niveau vis à vis de la mise en oeuvre puisque il ne supporte pas le mot clé old dans les postconditions et que le désarmement de conditions se fait par leur mise en commentaires dans leur fichier de description. Dans les deux cas il n'est pas nécessaire de disposer du source des classes auxquelles on souhaite appliquer les contrats. Les préprocesseurs sont les outils les plus utilisés. De manière générale les assertions sont décrites au sein de commentaires java tagués spécialement. Chacun s'attache à la réalisation de propriétés particulières. ContractJava [32] garantit la validité du sous-typage comportemental car concrétisation d'une étude théorique sur le sujet. Dans iContract ([60], [31]) la syntaxe des conditions respecte celle d'OCL. Jass [12] permet d'insérer des assertions dans le corps des méthodes et des invariants de boucle. Il propose de plus un système semblable au retry et rescue d'eiffel. Il permet surtout la vérification de la validité de séquences d'appels. Enfin JContract repose sur le assert fourni par le JDK est susceptible de prendre en compte des gestionnaires d'exception fournis par l'utilisateur.
D'un point de vue pragmatique les approches réflexives semblent les plus intéressantes car elles proposent une instrumentation dynamique du code exécutable sans nécessité de recompilation du code source. D'autre part elles utilisent le langage java comme expression des conditions et sont ainsi accessibles aux développeurs. Enfin si leurs contrats n'offrent pas, pour eux mêmes, de possibilité de réflexion, ils n'en sont pas moins des entités indépendantes des programmes dont elles pourraient néanmoins permettre l'ajustement avant tissage. Toutefois quasiment tous ces contrats ne reflètent que la conformité des objets à leur spécification. Seuls ceux prenant correctement en compte l'héritage offrent une première approche de la compatibilité, sans pour autant réifier ces propriétés ni expliciter les responsabilités.

2.4.1.2 OCL : les assertions en conception

Plusieurs approches mettent en oeuvre les assertions en phase de conception. BON par exemple reposant fortement sur le langage Eiffel utilise largement la conception par contrat en plus de l'approche de conception par objet traditionnelle. Les assertions sont utilisées pour guider la création et la définition des éléments des modèles statiques et dynamiques. Nous nous intéresserons plus précisément au modèle de contraintes définies par OCL de par l'importance des travaux associés à UML.

le langage OCL L'approche contractuelle est fortement liée à l'activité de conception objet. En effet la première repose sur l'expression des conditions de collaborations entre entités. Or l'expression de ces collaborations est à la base de la distinction du

contour des objets par les responsabilités qu'elle amène à faire apparaître. OCL est apparu en 1997 avec la version 1.1 de UML [80]. C'est un langage de formalisation de contraintes d'une modélisation UML qui se veut accessible aux non spécialistes de techniques formelles. Il propose un ensemble de fonctionnalités consistant en l'expression d'invariants de modélisation et d'exécution du modèle. Toute information OCL attachée au modèle UML l'est par l'intermédiaire d'un contexte décrit à l'aide d'une combinaison de classifiers. Chaque classifier spécifie une entité dans l'environnement de laquelle les expressions OCL doivent être interprétées. Les classifiers accessibles comprennent aussi bien les classes, les types, méthodes etc... que les liens entre classes. Ce contexte est complété d'un stéréotype : inv, pre, post décrivant sa portée temporelle. " inv " est associé aux classes, types et Stéréotype, alors que " pre ", " post " s'appliquent aux méthodes ou opérations des classes. " inv " s'étend sur la durée de vie de l'entité, alors que " pre " définit l'instant précédent et " post " l'instant succédant le passage dans une méthode. OCL permet d'associer à ce contexte des contraintes sous forme d'expressions booléennes dont la validité devra être assurée sur celui-ci. En tant que langage, OCL est fortement typé et garantit que toute évaluation d'une de ses expressions est exempte d'effet de bord sur le système contraint. Pour ce faire il n'autorise les appels que des seules méthodes portant l'attribut isQuery = true.
Afin d'être accessible aux développeurs la syntaxe des expressions OCL est typique de langages impératifs traditionnels tel que le C, associée à une sémantique fonctionnelle. Les expressions OCL peuvent faire intervenir les entités définies dans le modèle UML. Il est possible d'accéder à leurs propriétés par la notation pointée, sauf pour les collections ou la flèche est utilisée (collection->size). Des collections génériques sont disponibles avec Collection au sommet d'une hiérarchie faisant apparaître Set, Bag et Sequence.
Exemple de contrainte :

```
context Person::getCurrentSpouse() : Person
     pre: self.isMarried = true
```

Dans la version 1.1 OCL fournit divers opérateurs sur les collections :

- `select : collection->select (var : type | expression-with-var)`: forme une collection de toutes les valeurs de la collection "collection" répondant à la condition expression-with-var,
- `reject` : désigne la collection de laquelle sont supprimés tous les éléments répondants au critère donné,
- `forAll et exists` : sont des opérateurs booléens exprimant une propriétés universelles ou existentielles sur un ensemble d'éléments :
- `self.employee->forAll (prénom = "André")` : retourne vrai si tous les employés s'appellent "André",
- `self.employee->exists (prénom = "André")` : retourne vrai si un employé s'appelle "André",

Les collections sont "aplaties" avant d'être parcourues ce qui empêche l'utilisation de collections de collections. L'opérateur @pre autorise l'accès dans des contraintes post à des valeurs conservées d'avant l'entrée dans la méthode.

La version 2.0 d'UML [81] propose une version améliorée d'OCL. Un mot clé body permet de décrire des méthodes d'interrogations du modèle, par exemple :

```
context Person::getCurrentSpouse() : Person
    pre: self.isMarried = true
    body: self.mariages->select( m | m.ended = false ).spouse
```

la clause body contient la formule évaluant la valeur à retourner.
Il est aussi possible de contraindre l'évolution de la valeur d'attributs. D'autre part les structures de données ensemblistes ne sont plus aplaties lors de leur parcours, ce qui permet de disposer de listes de listes. Des quantificateurs universels ou existentiels portant sur les extensions de classes sont aussi disponibles :

```
   voiture->forAll (couleur == verte)
   voiture->exists (condition)
```

Enfin OCL propose une entité réifiant un message, ainsi qu'un opérateur indiquant son envoi, nommé hasSent, et noté "$\wedge$". Ce dernier permet par exemple de stipuler la nécessité qu'un message ait été envoyé au cours de l'exécution d'une méthode :

```
context trompette.joueMusique :
   post : trompette^sonne (note : integer)
```

dans cet exemple, la méthode joueMusique devra avoir émis un message "sonne".
En outre les messages réifiés d'OCL permettent la gestion de l'asynchronisme, chaque message dispose en effet d'une méthode hasReturn() et d'une méthode result(). Respectivement elles permettent de savoir si une réponse à été faite au message et si oui sa teneur.
Bien que ce langage n'ait pas été prévu pour être exécuté, divers projets ont mis en oeuvre OCL de manière opérationnelle. Plusieurs ([46], [76]) sont décrits dans "l'état de l'art des approches contractuelles" ([23]) auquel on pourra se reporter pour plus de détail, mais dont on peut retenir que ces travaux ont en commun des problèmes d'efficacité par rapport à l'évaluation des conditions. Plus récemment une réalisation reposant la programmation par aspect a été produite [16] qui supporte quasiment toute la syntaxe de la version 1.3 d'OCL (seules l'Enumeration et l'opération OclInState ne sont pas supportées). Les contraintes OCL de la modélisation UML sont dans un premier temps traduites en arbres syntaxiques. Ceux-ci sont par la suite transformés en code java avec l'assistance de requêtes au modèle UML et au programme contraint. Puis AspectJ est utilisé pour leur tissage dans le code du système étudié. Les auteurs présentent des mesures de mise en oeuvre concernant l'évolution des temps d'exécution, de l'empreinte mémoire, et de la taille du programme. Toutefois à nouveau seul la conformité des objets à leur spécification est prise en compte.

2.4.1.3 JML une approche hybride

JML [19] est un Behavioral Interface Specification Language (BISL) dans la lignée de Larch. Ce dernier insère des commentaires dans un code source pour lui associer un modèle algébrique abstrait développé séparément. Le modèle abstrait étant écrit dans un langage commun Larch Shared Langage, cela permet de réutiliser des outils de preuves indépendamment des langages traités. Ainsi la spécification JML se compose de :

- une spécification d'interface qui porte sur les éléments de langages que cette dernière présente, telles les méthodes,

- un description de comportement, ainsi que d'autres propriétés ou contraintes, reposant sur les éléments décrits dans l'interface, exprimée à l'aide d'un modèle,

Ainsi comme pour Larch, la spécification se divise entre une partie abstraite par rapport au programme cible, le modèle formel, et une autre réalisant le lien avec le programme cible. Toujours à la manière de Larch le lien entre le modèle et le programme auquel on l'applique est réalisé à l'aide de pre, post conditions et d'invariants. JML permet donc aussi de décrire des assertions classiques de l'approche par contrat en java. Du point de vue du formalisme, JML utilise java pour décrire le modèle et se présente donc comme accessible au développeur. Il offre une approche de la généricité de la description du comportement des objets grâce aux variables du modèle abstrait. Néanmoins il n'offre pas d'outil de réflexivité, et n'observe que la conformité des objets à leur spécification.
JML présente la particularité d'être à la fois exécutable et statiquement vérifiable. Pour ce qui est de la vérification dynamique, il existe un compilateur jmlc [64] qui prend en compte les spécifications JML qui apparaissent sous forme de commentaires dans le code source. Ce compilateur prend en compte les conditions de sous-typage comportemental (combinaisons des conditions), gère les expressions ensemblistes (quantificateurs universels et existentiels), les possibles cas d'indétermination (exception à l'évaluation d'une condition), la prévention d'effet de bord à l'aide d'un système de typage des entités de la spécification.
Du point de vue des outils statiques il en existe différents types ([18], auquel on pourra se reporter pour plus de détails). Il y a d'un côté les vérificateurs statiques, mais aussi des outils de test et d'assistance à la production de spécifications qui sont présentés en annexe. L'utilisation de JML a lieu à des moments variés du cycle de vie de l'application. Il est remarquable que pour chaque tâche associée à la spécification des outils existent permettant de l'automatiser au moins en partie. La définition des spécifications, puis leur vérification statique partielle, la production d'outils de tests d'exécution et finalement la compilation des assertions et modèles en code exécutable sont assistées.

2.4.1.4 Bilan

Les assertions sont focalisées sur la vérification de la conformité des objets à leurs spécifications. Elles ne réifient toutefois pas cette propriété car ce ne sont pas des entités de premier ordre. Elles n'expriment ni la compatibilité ni la responsabilité de leurs porteurs. Bien que modulaires, elles sont systématiquement rattachées à l'objet serveur et jamais à l'objet client. Pourtant il serait simple d'évaluer la compatibilité de deux assertions, portées par le client et le serveur, en observant si pour une grandeur qu'elles contraignent en commmun, elles sont simultanément satisfaites. Néanmoins dans les systèmes à base de hiérarchie de classes, il a été défini, pour les assertions, une notion de compatibilité entre classes parentes, qui étendue à la notion de sous typage, permet de valider les relations de référencement entre les instances (une variable ne pouvant désigner qu'une instance d'un sous type du sien). Mais cette propriété n'est pas réifiée, et n'autorise pas la spécification du client.

2.4.2 La conception par contrat appliquée aux composants

Comme nous avons vu (2.2) la notion de contrat fait partie de la définition des composants. Nous allons voir dans cette partie comment la conception par contrat classique est appliquée dans le cadre des plateformes à composant industrielles. Puis nous observerons comment elle est étendue dans le cas d'une plateforme de composants plus classiques. Enfin nous nous intéresserons à l'utilisation de formalismes formels de spécification.

2.4.2.1 Plate-formes industrielles

.Net Différents travaux tendent à intégrer l'approche par contrat dans ".Net". Nous avons retenu deux approches qui mettent en oeuvre différemment les assertions.
L'intégration la plus traditionnelle est réalisée par le ContractWizard ([53]). Cet outil tire parti de l'intégration de Eiffel dans la plateforme " .Net ". Il permet d'appliquer des pré/post conditions et invariants sur les classes des langages à objet supportés par .Net, en produisant pour chacune d'entre elle un wrapper en langage Eiffel. Néanmoins Tran et Mingins font remarquer ([103]) que l'approche précédente manque d'homogénéité (entre assembly décorés et pas décorés par exemple) par rapport à l'intégration de la gestion des pré/post conditions, invariants directement dans la machine virtuelle. Dans ce dernier cas, le traitement des spécifications à l'exécution est alors uniforme pour tous les langages. Un bout de code dans un langage A est apte à reconnaître une erreur d'assertion issue d'un bout de code en langage W. Enfin la prise en compte est paramétrée dynamiquement au niveau de la machine virtuelle. Ces deux approches sont détaillées en annexe.

J2EE Une utilisation classique de l'approche par contrat appliquée aux EJB est décrite par [104]. Dans le cadre du développement de High Confidence Software (HCS), les auteurs souhaitent placer les services nécessaires à la sûreté de leurs applications dans le container EJB pour minimiser les coûts de test, d'évolution de leurs logiciels dont la partie évolutive se retrouve dans des beans. Le container EJB est modifié pour inclure des intercepteurs dans les interfaces EJB qu'il produit, et le serveur (JBoss) est monitoré à travers son interface JMX. La vérification des pré/post conditions est déclenchée par les intercepteurs en amont et aval de l'exécution des services des EJB. Les invariants sont exprimés au niveau de l'application et leur évaluation peut être effectuée régulièrement dans le temps à l'aide des fonctionnalités JMX, ou bien déclenchée par des intercepteurs. Des détails sur ces travaux sont donnés en annexe.

Les démarches exposées mettent en oeuvre la conception par contrat tant dans .Net que dans J2EE. Néanmoins l'exemple de J2EE expriment ses invariants au niveau de la modélisation JMX de l'application et non plus au niveau des interfaces des objets ou composants. Elles se distinguent par ailleurs par leur trois niveaux de mise en oeuvre, les wrappers Eiffel de .Net se placent au niveau du programme, tandis que la mise en oeuvre dans J2EE se fond dans le middleware de la plateforme et que enfin Tran et Mingins modifient directement la machine virtuelle. Elles présentent ainsi des niveaux de dynamicité et d'abstraction décroissants, sans pour autant refléter d'autre propriété que la conformité des composants à leurs spécifications.

2.4.2.2 CCLJ

CCLJ [24] est un langage d'assertions (pre, post, invariant...) dont la syntaxe est partiellement inspirée d'OCL, et adaptée au système de composants Fractal. Il se distingue des assertions classiques d'OCL sur deux points :

positionnement des assertions : une assertion n'est plus associée simplement à une méthode comme dans OCL, mais à une méthode d'une interface d'un composant donné, cela se traduit par des expressions du type :

```
on <CruiseCtrl> context sns.on () :
  pre : {contrainte OCL}
```

pour décrire une précondition sur la méthode `on()` de l'interface `sns` du composant `CruiseCtrl`.

portée des assertions : étant associées à un composant, les assertions ne se limitent plus à contraindre les grandeurs liées à une méthode, mais elles peuvent contraindre celles d'autres méthodes d'autres interfaces du composant auquel elles sont associées. Dans cette optique, les assertions sont classées en trois catégories suivant leurs portées, c'est à dire les entités architecturales apparaissant dans la "contrainte OCL" :

- spécification d'interface : l'assertion peut comporter des appels aux autres méthodes de l'interface possédant la méthode sur laquelle elle porte,
- spécification externe : l'assertion peut comporter des appels aux méthodes des interfaces externes du composant porteur de l'interface exposant la méthode qu'elle contraint,
- spécification interne : l'assertion porte sur une interface d'un composant composite et elle peut comporter des appels aux méthodes des interfaces : a) internes du composites, b) externes des sous composants,

Les outils associés à CCLJ permettent de produire les assertions au fur et à mesure de l'évolution du système de composants sur la base de fichiers de spécifications distincts des ressources architecturales. Par contre ils limitent leur vérification à la satisfaction des assertions vis à vis de l'exécution du système, elles ne sont pas comparées. Plus de détails sur le langage sont disponibles en annexe.

2.4.2.3 Utilisation de spécifications formelles

De manière générale les formalismes formels relèvent de niveaux d'abstraction qui rendent délicats leur mise en oeuvre. Ils sont exigeants tant vis à vis de leur utilisateur que des outils et techniques de vérification. Néanmoins nous étudions quatre catégories d'entre eux :

- la spécification par automates (partie A),
- les algèbres de processus pour l'importance qu'elles accordent aux interactions (partie B),
- les spécifications boites grises (partie C),

- les logiques de description pour le raisonnement qu'elles proposent sur des propriétés génériques des éléments (partie D),

L'utilisation de ces formalismes ne nécessite pas comme pour d'autres la spécification complète du système. Les propriétés qu'ils définissent peuvent s'ajouter et se composer. Enfin ils permettent des traitements statiques en amont de l'exécution des entités qu'ils décrivent. En particulier, hormis les spécifications boites grises, ceux cités autorisent la comparaison de spécifications.

A. La spécification par automates : CoCoNut/J Les automates sont un outil commun de représentation du comportement d'entités logicielles. Il semble donc naturel de les voir utilisés dans le cadre de la spécification de comportements. Les EJB, profitant de la réflexivité de java, sont le cadre de la mise en oeuvre d'un de leurs représentants : CoCoNut/J, utilisé pour l'adaptation du comportement des interfaces [91]. Dans ce cadre, l'approche par contrat est complétée par une modélisation du comportement des interfaces par des automates qui sont l'objet effectif de l'adaptation. Chaque interface est décrite par deux types d'automates :

- l'automate d'appel : définit un sous ensemble de toutes les séquences d'appel valides sur l'interface (Figure 2.1),

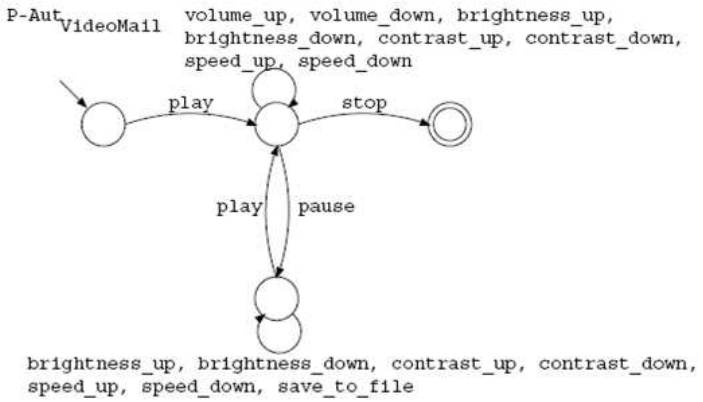


FIGURE 2.1 – Automate d'appel

- l'automate de fonction : définit pour chacune des fonctions de l'interface toutes les traces possibles d'appel à des services extérieurs à son composant porteur (Figure 2.2),

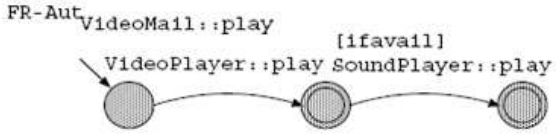


FIGURE 2.2 – Automate de fonction

Ainsi quand un composant A s'adapte à un composant B, son automate d'appel restreint ses fonctionnalités à celles compatibles avec les automates de B. Ainsi le système

peut adapter les fonctionnalités offertes d'un composant au moment de son insertion. A l'époque de l'article les auteurs proposaient de produire les automates à l'aide d'annotations ajoutées aux signatures des méthodes. Le prototype ne semble pas avoir été développé. Cette formalisation du comportement des composants permet d'envisager la conformité de ces derniers à leurs spécifications, ainsi que la compatibilité de ces dernières.

B. Les algèbres de processus

description

Dans sa " brève histoire des algèbres de processus " ([11]), Baeten présente les concepts fondamentaux des algèbres de processus. Tout d'abord, il définit un processus comme un système auquel est associé un comportement décrit sous forme d'un ensemble d'événements ou d'actions discrets. Sur cette base il décrit l'algèbre de processus comme une structure mathématique qui, de la même manière que les groupes en algèbre, définit un ensemble d'entités (les processus) stable pour des opérateurs de composition tels que le choix entre deux, la mise en séquence. Beaten fait remarquer que si la définition s'arrêtait là, elle pourrait aussi bien s'appliquer aux automates, qui sont des processus dont le comportement est l'exécution, et dont une algèbre pourrait alors être celle des expressions régulières. C'est la notion fondamentale d'interaction entre processus qui fait passer les algèbres de processus de la théorie des " processus " à celle de la " concurrence ". L'opérateur de mise en parallèle de processus et la communication deviennent les concepts essentiels des algèbres de processus. Pour finalement devenir un outil de description de la sémantique de systèmes interagissant de manière générale, la théorie des AP a eu besoin de deux évolutions par rapport aux concepts utilisés en sémantique à l'époque :

- le programme n'est plus considéré comme une fonction avec une entrée et une sortie mais comme une entité dont les états intermédiaires sont pris en compte,
- les variables globales sont abandonnées au profit de variables locales et d'échanges de messages.

Ces pas ont été franchis dans le cadre des Communicating Concurrent Systems (CCS) et Communicating Sequential Process (CSP). Baeten explique qu'ainsi " les algèbres de processus sont devenues une théorie applicable à tout système parallèle ou distribué, en ajoutant au langage formel et à la théorie des automates la notion fondamentale d'interaction ". Ainsi une algèbre de processus peut être vue comme un ensemble de processus, supportant les opérateurs composition en parallèle (" || "), séquence (" ; "), et alternative (" + ") tels que :

```
1.        x+y = y+x
2.        x+(y+z) = (x+y)+z
3.        x+x = x
4.        (x+y);z = (x;z)+(y;z)
5.        (x;y);z = x;(y;z)
6.        x || y = y || x
7.        (x || y) || z = x || (y || z)
```

ainsi que des opérations d'émission et réception de message entre processus, soit en spécifiant le destinataire (CSP...), soit en utilisant des canaux (pi-calcul etc...) ...
Comme l'explique [96], l'approche compositionnelle que les algèbres de processus proposent, a favorisé leur utilisation dans la formalisation et la vérification de comportement de modèles complexes. En effet l'étude de relations d'équivalences entre des modèles, peut dans certains cas se ramener à celle des composants constitutifs des systèmes.
Par essence les spécifications à base d'algèbre de processus peuvent se modéliser à l'aide d'automates, dont les changements d'état sont attachés aux messages échangés entre les processus. La conformité du code à sa spécification peut se faire via son instrumentation soit par des clauses traditionnelles d'assertions, soit en faisant intervenir des outils de rewriting logic. La compatibilité des spécifications peut être déterminée à l'aide de prouveurs ou de modelchecker.

mise en oeuvre des algèbres de processus

Les algèbres de processus ont été mises en oeuvre dans plusieurs ADL. En effet les composants alors vus comme des processus indépendants s'échangeant des messages se prêtent bien à cette modélisation. SOFA, Wright et Darwin sont trois représentants de cette approche. Ainsi l'ADL Wright utilise les CSP pour spécifier le comportement de ses d'interfaces [38], tandis que Darwin est un ADL qui base la sémantique de ses descriptions structurelles sur le pi-calcul [47]. Ils ont tous les deux en commun d'accepter des outils qui vérifient la compatibilité de leurs descriptions des composants. Une description détaillée de ces deux approches est donnée en annexe.

SOFA est un modèle de composants hiérarchiques présentant classiquement un ensemble d'interfaces requises et fournies. La communication entre composants est formellement décrite par des Behavior Protocol [88]. Ceux-ci reposent sur le principe que chaque appel ou réception d'appel de méthode constitue un événement. Le behavior protocol d'une entité SOFA consiste alors en l'ensemble des traces de ces événements. Les entités architecturales considérées sont de deux natures : la "frame" qui est l'ensemble des interfaces exposées (requises et fournies) par un composant, l'"architecture" qui est l'ensemble des interfaces exposées par un assemblage de composants. Le comportement d'un composant pourra par exemple être décrit de la manière suivante :

```
frame F {
   provides:
     I i1;
   requires:
    J i2;
  protocol:
    ?i1.m; (!i2.n + (?i1.n; ?i1.n));
};
```

Dans cette expression, "!" et "?" dénotent respectivement les appels et les réceptions d'appel de méthode. "+", ";" représentent l'alternative et la séquence d'événements. Les outils associés aux Behavior Protocol permettent de vérifier :

- de manière statique et dynamique, la conformité du comportement d'un composant à sa spécification,

- de manière statique, la validité de la composition de composants, c'est à dire leur compatibilité, du point de vue de leurs comportements. Ceci concerne aussi bien la compatibilité entre des composants du même niveau hiérarchique (via un opérateur de composition parallèle $\cap$), qu'entre un composite et ses sous composants (via un opérateur de composition verticale $\underline{compl}$),

Bilan

La mise en oeuvre des algèbres de processus en tant que telles est assez complexe. Néanmoins des formes dégradées de cette spécification deviennent plus abordables quand elles se fixent l'objectif de spécifier essentiellement le séquençage des échanges de messages tels que Wright, Darwin, SOFA ou Coconut/J. L'avantage de la plateforme SOFA est de mettre en oeuvre cette approche dans un environnement plus avancé en terme de génie logiciel que Darwin ou Wright qui restent assez théoriques.

C. Les spécifications boites grises Les spécifications par algèbres de processus mettent en relief un besoin de clarté de spécification décrit par les spécifications "boite grise" ([69]). En effet les spécifications boite noires y sont critiquées comme trop opaques, vis à vis des comportements, alors que les spécifications boite blanche ne sont pas assez abstraites pour être réutilisables. Les spécifications boites grises ont pour particularité de ne plus considérer le déroulement d'une méthode comme atomique. Sans lever l'hypothèse de visibilité sur le contenu du composant, les messages, entrant et sortant durant l'exécution d'une opération, peuvent être spécifiés. Cette hypothèse est importante comme le montre l'exemple de spécification de l'appel d'un call-back au sein d'une opération qui ne peut pas être effectuée à l'aide de spécifications boite noire traditionnelles.
Barnett et al. proposent une approche par modèle ([73] détaillée en annexe), ce dernier étant décrit en un langage nommé Asml : Abstract State Machine Language. Ces programmes "modèles" peuvent être utilisés seuls en tant que simulation ou associés aux programmes pour vérifier leur conformité à leurs spécifications. Les auteurs motivent plus précisément l'utilisation d'un modèle Asml, par le fait qu'il est réutilisable, permet la prise en compte des séquences de messages et du non déterminisme de certaines spécifications.

D. Les formalismes logiques L'intérêt des formalismes logiques est de permettre le raisonnement sur les spécifications. Différents modèles tentent d'établir leur compatibilité, ou leur équivalence. Un intérêt non négligeable de ces approches est qu'elles sont génériques. En effet elles sont basées sur des descriptions, celles des entités et de leurs propriétés.

PECOS

Des règles PROLOG sont mises en oeuvre dans le cadre du projet Pecos [41]. La validité statique d'un modèle est exprimée à l'aide d'un ensemble de règles inhérentes au modèle de composants et d'un autre correspondant aux spécifications de chaque composant. Ces règles reposent sur des données statiques de chacun des composants (propriétés, identifiants etc...). Deux types de règles sont distinguées :

- les règles de consistance : "si le composant a un champs X alors il doit avoir un champs Y ",

- les règles de compositions : qui vont de la contrainte structurelle au style architectural,

Dans ces travaux les règles ne remplacent pas les contrats "traditionnels" de la conception par contrat qui eux sont réservés à une évaluation dynamique. Néanmoins il est intéressant de constater que dans le cadre de la composition, la mise en commun de propriétés est résolue statiquement.

Description Logic et Kind Theory

Un autre type d'approche par règles est concrétisée par la description logic [5] ou par la kind theory [57]. Dans ces dernières, des informations "du domaine du discours" sont associées aux composants, interfaces, méthodes etc, et un service de raisonnement sur ces informations est fourni. Ainsi dans [5] les auteurs se proposent d'augmenter les informations contenues dans l'IDL, de données, décrites à l'aide et manipulables par des langages de Concept Description Language/Logic (les " DL "). Ces langages sont issus de travaux en Intelligence Artificielle sur la représentation des connaissances. Ils présentent l'intérêt de mettre en oeuvre les notions d'objets, de concepts, d'attributs, compatibles avec l'approche objet du génie logiciel, mais surtout ils offrent des outils efficaces de raisonnement.

Pour résumer on peut distinguer ces trois approches de la manière suivante : PECOS pose des règles sur des propriétés logicielles. Les Logiques de Description et la Kind Theory appliquent des outils conceptuels génériques à la vérification de sous-typage ou de cohérence de description d'entités logicielles. Vis à vis des contrats, des chercheurs ont montré de la première que dans certains cas elle permettait de décrire et comparer des assertions de la conception par contrat. La seconde est utilisée pour produire automatiquement du code d'adaptation entre deux interfaces, ce qui permet ainsi l'évaluation de leur compatibilité.

Du point de vue des propriétés recherchées pour le métamodèle de contrat, PECOS offre l'approche la plus pragmatique. Les logiques de description et la kind theory nécessitent une modélisation du système dans leur formalisme pour s'appliquer. Toutefois, ces trois approches possèdent des outils pour vérifier la compatibilité des spécifications, qu'il s'agisse de Prolog pour PECOS ou de prouveurs et modelchecker pour les logiques de description et la kind theory. De plus ces deux dernières ont par rapport à la démarche de PECOS, l'avantage de la généricité vis à vis du code avec toutefois pour inconvénient le problème de l'assurance de la synchronisation du code et de la spécification. Il est finalement à noter qu'aucune de ces approches ne considère la conformité du composant à sa spécification.

2.4.2.4 Bilan

L'approche par contrat classique des plateformes industrielles se limite à l'expression et l'évaluation de la conformité des composants à des spécifications sous forme d'assertions. Elle ne diffère que peu de sa mise en oeuvre sur les objets. CCLJ, pour la plateforme Fractal, est un formalisme à base d'assertions qui par contre réifie ces dernières et prend en compte des spécificités propres aux composants. Il se limite toutefois à l'expression de la conformité. Mais comme nous l'avons vu pour les objets, la compatibilité d'assertions n'est pas complexe à exprimer. La compatibilité des composants devient

explicite et évaluable pour d'autres formalismes plus formels, notamment les algèbres de processus. Différents représentants de ces dernières offrent des outils pour évaluer la conformité et compatibilité de leurs spécifications. Enfin les formalismes logiques n'envisagent globalement que la compatibilité des spécifications mais ils fournissent des outils pour l'évaluer.

2.4.3 Spécifications non-fonctionnelles appliquées aux objets et interfaces

Différentes propriétés de qualité de service peuvent être prises en compte dans le cadre de systèmes à objets. Nous allons envisager deux approches de celles-ci : la première présentera une technique d'expression et de vérification de ces propriétés, la seconde une formalisation de celles-ci permettant de raisonner dessus.

2.4.3.1 QuO : les régions de fonctionnement

Un système modélisé par QuO [65] possède un nombre fini d'états vis à vis de la QoS. Chaque état correspond à l'appartenance de grandeurs de qualité de service (qos) à un domaine de qos, décidée par la validité d'un prédicat associé à ce dernier. Grandeurs qos et domaines sont spécifiés dans un contrat qui définit aussi le comportement à adopter en cas de transition d'un domaine à un autre. Un langage spécifique est utilisé, le Contract Description Language (un exemple de contrat est donné en annexe).
Dans sa réalisation, QuO [65] propose sous forme d'aspects l'intégration des ressources de spécifications, monitoring et adaptation du système de QoS. Au schéma d'exécution traditionnel d'une application en Corba, QuO ajoute les éléments suivants (Figure 2.3) :

- contrats : définissent les niveaux de services requis par le client, fournis par le fournisseur, à l'aide de domaines de valeurs des variables de qos. Ils associent aussi des actions au passage d'un niveau de QoS à un autre.

- délégués : ces objets sont des wrappers locaux des objets distants, fournissant des outils d'adaptation du comportement du système.

- objets d'état du système (SysCond) qui lient les instances de contrats et aux entités fonctionnelles du système, et qui permettent ainsi la mesure de la QoS.

L'évaluation d'un contrat consiste en l'évaluation des domaines de fonctionnement du système et en l'appel des méthodes de réaction associées à la transition d'un domaine à un autre. Cette évaluation est déclenchée (Figure 2.3) soit par le délégué à l'occasion d'interception d'appel de méthode (1,8), soit à la suite de l'évaluation d'un autre contrat (7).
La mise en oeuvre de QuO si elle n'est pas intrusive, reste assez complexe puisque différents langages sont à utiliser. Elle n'est d'autre part pas très dynamique imposant des recompilations des contrats et des délégués en cas de modification de politiques. Surtout elle n'est pas réflexive et les régions de fonctionnement ne peuvent pas être comparées entre elles, ce qui bride l'étude de la compatibilité des composants. Plus de détails sur QuO sont donnés en annexe.

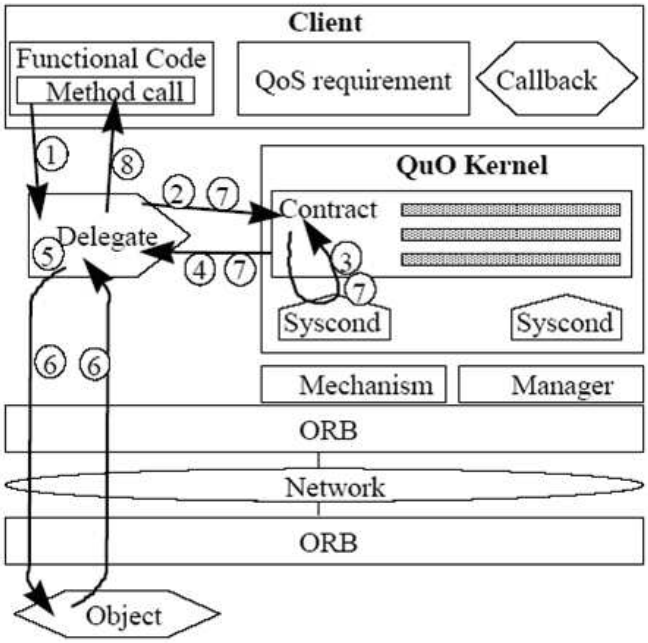


FIGURE 2.3 – QuO

2.4.3.2 QML : définition de métriques

La définition de qualité de services par QML [37] repose sur trois concepts clés :

- le type de contrat : nommé, il définit les dimensions sur lesquelles peuvent porter le contrat, à chacune est associée un domaine (intervalle ou ensemble), éventuellement ordonné, ex. :

```
type Reliability = contract
{
   numberOfFailures : decreasing numeric no/year;
   TTR: increasing numeric sec;
   availability : increasing numeric;
}
```

- le contrat : consiste en une instance d'un type de contrat pour une spécification donnée, c'est à dire en un ensemble de contraintes portant sur les dimensions du type de contrat, ex. :

```
systemReliability = Reliability contract
{
   numberOfFailures < 10 no/year;
   TTR {
     Percentile 100 < 2000;
     Mean < 500;
    variance < 0.3;
   }
```

```
   availability > 0.8;
}
```

Plus de détails sur les contrats sont donnés en annexe.

- le profil : applique des contrats de manière éventuellement indépendante aux entités d'implémentation, interface, méthodes.... Par exemple pour l'interface de taux de changes :

```
RateServiceI
{
   Rates latest (in Currency c1, in Currency c2)
   raises (Invalid C);
   Forecast analysis (in Currency c)
   raises (Failed);
}
```

On aura par exemple :

```
rateServerProfile for RateServiceI = profile
{
   require systemReliability;
   from latest require Performance contract
   {
      delay
      {
       percentile 50 < 10 msec;
       percentile 80 < 20 msec;
       percentile 100 < 40 msec;
       mean < 15 msec;
      };
   };
};
```

QML se distingue par sa logique de mise en oeuvre de métrique qui a pour intérêt de rendre comparable des spécifications de manière statique. Compatibilité et substituabilité des entités spécifiées peuvent ainsi être envisagées de manière statique. Toutefois la mise en oeuvre de la vérification de la conformité des objets à leur spécification n'est pas considérée.

2.4.4 Spécifications non-fonctionnelles appliquées aux composants

L'expression des propriétés de QoS est d'autant plus importante dans le cadre des composants que ceux-ci se veulent plus réutilisables encore que les objets. Nous allons présenter deux approches issues de QML. Puis nous décrirons une mise en oeuvre de propriétés de QoS dont les modalités nous ont parues intéressantes.

2.4.4.1 CQML : l'extension de QML

CQML [3] reprend des traits caractéristiques de QML en les développant largement. Il s'articule autour de 3 concepts : la caractéristique de QoS (définie par sa métrique), le contrat, le profil (association avec les entités d'implémentation).

La caractéristique de QoS La caractéristique qos correspond au minimum à une grandeur dont le domaine est un intervalle ou un ensemble, possiblement ordonné, ou une énumération. Comme QML, CQML propose la dérivation qui consiste à définir une grandeur en fonction d'une autre. Les dérivations fournies sont essentiellement statistiques et sont plus nombreuses que dans QML, ex . :

```
quality_characteristic Q0
{
   domain : decreasing numeric millisecond;
}

quality_characteristic Q1 : Q0
{
   mean;
}
```

Q1 est définie comme la moyenne de Q0.

Le contrat Le contrat regroupe sous un nom une liste de contraintes sur des caractéristiques de qos, à la manière des contrats de QML. Un exemple basique en est :

```
quality guaranteed_high
{
   frameOutput >= 25;
}
```

Les profils Les profils QML et CQML ont en commun le principe d'attacher un contrat à une entité d'implémentation. En effet le profil CQML définit les spécifications offertes (provides) et requises (uses) pour un composant (ou objet) donné. Par exemple :

```
quality high (flow : Flow)
{
   frameOutput (flow) >= 25;
}

profile goodCamera for myFastCamera
{
   provides high (outgoing.videoFlow);
}
```

Pour la négociation, les profils sont comparables à l'aide d'une relation de conformité semblable à celle de QML, et sont ordonnés (clause precedence). Toujours par rapport à

QML, les clauses uses et provides des profiles explicitent directement offres et attentes des composants. Comme pour QML, la conformité des composants à leur spécification n'est pas instrumentée mais le formalisme facilite l'étude de la compatibilité.

2.4.4.2 CQML+

CQML+ [94] est une extension à CQML. Il définit précisément un métamodèle (Figure 2.4) d'exécution dans le cadre duquel les spécifications s'expriment.

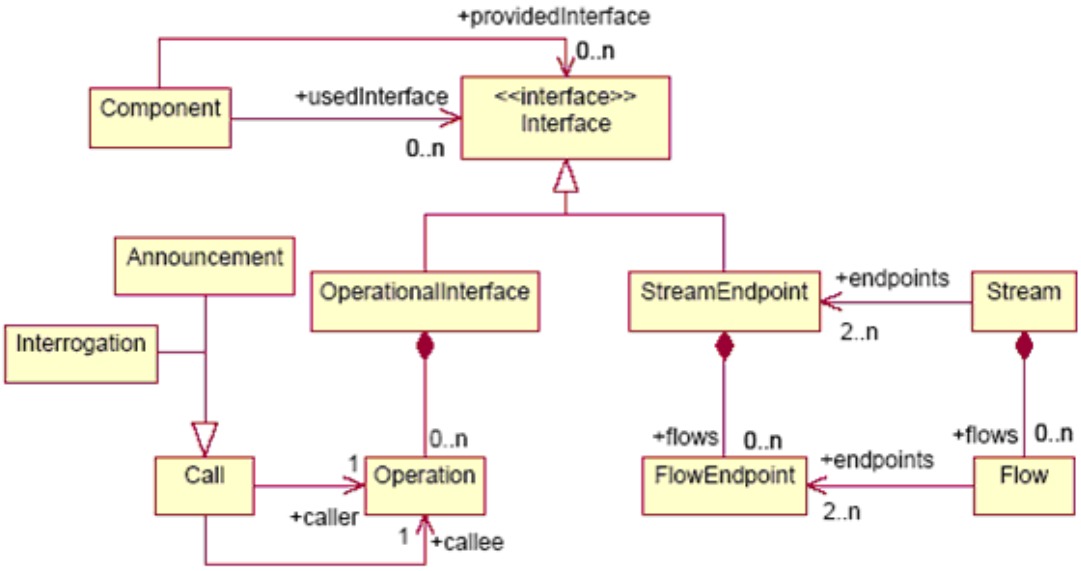


FIGURE 2.4 – CQML+

Ainsi, tant que les expressions QoS ne font intervenir que des grandeurs du métamodèle, elles sont statiquement comparables, alors que la mise en oeuvre de grandeurs extraites du système cible implique la nécessité de vérification dynamique.

2.4.4.3 Comparaison de QuO, QML, CQML et CQML+

	CQML	QML	QuO	CQML+
adaptation	O	N	O	O
séparation specification/implementation	O	O	O	O
comparabilité des spécifications	O	O	N	O
combinaison de composants	O	N	N	O
séparation fonctionnel/non-fonctionnel dans la description des spécifications	O	O	N	O
spécialisation des caractéristiques et des contrats	O	O	N	O
généricité par rapport au cycle de vie de l'application	O	O	N	O
métamodèle du système contractualisé	N	N	N	O
prise en compte explicite des ressources	N	N	N	O
caractéristiques composites	N	N	N	O
liaison entre les expressions des spécifications requises et fournies	N	N	N	O

2.4.4.4 Le modèle des KComponent

Dans ce modèle [48] de composants, décrit en annexe, la configuration du système est réifiée en un graphe, manipulé et transformé par des programmes réflexifs nommés "contrats d'adaptation". Chaque contrat contient un ensemble de règles spécifiant des contraintes architecturales et les opérations de reconfigurations à entreprendre en cas de violation. Même s'il ne présente pas de contrat associant des garants et bénéficiaires à des clauses, le KComponent Framework est intéressant de par la mise en oeuvre des contrats qu'il effectue. En effet ceux-ci apparaissent comme des entités de premier ordre, et non plus au niveau de l'application mais au niveau méta. Par ailleurs une entité, le Configuration Manager gère le couplage entre le cycle de vie des contrats et celui de l'application. D'autre part les contrats s'appuient sur une vue de l'application au travers de son graphe de configuration, et suivent son activité au travers d'événements.

2.4.5 Spécifications non-fonctionnelles appliquées aux services

La prise en compte des propriétés de QoS des services est d'autant plus importante qu'elle peut être l'objet de transactions commerciales entre client et fournisseur du service. Nous présenterons donc une des techniques les plus répandues pour exprimer ces propriétés sur les services.

2.4.5.1 Service Level Agreement

Une définition commune des Service Level Agreement est la suivante :

> un accord sur la qualité de service passé entre un fournisseur et un utilisateur d'un service

Cet accord fait intervenir différentes dimensions destinées à expliciter et garantir l'utilisation du service. Le site de la société "ITIL & ITSM WORLD" fournit un ensemble

non exhaustif de celles-ci : "domaine de fonctionnement, performance, suivi et retour d'utilisation, gestion des problèmes, compensation, devoirs et responsabilités de l'utilisateur, garanties et solutions, sécurité, propriété intellectuelle, etc". Par défaut cet accord est l'objet de documents papier traditionnels, néanmoins avec l'automatisation de la mise en oeuvre de service il est naturel de voir leurs équivalents électroniques apparaître. Ainsi IBM propose sur son site une spécification des Web Service Level Agreement [66]. Appliqués aux Web Services, ces contrats reposent sur trois principes fondamentaux (Figure 2.5) :

- les parties impliquées dans le contrat
- la description du service
- les obligations de chacune des parties par rapport à la description du service

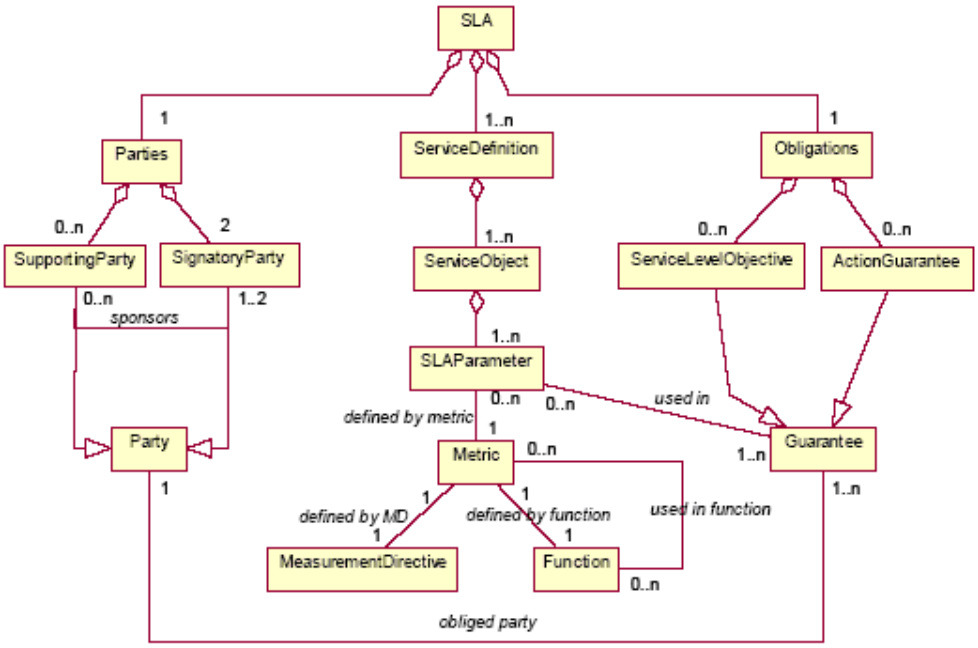


FIGURE 2.5 – SLA

Parties impliquées Les parties impliquées par le contrat sont de deux types. Il s'agit :

- soit des signataires, c'est-à-dire le consommateur et le fournisseur de service,
- soit de contributeurs autorisés par les signataires pour effectuer les mesures, la vérification des obligations à respecter, ou la coordination des actions d'une partie,

La description du service La description du service repose sur un ensemble de ServiceObject, dont chacun définit un ou plusieurs SLAParameter. Chaque SLAParameter correspond à une grandeur, associée à une métrique qui a pour objet de définir la manière dont sa valeur est mesurée, ou évaluée à l'aide d'une fonction. Par ailleurs,

cette métrique peut être composite et s'appuyer sur d'autres. La classe ServiceObject sert de classe de base aux classes de description d'opération OperationDescription. Ces classes décrivent les opérations du service de manière classique (WSDL : nom, signature...) tout en leur associant des SLAParameters.

Voici un exemple de SLAParameter :

```
<SLAParameter name="TransactionRate"
       type="float"
       unit="transactions / hour">
   <Metric>Transactions</Metric>
</SLAParameter>
```

La mesure de la valeur du paramètre fait appel à la métrique Transactions qui définit les fonctions d'évaluation. Un exemple de description d'opération est donné en annexe.

Les Obligations Les obligations sont de deux types :

- le Service Level Objective décrit des contraintes sur les valeurs des SLAParameters pour une certaine période. Le garant de cette obligation est spécifié par son attribut "obliged", c'est le provider. Par contre, une autre entité que le provider peut être sollicitée pour l'évaluation de la condition décrite par le Service Level Objective. Un exemple en est donné en annexe.
- le ActionGuarantee décrit l'engagement d'effectuer une certaine action dans une situation donnée ; classiquement quand un certain événement est détecté, un appel de service ou une notification doit être effectuée. Un exemple en est donné en annexe.

2.4.5.2 Cycle de vie des SLA

Différents travaux ont étudié le cycle de vie des SLA. De manière récurrente nous constatons que certaines phases leur sont communes. Comme point de départ nous considérerons le cycle de vie des SLA dans le cadre du framework WSLA [55] :

1. Négociation et établissement du SLA : dans cette phase, les différents intervenants du SLA sont désignés, les paramètres et leurs métriques sont définis, les contraintes qui forment les obligations des participants par rapport à ces grandeurs sont fixées. Un document complet décrivant le SLA est ainsi obtenu.
2. Déploiement du SLA : chaque signataire du SLA informe les tierces parties (chargées des mesures etc) de leur rôle (mesure de paramètres, vérification de contraintes etc).
3. Mesures et vérifications des contraintes : les tierces parties en charge des mesures les effectuent, ces dernières sont utilisées par les services dédiés à la vérification des contraintes du contrat pour évaluer celles-ci.
4. Actions correctives : le service de gestion du SLA obtient de celui-ci les actions à effectuer en cas de violation et il les applique au système contraint.

Par ailleurs, lors de la phase de déploiement du SLA il est possible d'ajouter le provisionnement de ressources [25] rendu nécessaire par les Service Level Objective du côté du fournisseur de service.

2.4.5.3 Bilan

L'approche des WSLA est intéressante car elle n'est pas initialement issue de besoins de génie logiciel. Au contraire, elle concrétise les préoccupations d'utilisateurs finaux vis-à-vis de la qualité de service. Ils ont l'avantage de présenter une logique générique par rapport à leur plateforme d'application. Par ailleurs, bien qu'assez verbeux du fait de l'utilisation d'XML, ils ne font pas intervenir de formalismes complexes. L'utilisation de métriques pourrait rendre envisageable la comparaison statique de spécifications à la manière dont QML procède. Toutefois seule la conformité du service à sa spécification est explicitement envisagée, bien qu'elle soit augmentée d'une notion de réaction à sa non-satisfaction.

2.5 Le raisonnement par hypothèse garantie

Nous avons vu que les premiers contrats logiciels, tels ceux de Meyer, étaient issus de la logique de Hoare [45]. Or celle-ci est une instance du paradigme de spécification hypothèse-garantie. Nous nous intéresserons dans cette partie aux liens de ce dernier avec la logique de Hoare, puis aux différents types de spécifications qu'il permet de décrire. Enfin nous terminerons en considérant comment la notion contractuelle d'accord peut être exprimée à l'aide d'hypothèses et de garanties.

2.5.1 Paradigme Assume-guarantee et logique de Hoare

Une définition d'assez haut niveau du paradigme assume-guarantee est donnée dans [99] :

> "The assumption characterizes the essential properties of the environment in which the specified program, from now on referred to as the agent, is supposed to run, while the commitment is a requirement which must be fulfilled by the agent whenever it is executed in an environment which satisfies the assumption"

Il est aussi nommé, d'après [110], paradigme "assumption-commitment" ou encore "rely-guarantee". La première dénomination étant a priori plus utilisée dans le contexte des agents [99], la seconde dans le contexte de la programmation concurrente [50]. Par ailleurs, le paradigme assume-guarantee a été utilisé dans de nombreuses approches de spécifications et preuves compositionnelles citées dans [85] et [110]. Une definition de la logique de Hoare, du point de vue du paradigme assume-guarantee est la suivante [45] (le "triplet de Hoare") :

> "P {Q} R : This may be interpreted : 'If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion'".

Ces deux définitions montrent bien le rapprochement qui peut être effectué entre les deux modèles : prendre la précondition pour l'hypothèse et la post condition pour la

garantie. Le positionnement historique du modèle de Hoare et du paradigme assumption-guarantee est le suivant [77] :

- 1969 : Hoare propose un système formel d'axiomes et de règles d'inférences pour la vérification des programmes séquentiels impératifs

- 1976 : Susan Owicki et David Gries étendent le système de Hoare pour la vérification de programmes parallèles avec des variables partagées :

  – Le programme est décrit par une "proof outline" [77], c'est à dire pour tout a_i (i entier), action atomique, et p_i assertion : "$\{p_1\}$ a_1 $\{p_2\}$ a_2 $\{p_3\}$..." : chaque assertion p_i atteinte par le programme est vraie pour son état courant,

- 1981 : Cliff Jones introduit le concept de la méthode rely-guarantee, une version compositionnelle du système de Owicki-Gries. Dans ce dernier, la définition des hypothèses et garanties est la suivante ([77], [50]) :

  – " Rely : any environment transition in the computation satisfies the rely condition",

  – " Guarantee : any component transition satisfies the guarantee condition",

D'un point de vue de classification, la logique de Hoare est le plus souvent définie comme un cas particulier du paradigme assume-guarantee. Dans [99] elle est classée dans les représentants de ce paradigme dont les parties hypothèses et garanties sont explicites et distinctes, par opposition aux représentants du cas contraire. Elle se retrouve rangée au côté de travaux tels que la méthode rely/guarantee de Jones, dont elle est rapprochée dans [50], la précondition étant le "rely" et la post condition le "guarantee". De la même manière, Lamport décrit dans [2] le triplet de Hoare comme un cas particulier, pour la correction partielle des programmes, de l'approche de spécifications par hypothèse/garantie qu'il considère dans le cadre de son principe de composition de celles-ci.
Du point de vue des évolutions, selon [107] les règles assumption-commitment furent au commencement étudiées comme des extensions de la logique de Hoare. Ce cas de figure, dans lequel on considère l'extension des règles de Hoare, est illustré en particulier par les travaux sur la "Generalized Hoare Logic" [61]. Dans ce dernier cas, les formules sont de la forme : " {I} π {I} ", ou "I" est un prédicat et "π" un fragment de programme, et signifient que l'exécution de toute action atomique dans π débutant dans un état où I est vrai, laisse I vrai. Une autre étude [98] présente explicitement la logique de Hoare comme un cas particulier du système de spécification qu'elle définit et qui est basé sur le paradigme rely/guarantee. Dans ce cas, les auteurs font remarquer qu'ils élargissent le champ d'application de la méthode puisque celle de Hoare ne permet la preuve que des propriétés de sûreté ("sous certaines conditions quelque chose ne va jamais arriver") alors que la leur autorise la démonstration de propriétés de vivacité ("sous certaines conditions quelque chose finira par arriver"). Toutefois dans [107], les auteurs font remarquer que, dans le cas de travaux sur le paradigme assume-guarantee dont les spécifications reposent sur des prédicats temporels (Abadi & Lamport, Pnueli, Barringer & Kuiper), alors celui de l'hypothèse est restreint à une propriété de sureté.

2.5.2 Portées architecturales des différents types de spécifications

Les travaux relevant du paradigme assume-guarantee se divisent en deux grandes catégorie, suivant qu'ils séparent ou non l'objet spécifié (composant ou programme) de son environnement. Dans la catégorie séparant l'entité spécifiée de son environnement, on trouve les travaux sur la composition architecturale (citées dans [85], [44], [107], [2]) et la concurrence ([50], [110], [98]), alors que ceux plus proches de la logique de Hoare ([77], [45], [61]) ne font pas la distinction. Ainsi les travaux sur la composition architecturale considèrent que :

- l'hypothèse porte sur l'environnement de l'entité spécifiée,
- la garantie s'applique à l'entité spécifiée elle-même,

Il en va de même pour les études sur la concurrence, dont certaines néanmoins font porter l'hypothèse et la garantie sur des variables globales de l'environnement de l'entité spécifiée. Par contre dans le cas de la logique de Hoare [45], de sa généralisation [61], ou des travaux de Owicki Gries [77], préconditions et postconditions portent sur l'état du programme sans le distinguer de son environnement.

2.5.3 Positionnement par rapport à la notion de compatibilité

Les différentes approches d'assume-guarantee considérées fournissent des formalismes pour spécifier des entités sur la base de propriétés dont elles dépendent et d'autres qu'elles garantissent. Or dans le cadre de notre approche, comme expliqué dans l'introduction, nous nous intéressons non seulement à la conformité des participants du contrat à leurs spécifications, mais aussi à la compatibilité de ces dernières. En particulier il est important que les hypothèses de bon fonctionnement des entités soient compatibles avec les garanties que les autres fournissent. En effet les garanties de ces dernières décrivent l'environnement sur lequel portent les hypothèses des premières. Nous tentons donc de distinguer parmi les travaux existants ceux chez qui la notion de compatibilité telle qu'elle nous intéresse serait présente.

Ainsi dans les travaux de Misra et Chandy [75] sur la composition de processus spécifiés à l'aide d'hypothèses et de garanties, un théorème de composition fait apparaître cette notion de compatibilité sans pour autant en faire une propriété explicite. Dans l'expression de la figure 2.6, H est un processus issu de la composition de processus h_i, dont chacun a pour hypothèse r_i et pour garantie s_i.
$(S\ and\ R_0) \Rightarrow R$ et $S \Rightarrow S_0$: est une expression de compatibilité dans le cadre de laquelle la conjonction des garanties de chacunes des sous parties de H (h_i) et de l'hypothèse du composite H doit impliquer la conjonction des hypothèses des sous parties h_i. Mais aussi, dans lequel la conjonction des garanties des sous parties de H doit impliquer la garantie de H.

D'autres règles de composition d'entités spécifiées à l'aide d'hypothèses et garanties font encore apparaître la notion de compatibilité toujours non réifiée. Ainsi la règle de composition de la figure 2.7 apparaît dans [107], dans le cas particulier où la dépendance entre les composants n'est pas cyclique.

Theorem of Hierarchy:

$$\frac{\text{For all } i, r_i|h_i|s_i;\ \ (S \text{ and } R_0) \Rightarrow R, S \Rightarrow S_0}{R_0|H|S_0}$$

where R_0, S_0 are assertions on the external trace of H, and R, S, denote $\underset{i}{\text{and}}\ r_i$, $\underset{i}{\text{and}}\ s_i$, respectively.

FIGURE 2.6 – Théorème de Misra Chandy

$$\frac{\begin{array}{c} A \wedge C_1 \wedge \cdots \wedge C_{i-1} \Rightarrow A_i \\ P_i \ \ \underline{sat} \ \ A_i \Rightarrow C_i \end{array}}{P_1 \parallel \cdots \parallel P_n \ \ \underline{sat} \ \ A \Rightarrow C_1 \wedge \cdots \wedge C_n}$$

FIGURE 2.7 – Règle de Xu Qiwen et Mohalik Swarup

Les composants sont notés P_i, les A_i sont leurs hypothèses et les C_i leurs obligations. A est l'hypothèse de la composition. L'interprétation de cette règle conduit à une expression de l'accord suivante : $A \wedge C_i \wedge ... \wedge C_{i-1} \Rightarrow A_i$ qui serait à comparer avec la compatibilité extraite des travaux de Misra et Chandi vus précédemment [75].

L'interprétation du théorème de composition de Lamport (détaillé dans l'annexe A.2.6) amène à une expression de compatibilité moins simple que les précédents :

$$\frac{E \cap M_1 \cap M_2 \subseteq \ E_1 \cap E_2}{(E_1 \rightarrow\!\triangleright M_1) \cap (E_2 \rightarrow\!\triangleright M_2) \subseteq (E \rightarrow\!\triangleright M_1 \cap M_2)}.$$

FIGURE 2.8 – Théorème de Abadi Lamport

La figure 2.8 exprime que :

- pour deux composants C_1 et C_2,
- E_1 et E_2 les hypothèses respectives de ces composants sur leur environnement, E l'hypothèse de leur composition sur son environnement,
- M_1 et M_2 les garanties respectives offertes par ces composants,
- chaque composant a pour spécification $E_i \rightarrow\!\triangleright\ M_i$: tant que son environnement satisfait à l'hypothèse E_i alors le composant satisfait à son obligation M_i,

si ces composants sont compatibles (expression de la prémisse de la règle d'inférence) alors on connait la spécification de leur composition ($E \rightarrow\!\triangleright\ M_1 \bigcap M_2$) et celle-ci est vérifiée pourvu que les composants satisfassent aux leurs. L'expression de l'accord dans le cadre de ce théorème est la prémisse de la règle d'inférence : $E \cap M_1 \cap M_2 \subseteq \ E_1 \cap E_2$:

les garanties des composants et l'hypothèse sur l'environnement doivent impliquer les hypothèses des composants. Cela revient à exprimer que les besoins des composants doivent être satisfaits par ce qu'ils fournissent et ce théorème est généralisable à N composants.

2.5.4 Conclusion

Les différentes approches hypothèse-garantie font porter leurs contraintes sur l'état des programmes et de leur environnement. Toutefois il est intéressant de constater que pour qu'une donnée soit échangée entre un programme et son environnement, celle-ci doit dabord être produite et exposée par l'un ou l'autre. Cette donnée reflète donc l'état à un certain moment du programme ou de son environnement. Ainsi les approches hypothèse-garantie s'appliquent aussi aux échanges entre entités logicielles. Elles sont même un moyen particulièrement simple et minimal de décrire les interactions entre un programme et son environnement. Un autre détail remarquable concernant les approches que nous avons citées est que l'expression de la compatibilité des hypothèses et des garanties d'un système d'entité logicielles, exprimée dans la notion d'accord, ne fait pas intervenir explicitement leur architecture ou organisation. Or cette dernière joue bien sur un rôle central dans les dépendances qui peuvent apparaître entre les entités du système.

2.6 Les modèles et métamodèles de contrat

Nous présentons dans cette partie un ensemble de modèles et de formalismes de contrat, en s'attachant aux propriétés qu'ils réifient. Nous nous intéresserons ainsi à la représentation des responsabilités, à la définition du cycle de vie du contrat, aux liens du contrat avec l'architecture qu'il contraint, etc... . Nous nous intéresserons aussi à ce que chacun de ces modèles de contrat exprime. En effet dans [9] le SEI[3] distingue deux types de contrat. Un premier exprime l'adhérence d'un élément à sa spécification alors que le second concerne la compatibilité d'un ensemble d'éléments collaborant via celle de leurs spécifications. Ces modèles et formalismes sont regroupés suivant la nature des éléments qu'ils contraignent, objets, composants, services et agents.

2.6.1 Méta-modèles de contrat

L'Ingénierie Dirigée par les Modèles (IDM) est une branche de l'informatique qui produit des outils, des concepts et des langages pour créer et transformer des modèles. L'objectif de cette approche est de générer tout ou partie des logiciels sur la base de modèles. Le MDA[4] en est un représentant qui s'est spécialisé en distinguant les modèles indépendants de la plate-forme de déploiement ou d'implémentation, de ceux relatifs à celle-ci. Dans le cadre de l'IDM, un modèle est défini de la manière suivante dans [58] :

- "A model is the description of a system written in a well defined language"

[3]Software Engineering Institute de l'université Carnegie Mellon : http ://www.sei.cmu.edu/
[4]Model Driven Architecture [83]

- "A well-defined language is a language with well-defined form (syntax) and meaning (semantics), which is suitable for automated interpretation by a computer"

Pour notre part, nous considérons que :

> *un modèle de contrat décrit une organisation des spécifications (et de leur vérification) des échanges au sein d'un système pour garantir à celui-ci certaines propriétés*

L'exemple le plus simple est le modèle de contrat à la Meyer qui associe deux contraintes, la pré et la post condition, avec la sémantique d'évaluation de l'approche hypothèse-garantie. Suivant les ensembles d'éléments architecturaux contraints par un modèle de contrat, l'organisation qu'il décrit peut être instanciée plusieurs fois sur un même système, pour garantir ses propriétés à différents endroits. Ainsi le contrat à la Meyer est instancié sur chaque méthode présentant des pré et post conditions, dans Eiffel sous forme de code ajouté à celui du programme. Le modèle de contrat est utilisé pour guider la mise en oeuvre et l'interprétation des évaluations des spécifications propres aux éléments architecturaux sur lesquels on l'instancie. Un métamodèle de contrat présente l'intérêt d'offrir les éléments pour décrire le modèle de contrat. Il permet ainsi de produire différents modèles de contrat décrivant des organisations différentes des spécifications dans le cadre d'échanges variés. Nous allons voir dans les parties suivantes différents exemples de métamodèles de contrat, soient des langages pour décrire différentes formes de contrat.

2.6.2 Objets

Les contrats sur les objets sont sans doute les premiers historiquement apparus. On peut distinguer les contrats exprimant qu'un objet satisfait à une spécification, des contrats exprimant la compatibilité entre deux spécifications. L'approche fondatrice du Design By Contract de Meyer [71], permet à la fois la vérification de la satisfaction de pre et post conditions par une classe et l'étude de la compatibilité entre ces assertions via celle du sous typage comportemental [33]. Dans ce dernier cas, la compatibilité d'assertions portées par des classes parentes sur des méthodes identiques est étudiée, ce qui valide le polymorphisme du point de vue de cette approche.
OCL. Toutefois la métamodélisation d'OCL (figure 2.9) proposée par [92] bien qu'elle réifie les pré et post conditions semblables à celles du design by Contract ne fait pas apparaître cette distinction. Elle présente les assertions et leur lien avec les objets qu'elles contraignent mais n'exprime pas le sous typage comportemental, ainsi que le montre la figure .
Contrats et collaboration. Une autre approche contractuelle est mise en oeuvre dans [43] (figure 2.10). Dans celle-ci les contrats sont explicitement réifiés, constitués de participants et d'obligations, et appliqués aux compositions d'objets. Les principes de satisfaction des participants à des spécifications et de compatibilité entre ceux-ci sont exprimés via des obligations de deux genres. Le premier principe est rendu par une catégorie d'obligations portant sur les propriétés de typage des participants. Le second principe est exprimé via des obligations "causales", capturant des dépendances comportementales entre objets, en contraignant les séquences de messages émis et reçus de chacun (clauses `supports`). Des invariants doivent aussi être satisfaits par les participants. Il faut noter que ces contrats supportent entre eux des relations de raffinement

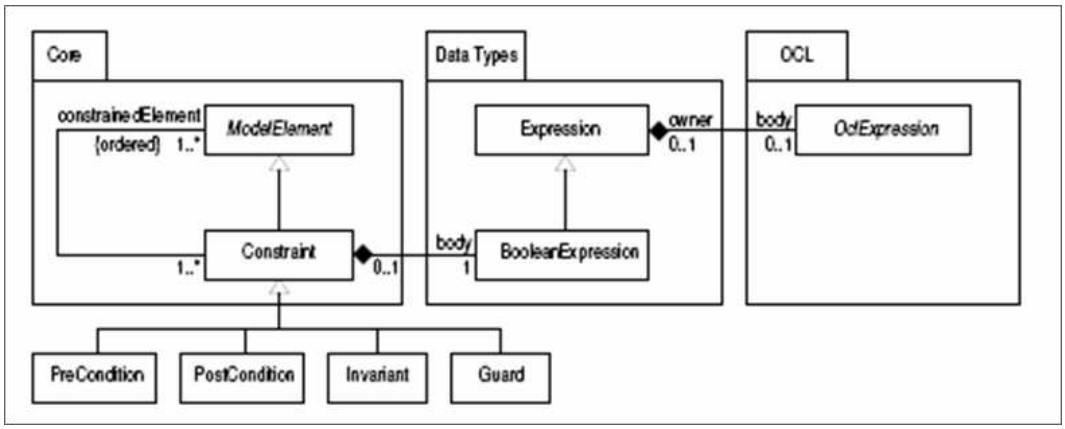


FIGURE 2.9 – métamodèle OCL

et d'inclusion (`includes`). Il est aussi intéressant de constater que l'instantiation des contrats peut être soumise à des conditions, on voit ici ainsi apparaître une notion de cycle de vie du contrat. Enfin, les auteurs proposent une notion de conformance pour rendre compte de l'acceptabilité d'une classe à un rôle défini dans un contrat. Néanmoins il faut remarquer que ce type de contrat spécifie la composition en elle même, et non la compatibilité entre les spécifications de ses participants.

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

FIGURE 2.10 – Contrat d'interaction

Contrats et coordination. Une extrême de l'approche précédente est rendue dans les contrats de coordination définis par [6] (figure 2.11). Ces contrats, réifiés, possèdent des participants, auxquels sont appliqués des invariants exprimant leurs relations. Mais ces contrats contiennent aussi l'expression de la coordination entre leurs participants. Dérivés de classes d'associations, ils sont placés entre les instances de ceux-ci et interceptent de manière transparente leurs messages, déclenchant éventuellement ainsi des actions définies par des règles "when <condition> do <actions> with <condition>". Dans ces contrats la garantie de la compatibilité entre les participants ne porte pas sur

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

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

FIGURE 2.11 – Contrat de coordination

Contrats et responsabilités. Des contrats issus des travaux sur les agents sont utilisés dans [87] pour modéliser de manière formelle les Use Cases de la conception objet. Ils reposent sur un langage simple définissant une algèbre de contrats, sachant que les auteurs définissent un contrat comme la description des voies par lesquelles un acteur peut modifier le système. Dans ce cadre le système et les acteurs sont considérés comme des agents et contraints par un langage dont la syntaxe de base est la suivante :

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

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

Conclusion
Si le contrat tend à être réifié quand on s'éloigne du Design By Contract, les responsabilités ne le sont en général pas, ni le cycle de vie du contrat. Hormis dans le dernier exemple, il n'y a pas de dépendance exprimée entre les clauses du contrat, de sorte que l'une d'entre elles soit conditionnée par une autre. Le lien avec l'architecture est plutôt ténu, puisque la composition est considérée en dehors des relations individuelles entre les objets, et aucunement dans le dernier cas de figure des contrats.

2.6.3 Composants

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

QCCS. Un métamodèle de contrat (figure 2.12) est défini au sein des travaux relatifs à QCCS [95] dans le cadre de la modélisation UML version 1.4 des composants. Ce modèle aborde la garantie de propriété fonctionnelles et non fonctionnelles des interfaces de composants. En effet celles-ci sont complètement décrites par des contrats, tant pour leur signature que vis à vis de la qualité de service qui repose sur QML [36]. Pour ce faire ces contrats peuvent être composés. Leur mise en oeuvre est envisagée via un tissage dans le code à la manière des aspects. Ce modèle est intéressant par son intégration à UML, toutefois seuls les contrats en tant que satisfaction des composants à des spécifications sont envisagés. Par ailleurs, le contrat en lui même n'est pas détaillé, on ne voit ainsi par exemple ni ses clauses, ni son cycle de vie, etc...

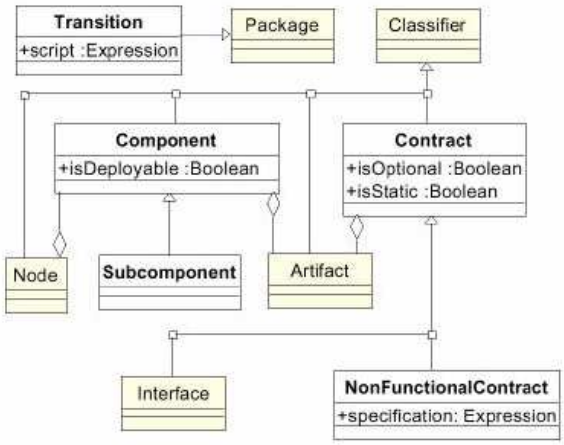


FIGURE 2.12 – métamodèle QCCS de contrat

QoSCL. Un autre modèle de contrat, QoSCL [51], est décrit dans le cadre de UML 2.0 (figure 2.13). Ce modèle intervient aussi bien au niveau de la garantie des spécifications des interfaces d'un composant, qu'à celui de la compatibilité de ces spécifications au sein d'un assemblage de composants. La notion de contrat reprend celle de QML, ainsi un type de contrat associe à une interface un ensemble de dimensions, c'est à dire de grandeurs non fonctionnelles contraintes. Ce contrat est vérifié à l'exécution via son tissage dans le code de l'application. Par ailleurs, les contrats sont des constituants d'un QoSComponent, dérivant de la classe UML Component, dont ils sont requis et fournis. Le formalisme permet d'exprimer qu'au sein d'un composant, il existe des relations entre les dimensions des contrats requis et fournis. Les auteurs présentent une méthode de traduction en langage logique de ces relations et dimensions pour chacun des composants. Ils appliquent ensuite à cette description de l'ensemble du système de composants, un solveur de contraintes qui fournit les domaines de valeur acceptables pour les dimensions des contrats. Ce modèle présente l'intérêt d'expliciter des relations entre clauses du contrat. Par ailleurs même si le contrat ne garantit que l'adhérence d'un composant à ses spécifications, l'assemblage de composants est garanti par la vérification de la composition de leurs contrats. Toutefois le problème des responsabilités n'est pas abordé et la garantie de l'assemblage n'est pas l'objet d'un contrat unique.

GCCL. Dans le cadre des travaux [68] les auteurs définissent un langage de contrat destiné à vérifier la satisfaction des composants à leurs spécifications ainsi que la correction de leurs assemblages, tant du point de vue fonctionnel que non fonctionnel.

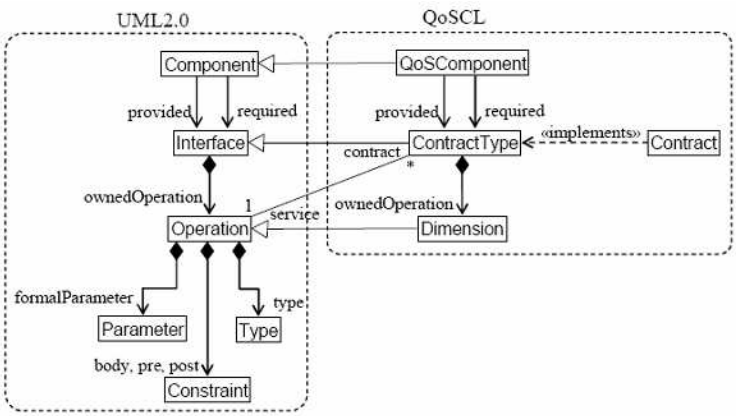


FIGURE 2.13 – métamodèle QoSCL de contrat

Ce langage GCCL (Generalized Common Contract Language, figure 2.14) décrit les contrats comme des ensembles de contraintes exprimées à l'aide de pré, post conditions et d'invariants appliqués à divers éléments. Il faut noter que ce langage fournit un modèle de l'architecture à laquelle s'applique les contrats, un modèle des caractéristiques de QoS qu'ils contraignent, un modèle des événements ainsi que des cycles de vie des éléments de l'architecture contrainte. L'intégration des contrats dans l'architecture contrainte est ainsi complète. Le modèle événementiel permet de positionner le déclenchement de la vérification des contraintes par rapport aux cycle de vie des entités sur les caractéristiques desquelles elles portent. Comme les contrats prennent en paramètres les composants ou interfaces présentant les grandeurs de QoS qu'ils contraignent, l'adhérence des composants à leur spécification peut être vérifiée à l'exécution. Ces contraintes peuvent par ailleurs dépendre des variables que les contrats prennent en paramètre, et se retrouver corrélées quand elles en ont en commun. D'autre part, le modèle de caractéristiques de QoS est défini de sorte qu'il soit possible d'appréhender les possibilités de traitement de leurs contraintes par des solveurs. La satisfaction des composants aux contrats peut ainsi être vérifiée statiquement. Finalement, des dépendances entre contrats peuvent être définies de diverses manières, par exemple via des paramètres qu'ils partagent. La compatibilité des éléments contractualisés et donc la validité des assemblages peut ainsi être vérifiée. Toutefois la notion de responsabilité n'est pas explicitée, ni les rôles des participants au contrat qui n'interviennent qu'en tant que "fournisseurs" de grandeurs contraintes.

Liaison entre comportements et architecture. Un modèle de contrat beaucoup plus orienté vers la garantie de l'assemblage des composants est décrit dans [102] (figure 2.15). Dans celui-ci sont réifiées les spécifications des ports des composants (`AssemblyContract`), mais aussi celles des liens entre ports (`Dependence`, `Synchronization`, `LinkBehavior`). Ainsi une connexion entre deux ports est validée par la vérification statique de la compatibilité de leurs `AssemblyContract` respectifs, constitués de pré et post conditions écrites dans un langage logique. Le contrat `Dependence`, exprime la dépendance comportementale entre ports d'un même composant, le `LinkBehavior` entre ports de composants distincts. Ces contrats sont en fait des spécifications des comportements dont la composition fournit celui de l'assemblage de composants, tout en garantissant leur compatibilité. Ce modèle de contrat présente l'avantage d'être très proche de l'architecture, puisque à chaque relation architecturale correspond un type de contrat. Par ailleurs les contrats comportementaux locaux sont composables afin d'obtenir le con-

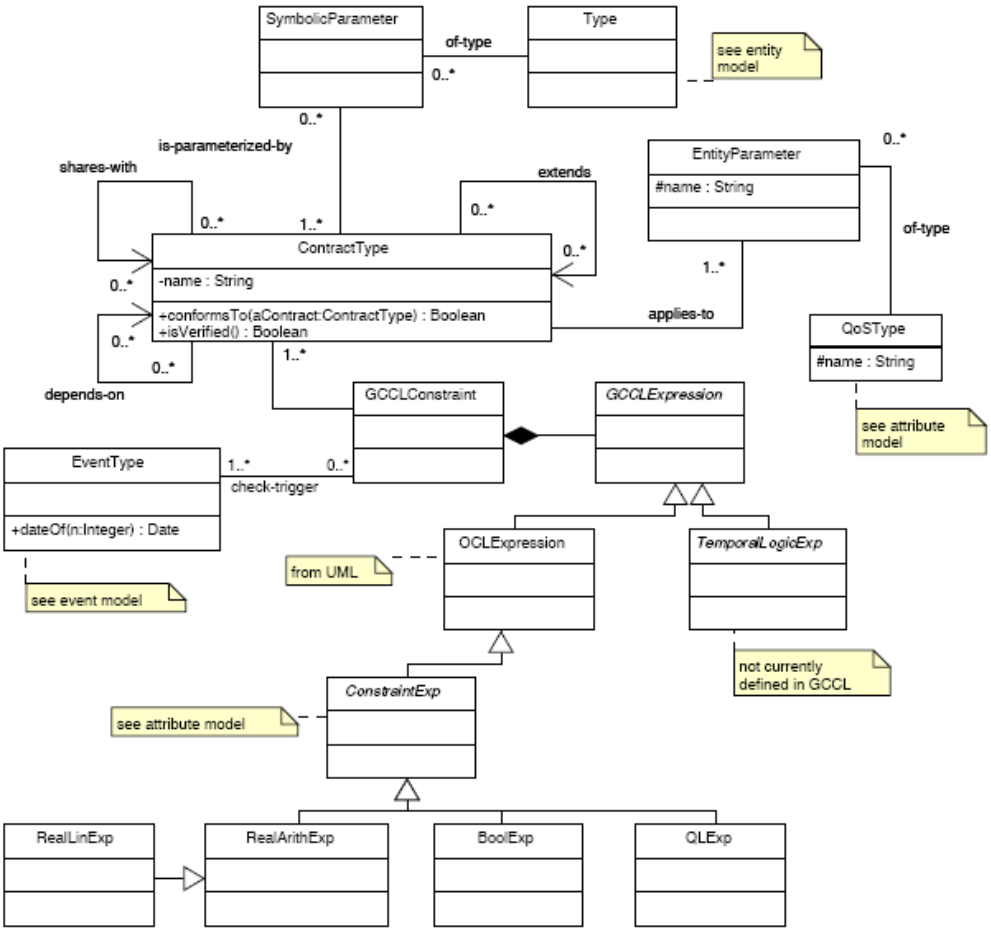


FIGURE 2.14 – métamodèle GCCL de contrat

trat de la composition prise globalement. Toutefois, la notion de responsabilité n'est pas abordée dans ces différents contrats. Par ailleurs, comme dans QoSCL, la validité de l'assemblage, c'est à dire la compatibilité de ses constituants est garantie par la composition des contrats mais pas par un contrat unique réifiant des rôles, responsabilités ..., sur lequel raisonner.

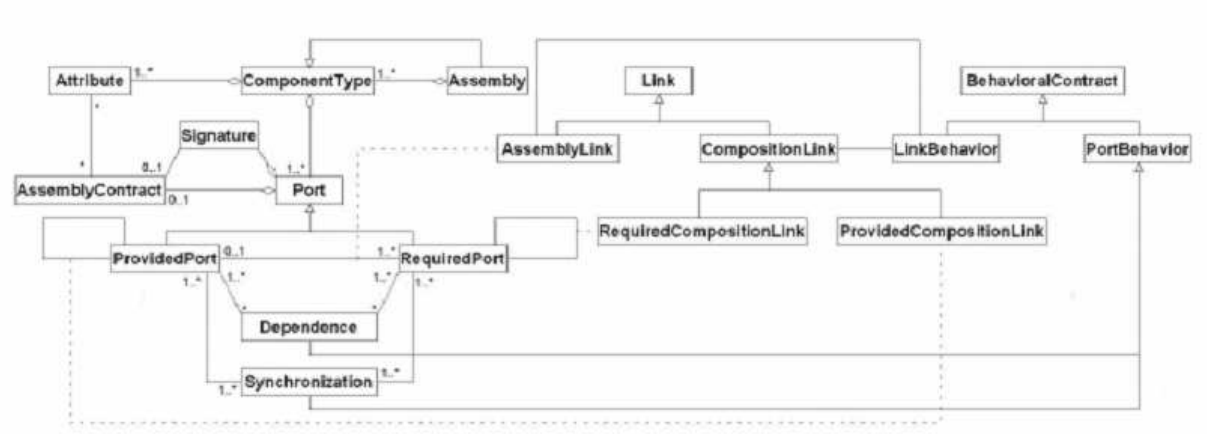


FIGURE 2.15 – métamodèle pour l'analyse de la composition

CQML. CQML [3] est un formalisme de description de qualité de service dédié aux composants (figure 2.16). Il ne réifie pas de contrat mais décrit comment en définir et vérifier un sur la base des termes qu'il fournit. CQML permet de décrire des qualités (`quality`) qui sont des contraintes sur des grandeurs de QoS (`quality_characteristic`). Un profil (`profile`) associe un ensemble de qualités à un composant, en distinguant les qualités qu'il requiert (`uses`) de celles qu'il fournit en échange (`provides`). Un contrat garantit dans ce cadre la validité d'une composition de composants : "Un contrat de QoS est un accord au moment du run-time entre des composants collaborants sur la QoS que chaque composant a la responsabilité de fournir". En résumé un contrat entre composants est l'association de l'ensemble de leurs profils. Pour que le contrat soit validé, il faut que ce que les uns fournissent corresponde à ce que les autres requièrent. Il est intéressant de voir apparaître la notion de responsabilité des composants et celle de garantie, inhérente aux contrats de la vie courante. Par ailleurs la compatibilité des spécifications fournies et requises est explicitement mise en oeuvre. Toutefois, les correspondances entre profils se font sans tenir compte du détail de l'architecture, l'ensemble des spécifications fournies doivent satisfaire à chacune de celles qui sont requises.

Requirements and Assurances Contracts. Les Requirements/Assurances Contracts [90] réalisent plus formellement les contrats décrits dans le cadre de CQML. En effet dans cette approche, la spécification des composants comprend des propriétés qu'ils requièrent et d'autres qu'ils fournissent. Il s'agit en particulier des interfaces des composants dont le comportement des méthodes est formellement spécifié. Les contrats formellement réifiés (figure 2.17) sur ces bases sont constitués de participants, les composants en interaction, et de correspondances entre propriétés requises et fournies par ces derniers. Le contrat est valide dans la mesure ou les correspondances entre propriétés requises et fournies sont satisfaites. Ces contrats présentent l'intérêt d'expliciter la notion de compatibilité entre les composants participants au contrat. Il faut noter que les correspondances sont établies sur le rapport entre spécifications de méthode requise

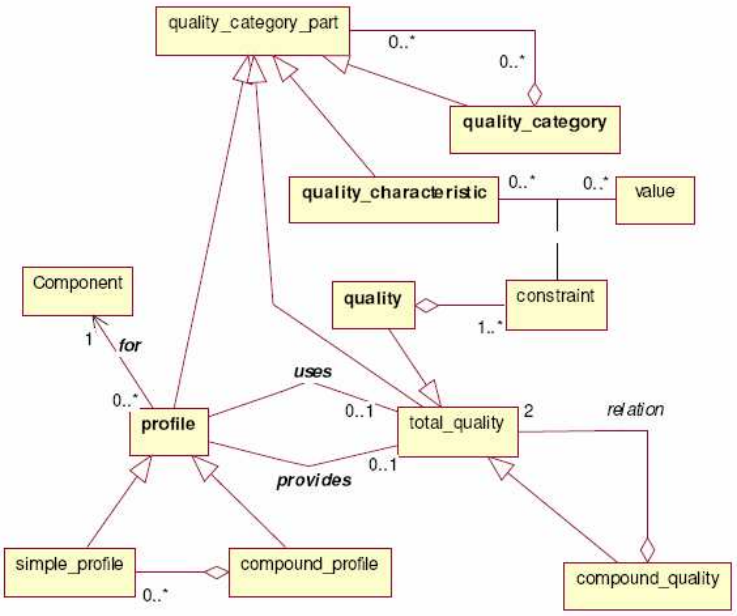


FIGURE 2.16 – Modèle de description de la QoS dans CQML

et fournie, ces clauses de contrats ne sont donc à chaque fois que binaires, même si le contrat peut faire intervenir plus de deux participants. La portée des spécifications des méthodes n'est pas clairement définie, il semble qu'elle recouvre le composant dans son ensemble.

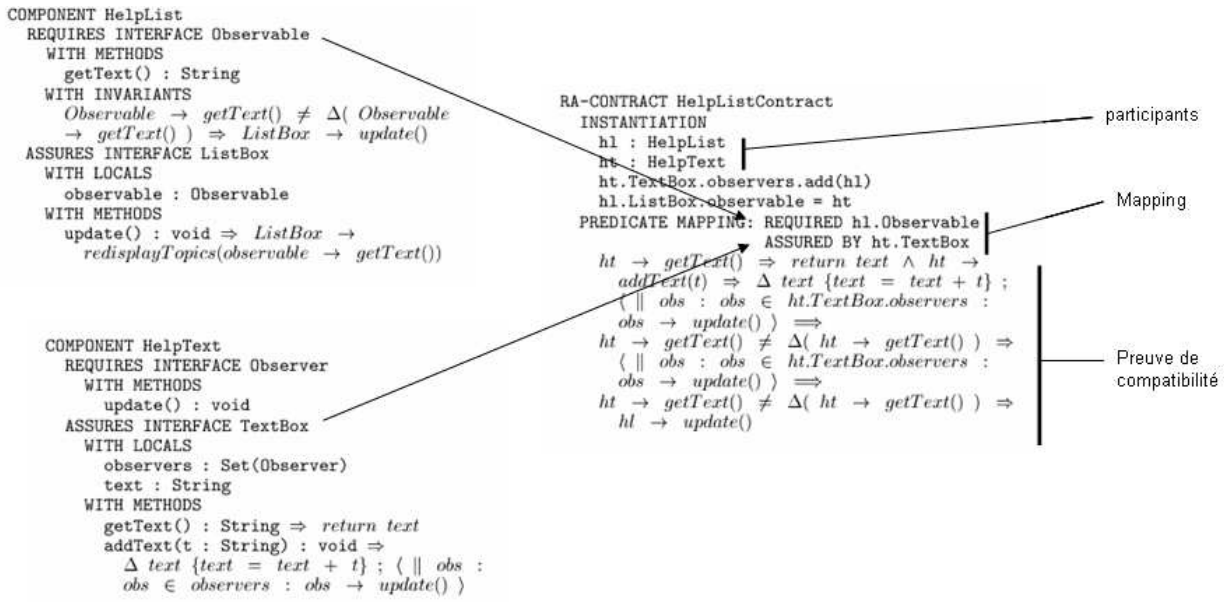


FIGURE 2.17 – Requirements/Assurances Contracts

Conclusion :
Les responsabilités et rôles des composants ne sont pas réifiés dans les contrats rencontrés. Par contre l'architecture est prise en compte à divers niveaux. L'approche la plus

simple consiste à constater que des composants collaborent et à vérifier que l'association de leurs spécifications n'est pas contradictoire, comme dans QoSCL et CQML. L'architecture est prise en compte plus finement dans les Requirements/Assurances Contract qui mettent en relation explicitement les parties fournies et requises. Enfin le modèle [102] prend en compte chaque relation entre les composants, mais ne dispose pas d'un contrat global à l'assemblage.

2.6.4 Service

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

SLA. Un métamodèle de contrat du type Service Level Agreement, SLA, a été proposé ([54]) avec la préoccupation de s'appliquer à des services à l'évolution dynamique. Le contrat réifié (figure 2.18) est constitué de trois types de composantes. Il possède une description du service ainsi que des participants au contrat, c'est à dire le fournisseur et les clients du service. Il contient aussi l'ensemble des contraintes (`Guarantee`), sur des grandeurs de QoS (`QoSParameterDefinition`), qui sont garanties par le fournisseur. Ce modèle a la particularité d'autoriser l'ajout et le retrait dynamique de contraintes (`Guarantee`). Toutefois, il n'exprime que la conformité d'un service à sa spécification et non une compatibilité entre des services. Enfin, la responsabilité vis à vis des garanties est implicitement celle du fournisseur de service bien que d'autres acteurs soient présents dans le contrat.

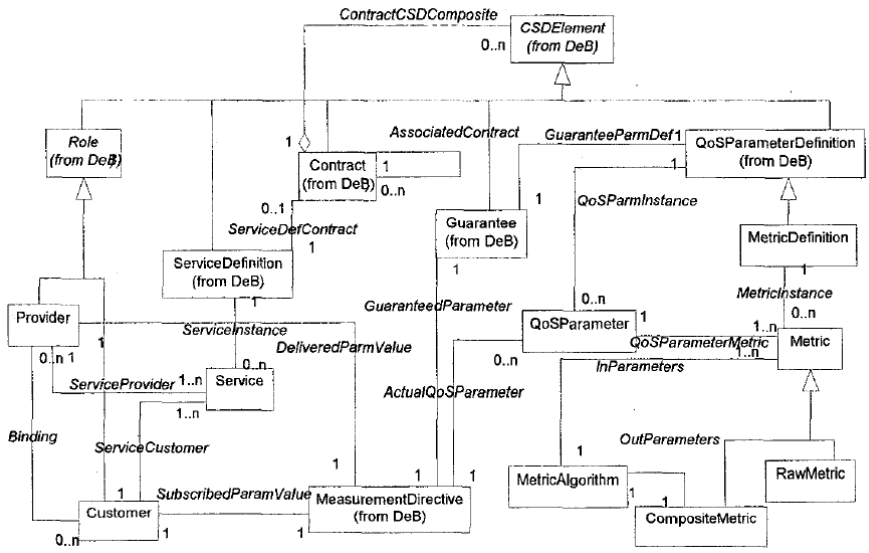


FIGURE 2.18 – modèle de contrat pour des services dynamiques

WSLA. WSLA [66] est un langage de description de SLA pour les WebServices conçu par les mêmes chercheurs que le précédent modèle, mais plus évolué. A nouveau le contrat est composé de trois parties (figure 2.19). Une partie contient les participants

au contrat, qui en plus des clients et du fournisseur, comprend des services tiers intervenant dans la vérification des clauses du contrat. Une seconde partie du contrat contient l'objet du service contraint sous forme d'un ensemble de paramètres de qualité de service, qui peuvent être mesurés par des participants tiers. Enfin le contrat présente un ensemble d'obligations portant sur ces paramètres qui peuvent être évaluées par des participants tiers. Il faut noter que les obligations sont non seulement des contraintes sur les paramètres mais aussi des obligations d'actions. Par ailleurs ces obligations peuvent être associées à n'importe quel participant du contrat, client ou fournisseur, le contrat exprime donc plus que la satisfaction d'un service à une spécification. On peut aussi remarquer que dans [55], le cycle de vie complet du contrat est détaillé, depuis sa négociation jusqu'à sa terminaison.

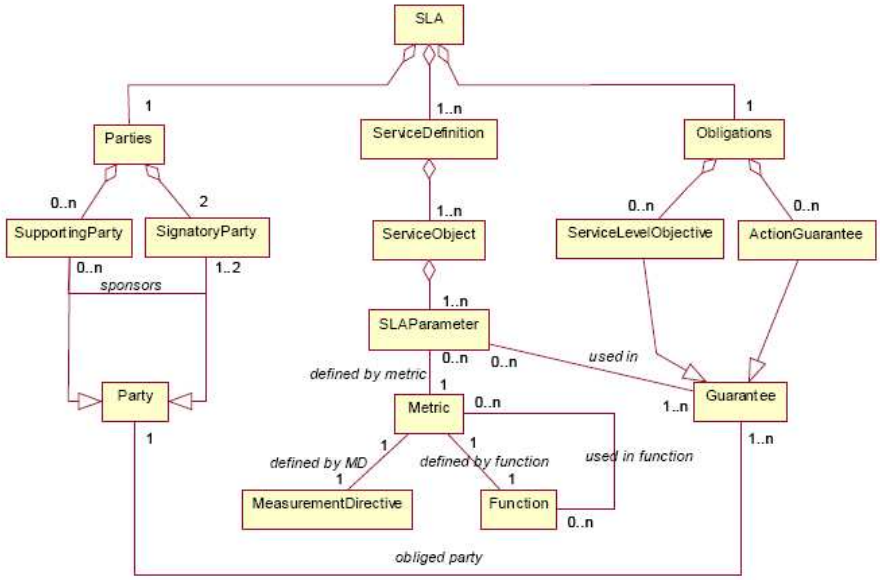


FIGURE 2.19 – modèle de contrat WSLA

WS-Agreement. Le WS-Agreement [7] est un langage basé sur XML destiné à exposer ce qu'un fournisseur de service propose, à permettre la passation d'un accord sur cette base et le monitoring de celui-ci à l'exécution du service. Un `agreement` contient trois parties. La première nommée `context` spécifie les participants de l'accord mais aussi entre autres sa durée. La seconde, les `Service Description Terms` décrit en particulier des grandeurs mesurables associées au service, par exemple un temps de réponse. Finalement la troisième partie d'un `agreement` contient les `Guarantee Terms` qui sont les contraintes sur les propriétés du service, éventuellement conditionnées par des `Qualifying Conditions`. C'est la même syntaxe qui est utilisé pour décrire ce qu'un fournisseur propose, un `agreement offer`, et ce que l'accord prévoit que le service effectue, l'`agreement`. Par exemple, dans le premier document peuvent se trouver des alternatives qui sont résolues pour aboutir au second document. Les travaux [4] proposent une formalisation des WS-Agreements. En particulier les auteurs modélisent à l'aide d'un automate les différents états de l'`agreement` sur la base d'une formalisation de celui-ci. En effet l'`agreement` est défini comme composé d'un ensemble de termes, chacun combinaison d'un service et d'une garantie, ces derniers ayant leurs états propres dont on déduit celui du terme. Par ailleurs les auteurs proposent des termes de l'accord réagissant à l'ouverture de ce dernier pour en

modifier d'autres et ainsi le rétablir. Le cycle de vie de l'accord est ainsi celui décrit dans la figure 2.20, un état `Revisited` représente l'accord modifié après rupture. Ainsi ce type de contrat est intéressant, car bien qu'il ne porte que sur la satisfaction d'une spécification par un service, il présente des clauses de renégotiation qui portent sur lui même et explicite son cycle de vie.

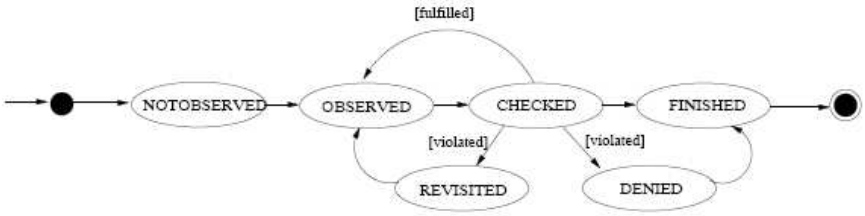


FIGURE 2.20 – cycle de vie de WS-Agreement étendu

Contrats et composition. Dans [74] les auteurs considèrent les contrats comme des outils de spécification des web services. Ils proposent un langage de description des contrats, réifiant un large éventail de propriétés des services. Celles ci comprennent la description fonctionnelle du service, son comportement (pré et post conditions), ses performances, sa fiabilité, etc... . L'intérêt de cette approche est qu'elle propose une méthodologie pour déduire le contrat d'une composition de services des contrats des services individuels. Ils traduisent pour ce faire les contrats en machines B (figure 2.21) dont les termes, c'est à dire les propriétés du contrat, supportent des annotations décrivant leur évolution en cas de composition. Par la suite, pour chaque opérateur de composition, on définit pour chaque propriété, le résultat de sa composition suivant son annotation, afin de déduire la machine B résultante, et par suite le contrat de la composition. Diverses vérifications doivent être effectuées pour valider la compatibilité des contrats, la satisfaction des invariants par celui résultant etc. Le trait remarquable de cette approche est qu'elle définit d'une part des opérations de composition sur les contrats, et d'autre part qu'elle établit une correspondance entre les opérations sur l'architecture et celles sur les contrats. Par ailleurs les opérateurs de composition ne sont pas limités, leur sémantique étant définie par rapport à l'évolution des propriétés composées.

Contrats et politiques. Dans [86] les contrats contraignent les relations entre les objets exposés par les services de processus métiers. Ils sont composés (figure 2.22) de participants qui y jouent des rôles (`Who`), de leurs objets qui sont les relations entre ce qu'exposent les participants (`What`), et des contraintes à vérifier sur ces relations pour que le contrat soit rempli (`How`). Il faut toutefois noter que les contraintes à vérifier sont exprimées sous formes de règles ECA associées à la relation qui doivent être exécutées pour que celle-ci soit satisfaite. Par exemple, une relation spécificiant qu'un service de réservation de billet (SRB) émet un billet est :

"émettre (SRB, billet)"

cette relation est contrainte sur sa date par :

Evénement : émettre (SRB, billet)

Contrainte : date (Evénement) < date_limite

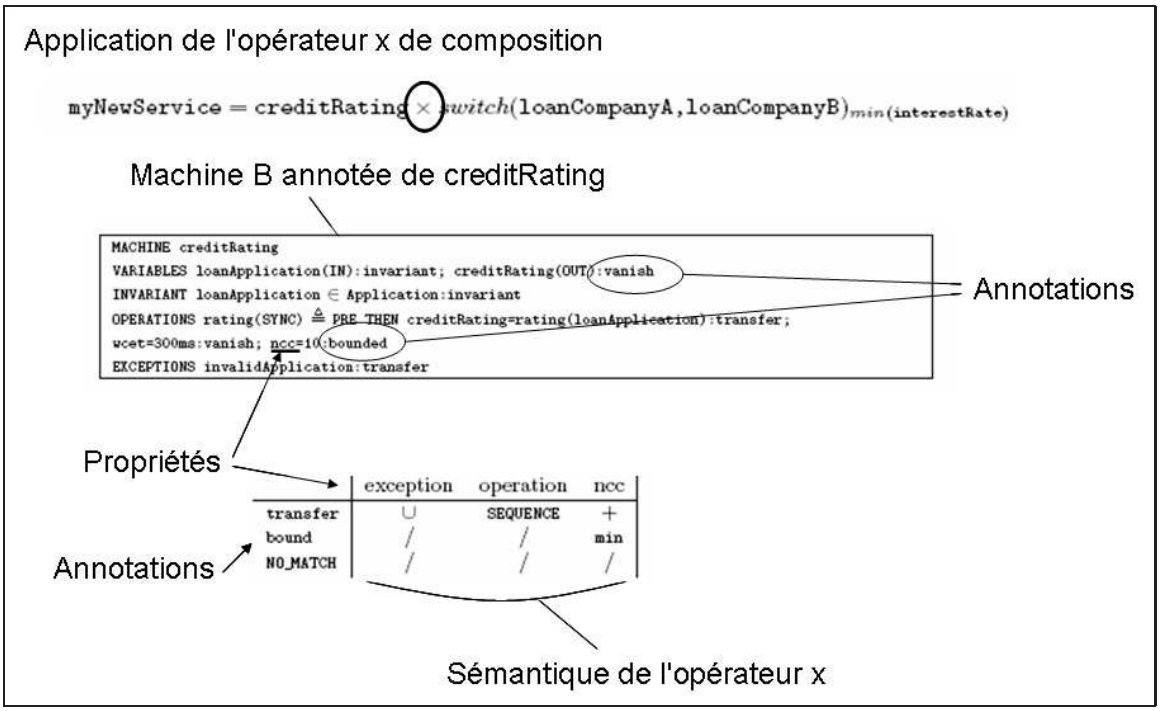


FIGURE 2.21 – méthodologie de composition des contrats

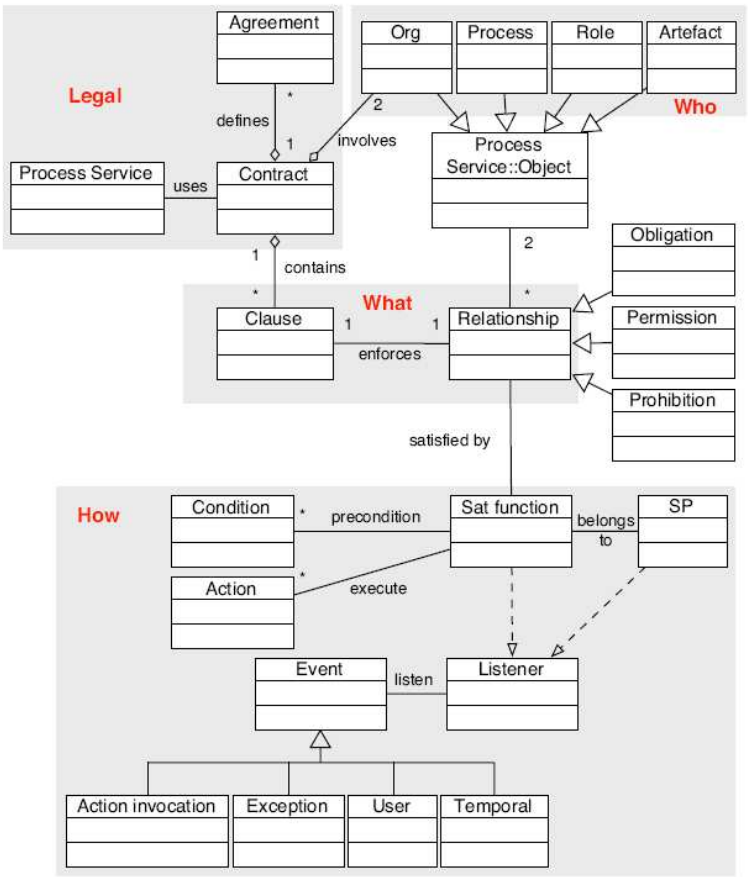


FIGURE 2.22 – contrats et politiques

Action : autoriser (émettre (SRB, billet))

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

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

2.6.5 Agents

Par rapport aux différentes formes de contrat sur les objets, composants, services les contrats sur les agents sont ceux qui, sans doute, se rapprochent le plus de ceux de la vie quotidienne. Leurs formalisations réifient les acteurs du contrat et les clauses, en donnant à ces dernières une sémantique déontique d'obligation, permission etc. Par ailleurs, on peut noter que certaines distinguent et intègrent les contraintes sur les états du système de celles sur les actions des acteurs [59], [10]. Plusieurs explicitent aussi le cycle de vie du contrat. Ainsi la durée d'application d'une clause est décrite dans [59], tandis que la composition même du contrat peut changer au cours du temps dans [28]. D'autres travaux expriment la différence entre violation de contrat et exception ou incorrection informatique [40]. Enfin, une algèbre de contrats est proposée par [10], basée sur un formalisme capable d'exprimer que les agents assurent des garanties et comptent sur des hypothèses.

2.6.6 Bilan

Nous pouvons remarquer que les modèles de contrat ont été développés en rapport avec les préoccupations centrales des entités qu'ils contraignaient. Ainsi, les contrats appliqués aux objets réifient essentiellement des contraintes sur leurs comportements et interactions, mais la notion de garantie n'est pas clairement explicitée. Par contre les modèles de contrat appliqués aux services réifient clairement les responsabilités ainsi qu'un ensemble très divers de propriétés aussi bien fonctionnelles que non fonctionnelles auxquelles le service doit adhérer.

Les modèles de contrat appliqués aux composants considèrent le problème de la garantie de leur composition bien qu'ils ne la contraignent pas toujours explicitement. Il s'agit parfois d'évaluer la compatibilité de contrats portés par des composants associés et non de confronter au sein d'un contrat leurs spécifications. Toutefois ils présentent souvent, à des degrés variables, des considérations architecturales. La notion de propriété requise et fournie associée au paradigme composant facilite l'appréhension de leur collaboration en termes d'échanges et donc de contrat. Globalement, chacun des modèles de contrat réifie les principes nécessaires à la garantie de son objet et à son intégration dans l'architecture qu'il contraint. Mais aucun ne regroupe tous les principes que nous avons identifiés. Il est par ailleurs rare de trouver des clauses de "second

ordre", c'est à dire contraignant le contrat lui même, comme par exemple le conditionnement d'une clause par la réalisation d'une autre.

Vis-à-vis de l'IDM, les modèles de contrat que nous avons vu, sont destinés à guider la mise en oeuvre de la vérification des spécifications dans le cadre de systèmes concrets. Cette mise en oeuvre peut passer par de la génération de code, ou l'instanciation et l'assemblage d'outils existants.

2.7 Conclusion

Il existe de nombreux modèles d'objets, de composants et de services, mais les systèmes qui les mettent en oeuvre, ont en commun de reposer sur leur collaboration. L'importance de cette dernière est d'ailleurs mise en relief par l'architecture explicite des systèmes à base de composants et services. Historiquement, c'est dans le cadre des objets que la conception par contrat [71] a été définie, avec un contrat sous la forme d'une extension du langage de programmation destinée à exprimer et vérifier la conformité des objets à leur spécification. La compatibilité n'est envisagée qu'entre classes parentes, les assertions qui forment les contrats sont exprimées dans un unique langage et évaluées lors de l'exécution du programme. Parallèlement, de nombreux formalismes de spécification ont été développés pour décrire les aspects aussi bien fonctionnels que non-fonctionnels des objets, composants et services. Nous avons vu que la plupart sont modulaires. Une bonne part de ceux que nous avons étudiés permettent de vérifier à la fois la conformité et la compatibilité de leurs spécifications. De plus, ceux-ci sont d'un degré d'abstraction intermédiaire qui rend envisageable non seulement leur application à des systèmes variés mais encore leur traitement mécanique. De manière plus générale, le paradigme hypothèse-garantie, qui sous tend un certain nombre de ces formalismes, nous est apparu comme un moyen simple et minimal de décrire les interactions entre une entité et son environnement, avec en plus des liens vers la notion de responsabilité. Toutefois, il est remarquable que les travaux sur la compatibilité de ce type de spécifications ne fassent pas intervenir l'organisation architecturale de leurs porteurs. Enfin pour ce qui est des composants et des services, divers modèles de contrat ont été proposés. Ils considèrent tous au moins la conformité des composants et services à leurs spécifications. Mais peu réifient la compatibilité de leurs participants, et ils sont alors dans ce cas associés à une architecture ou un formalisme. De plus, hormis le cas des SLA, qui se limitent aux interactions client-serveur, les responsabilités restent implicites. Nous n'avons pas trouvé de modèle de contrat à la fois indépendant des spécificités des spécifications et de l'architecture, qui présente les propriétés que nous nous sommes données pour objectif : conformité, compatibilité, responsabilité.

CHAPITRE 3

Modèle de contrat

3.1 Introduction

Nous souhaitons garantir qu'un assemblage d'objets, de composants ou de services est valide vis à vis de son fonctionnement (au sens large, configuration comme exécution). Or nous avons vu dans l'état de l'art que ce fonctionnement est constitué d'échanges. Notre démarche "par contrat" consiste à expliciter et à vérifier que chaque participant à l'échange reçoit ce qu'il requiert et fournit ce que les autres participants en attendent. L'assemblage définit la nature et les protagonistes de ces échanges, tandis que les spécifications relatives aux interactions des participants décrivent leurs propriétés. Aussi comme dans les métamodèles de l'état de l'art, nous décidons de faire du contrat une fonction dont la valeur de retour reflète l'état du système. Vis-à-vis de l'assemblage et de ses spécifications, notre contrat se positionne entre ceux-ci, qu'il prend en entrée, et un booléen, qu'il retourne et qui reflète la validité de l'assemblage du point de vue de son fonctionnement.

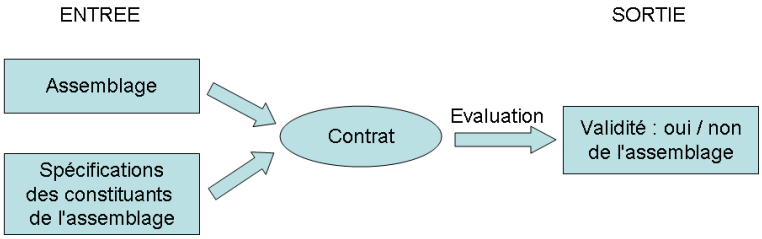


FIGURE 3.1 – Le contrat comme une fonction

Plus précisément, dans l'introduction, nous avons explicité la validité de l'assemblage en la faisant reposer sur la conformité des éléments d'architecture à leurs spécifications et sur la compatibilité de ces dernières. La plupart des formalismes que nous avons rencontrés dans l'état de l'art autorisent l'évaluation de l'une, l'autre ou de ces deux propriétés. Le contrat, qui accepte plusieurs formalismes, déduit donc l'état de validité de l'assemblage de l'évaluation de ces propriétés. Nous en concluons que le contrat occupe la position décrite sur la figure 3.2.

Par rapport aux propriétés de conformité, compatibilité et responsabilité que nous nous sommes données comme objectifs, ces positionnements nous ont guidé dans l'-analyse et la conception de notre modèle de contrat. Ils nous ont permis d'appréhender ce que le contrat consommait, soit une architecture et des spécifications, et ce qu'il produisait, c'est à dire des propriétés relatives à ces dernières qu'il réifie, organise et emploie. Nous présenterons donc en premier (partie 3.3) les choix qui ont guidé notre conception. Ensuite nous détaillerons la constitution du modèle qui en a résulté (partie 3.4). Enfin nous envisagerons son utilisation associé à des spécifications compositionnelles (partie 3.5).

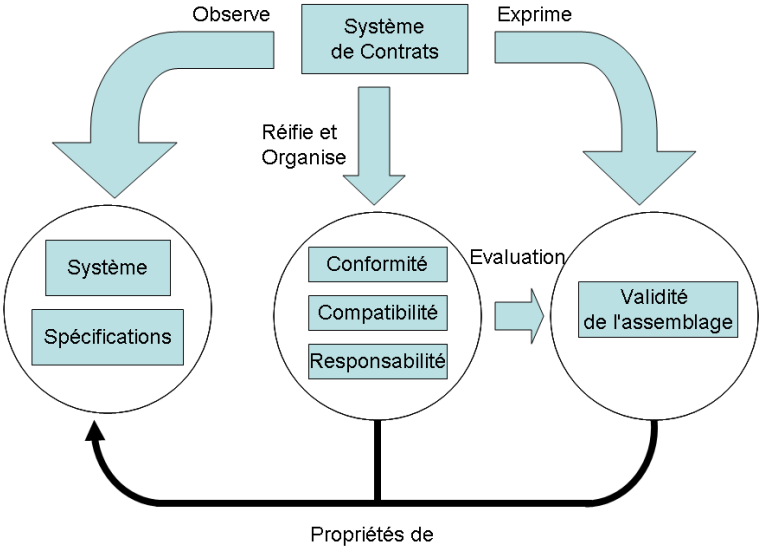


FIGURE 3.2 – Positionnement du contrat

3.2 Système exemple

Dans le cadre de la description du modèle de contrat nous nous appuierons sur des exemples issus d'un petit système de composants, conçu uniquement à des fins d'illustration. Ce système repose sur un modèle de composants classique dans lequel les composants sont connectés les uns aux autres via des interfaces fournies et requises, et peuvent être composites. Un de nos objectifs est de découpler notre modèle de contrat de la nature concrète de l'architecture, aussi la nature exacte des éléments de l'exemple n'est donc pas importante. Par contre, le fait que ce découplage permette de raisonner sur l'architecture indépendamment de l'implémentation concrète de cette dernière, suppose que pour un framework architectural donné un expert effectue un travail d'intégration de son système dans les termes de notre modèle.

3.2.1 Description

Nous considérons pour notre exemple un système de régulation de vitesse d'une automobile inspiré de [67]. Ce système est contrôlé à l'aide de trois boutons : *resume, on* and *off*. Quand la voiture roule et que *on* est pressé le système enregistre la vitesse courante et maintient la voiture à cette allure. Quand l'accélérateur, le frein ou *off* est pressé, le système cesse d'agir mais retient la vitesse programmée. Si *resume* est pressé, le système accélère ou ralentit la voiture pour la ramener à la vitesse précédemment retenue. La figure 3.3 décrit l'architecture et les interfaces du système. Du point de vue de l'utilisateur d'un tel système, il est important qu'une erreur soit détectée rapidement, c'est à dire avant qu'elle n'ait des conséquences critiques, mais aussi que son diagnostic soit aussi précis que possible pour en appréhender la gravité. Par ailleurs il serait préférable que ce diagnostic soit aussi synthétique que possible, expurgé des détails d'implémentation que l'utilisateur du système ne saurait pas interpréter.

D'un point de vue externe, le composant <Car> fournit une interface *sns*, du type *Sensor*, dont les méthodes permettent au conducteur (`<User>`) de piloter l'automobile. De manière interne, <Car> est constitué de deux sous-composants. Le composant <Crui-

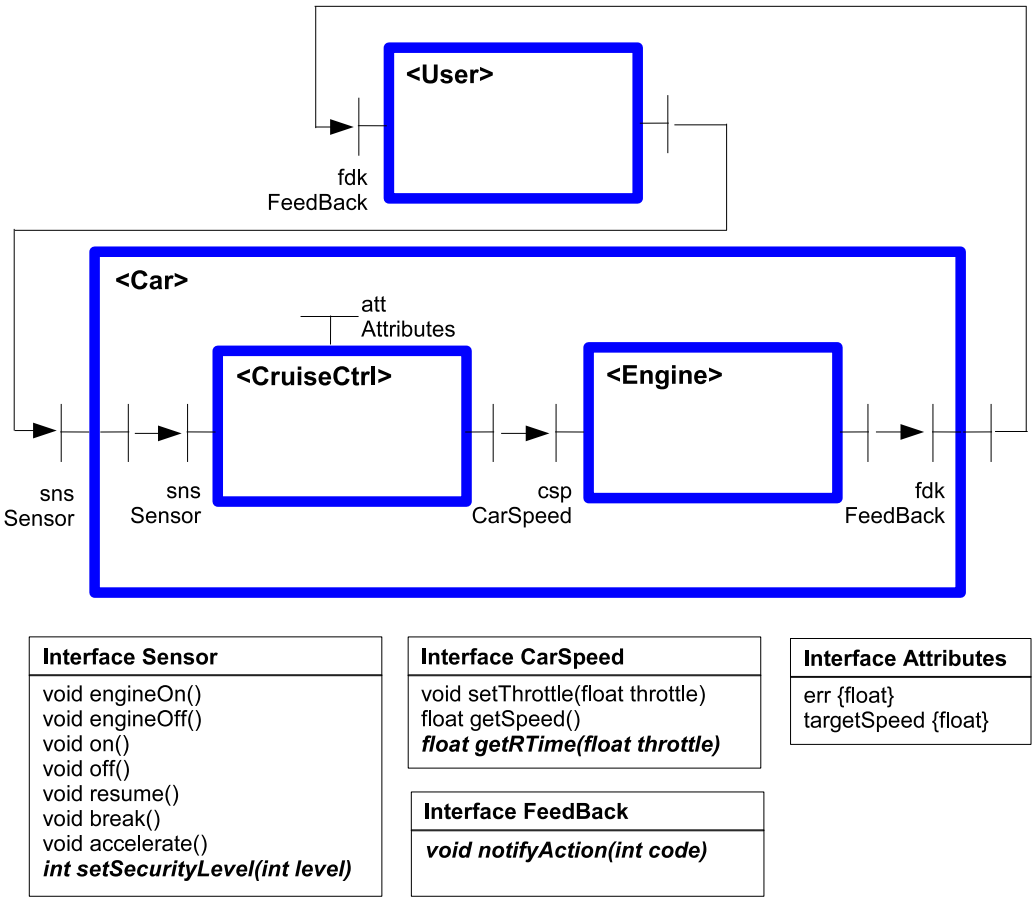


FIGURE 3.3 – Système exemple

seCtrl> est le système principal de commande, fournissant une interface *Sensor* et les attributs représentant la vitesse programmée et le code d'une éventuelle erreur. Il requiert par ailleurs une interface *CarSpeed* afin d'interagir avec le moteur. Le moteur, <Engine>, fournit une interface *csp*, du type *CarSpeed*, dont les méthodes permettent de fixer l'accélaration et d'obtenir la mesure de la vitesse courante. Nous pouvons au passage constater que pour le concepteur du système cette division des tâches n'est intéressante que s'il peut avoir l'assurance que chaque partie remplit bien la sienne, de manière compatible avec les autres.

Via l'interface `FeedBack` l'utilisateur de la voiture, `<User>`, se voit notifier des actions entreprises par le moteur `<Engine>` pour suivre les ordres du contrôleur de vitesse `<CruiseCtrl>`. Dans une voiture réelle, cette interface pourrait prendre la forme d'un afficheur sur le tableau de bord tenant le conducteur informé des initiatives du système de contrôle de la vitesse. Par ailleurs nous faisons maintenant l'hypothèse que le système de contrôle de la vitesse est relié à des capteurs détectant d'éventuels obstacles devant ou derrière la voiture. Suivant la présence, la distance, la vitesse de tels obstacles le système de contrôle de la vitesse peut être amené à modifier la vitesse de la voiture pour conserver un certain niveau de sécurité. Pour fixer ce niveau de sécurité nous avons ajouté la méthode `setSecurityLevel` à l'interface `Sensor`. Cette méthode fournit en retour à un niveau de sécurité désiré un indicateur du niveau de confort que le système peut alors garantir, sachant que plus le niveau de sécurité est élevé, moins le confort sera prioritaire donc affecté d'un indicateur faible. Enfin au sein du système le moteur `<Engine>` est capable de donner le temps qu'il lui faut pour atteindre une certaine accélération grace à la méthode `getRTime` de l'interface `CarSpeed`.

3.2.2 Spécifications du système

Afin d'autoriser la prise en compte de propriétés variées, nous avons pour objectif de découpler notre modèle de contrat des formalismes de spécification et de leurs interprètes et outils d'évaluation. Cette approche offrira une vision uniforme de ces propriétés à l'utilisateur du système, car elle ne considérera que les notions de conformité et de compatibilité des éléments et de leurs spécifications. Par contre à nouveau elle suppose, de la part d'un spécialiste du formalisme, l'intégration de ce dernier et de ses outils dans notre modèle. Nous considérerons pour notre exemple les formalismes CCLJ (cf 2.4.2.2) et Behavior Protocol (cf 2.4.2.3).

3.2.2.1 CCLJ

A l'aide de CCLJ, nous pouvons exprimer que le composant `<Car>` va demander quoiqu'il arrive, via l'interface `sns` qu'il requiert, un niveau de sécurité supérieur à 3. Tandis que composant `<CruiseCtrl>` n'accepte, via l'interface `sns` qu'il fournit, que les niveaux de sécurité inférieurs à 5.

```
S1 :
On <Car>
  context int sns.setSecurityLevel (int level)
    pre : level > 3

S2 :
On <CruiseCtrl>
```

```
  context int sns.setSecurityLevel (int level)
    pre : level < 5
```

et d'autre part il est possible de spécifier que le composant <Engine> garantit un temps de réaction à une accélération inférieur à 5 unités. Tandis que le composant <CruiseCtrl> requiert lui un temps de réponse inférieur à 10 unités.

```
S3 :
On <Engine>
  context float csp.getRTime (float throttle)
    post : retour < 5

S4 :
On <CruiseCtrl>
  context float csp.getRTime (float throttle)
    post : retour < 10
```

3.2.2.2 Behavior Protocol

Les Behavior Protocol permettent de décrire les séquences d'appels entrants et sortants qu'un composant accepte. Nous les avons utilisés pour spécifier les comportements des <CruiseCtrl>, <Engine> et <Car>.

- <CruiseCtrl>:

  ```
  (1)  (?sns.on; !csp.setThrottle*; ?sns.off)*
  ```

  Le composant <CruiseCtrl> commence par attendre un appel de la méthode on() sur son interface fournie sns, puis il émet une série indéfinie d'appels à setThrottle, jusqu'à ce qu'il reçoive un appel à méthode off() de son interface sns. Cette séquence peut être répétée un nombre indéfini de fois.

- <Engine>:

  ```
  (2)  (?csp.setThrottle; !fdk.notifyAction)*
  ```

  L'<Engine> attend un appel à la méthode setThrottle() qu'il fournit. Quand il le reçoit il émet un appel à notifyAction(), puis attend à nouveau un appel à la méthode setThrottle(). Ce cycle peut se produire un nombre indéfini de fois.

- <Car>:

  ```
  (3)  (?sns.on; !fdk.notifyAction*; ?sns.off)*
  ```

  Le composant <Car> englobe les deux précédents. Il attend un appel à la méthode on(), celui-ci arrivé il émet une série indéfinie d'appels à notifyAction(), jusqu'à ce qu'il reçoive un appel à la méthode off(). Ce cycle peut se répéter un nombre indéfini de fois.

L'intérêt de notre approche est que la personne qui écrit les spécifications n'a pas besoin d'être un expert de leur mise en oeuvre, leurs outils d'interprétation et de vérification ont déjà été intégrés au modèle de contrat. Elle n'a pas non plus à se soucier de l'implémentation du système contraint qui lui est masquée par le modèle de contrat.

3.3 Analyse et choix de conception

Dans cette partie nous allons dans un premier temps étudier les propriétés réifiées par le modèle de contrat (3.3.1). Puis nous aborderons la portée de ce dernier, tant du point de vue des architectures (3.3.2) que de celui des formalismes (3.3.3) qu'il va accepter. Nous terminerons par une synthèse de ces différents aspects (3.3.4).

3.3.1 Propriétés du système réifiées par le contrat

Comme nous l'avons vu précédemment, la propriété de validité d'un assemblage se déduit de ses propriétés de conformité, compatibilité et responsabilité. Nous étudierons donc l'expression et la mise en oeuvre de chacune de ces dernières. Il faut noter que le contrat doit produire une expression de ces propriétés mais aussi faire appel aux outils convenables pour obtenir leur évaluation, afin d'en déduire l'évaluation de la validité de l'assemblage.

3.3.1.1 Conformité

Définition

> Nous définissons la conformité comme une propriété d'un élément du système contraint par rapport à sa spécification. Il y a conformité entre un élément de l'architecture et sa spécification, si la réalisation concrète (exécutable) du premier satisfait bien à la seconde. Nous considérerons la spécification comme une contrainte qui peut être ou ne pas être satisfaite par la réalisation concrète de l'élément.

Une forme d'évaluation de la conformité

Les éléments de système que nous envisageons étant des composants ou des services, leurs spécifications sont communément "boites noires" même si certains travaux sont en faveur de boites grises. Ainsi notre connaissance de ces éléments se limite à leurs échanges avec leur environnement et aux spécifications qui contraignent ces derniers. Il est donc intéressant de noter que ces échanges peuvent avoir lieu à différents stades du cycle de vie des éléments. Durant la phase de configuration architecturale, des éléments fournis peuvent par exemple être mis en relation avec d'autres requis, tandis que durant la phase d'exécution, données produites et requises sont échangées. Par ailleurs l'évaluation de la conformité d'une spécification est effectuée à l'aide d'un outil qui est associé à son formalisme et qui prend en entrée les observations des échanges. Nous pouvons constater qu'il y a essentiellement trois objets d'observation sur lesquels évaluer la conformité d'un élément à sa spécification :

- observation de grandeurs d'exécution : sans observer la réalisation d'un élément il est possible d'observer ce que son exécution expose à son environnement, c'est à dire les données qu'il expose ou émet au cours de celle-ci (par exemple les paramètres d'une méthode pour l'évaluation de simples assertions),
- observation de paramètres de configuration : un élément d'architecture peut exposer des paramètres modifiables seulement durant sa phase de configuration

et qui peuvent être l'objet de l'observation (ces paramètres peuvent par exemple être utilisés dans une assertion en CCLJ),

- observation de la configuration architecturale : il s'agit d'observer à la fois les propriétés architecturales des éléments définies au moment de leur conception, et les relations établies entre eux au moment de la construction (ou reconfiguration) du système contraint (par exemple le nombre d'éléments connectés à une interface donnée peut être limité),

Dans notre exemple, si nous considérons la spécification suivante du `<CruiseCtrl>` :

```
(?sns.on; !csp.setThrottle*; ?sns.off)*
```

La vérification de la conformité du `<CruiseCtrl>` reposera sur l'observation des appels émis ou reçues sur les méthodes `on()` `off()` et `setThrottle()`. En effet ces observations serviront d'entrées à un "*runtimechecker*" initialisé avec la spécification à vérifier.

Réification de la conformité

La réification de la conformité d'un élément d'architecture à sa spécification consiste à expliciter différents points :

- l'élément d'architecture dont la spécification est considérée, et qui va être l'objet d'observation,
- l'expression de la contrainte contenue dans la spécification,
- les observations nécessaires à l'évaluation de la contrainte : elles seront déduites de l'expression de la contrainte et de l'outil d'évaluation associé au formalisme de celle-ci,

Problématique de l'observation

Il se pose alors le problème complexe de la définition et de la mise en oeuvre de l'observation d'un système. Il faut dans un premier temps parvenir à décrire ce qu'on veut observer et quand on souhaite l'observer. Notre objectif n'est pas de produire une taxonomie de ce qu'il est possible d'observer sur un système et à quels moments il est possible de faire des observations, mais de définir une approche pragmatique utile à la vérification des contraintes des spécifications. Nous procéderons donc en faisant ressortir des invariants dans les objets et moments d'observation :

- élément d'architecture attaché : la nature de la grandeur à observer change avec celle de la spécification (QoS, comportement, etc...), toutefois de par la nature des spécifications que nous retenons, les grandeurs sont toujours définies par rapport à un élément donné de l'architecture. En effet les spécifications sont par hypothèse modulaires, aussi ce qu'elles contraignent se définit obligatoirement par rapport à l'élément auquel elles sont associées. Ainsi quelque soit la nature de l'observation (temps de réponse, paramètre, configuration etc...), un constituant architectural de cet élément (interfaces, méthodes etc...) devra être mentionné dans la description de l'observation pour faire le lien avec le système observé,

- déclencheur : que l'observation soit discrète ou continue, la grandeur qu'elle retourne doit être arrêtée à un instant donné, ainsi elle peut être définie par un ou plusieurs événements. Toutefois il faut noter que ces événements peuvent être originaires du système contraint ou de son environnement. Il peut aussi bien s'agir d'une interception de l'exécution que d'un timer externe qui se déclenche, voire de la réalisation d'une condition portant à la fois sur l'intérieur et l'environnement du système. D'une manière plus générale l'observation doit disposer de déclencheurs sensibles à ces différentes sortes d'événements et conditions. Dans le cas ou les événements sont originaires du système, ils doivent se déduire à nouveau de l'élément auquel est associé la spécification pour des raisons de modularité.

En dehors de ces points, la part variable de la description des observations ne peut être modélisée par un système général de contrat car elle est propre à un formalisme ou une architecture donnée.

3.3.1.2 Compatibilité

Dans notre approche la compatibilité a pour objet d'exprimer que les échanges entre éléments de l'architecture se déroulent correctement, c'est à dire que les attentes des uns sont satisfaites par ce que fournissent les autres.

Définition

> Deux, ou plus, entités sont compatibles si : toute grandeur émise par l'une est acceptable par celle(s) qui la reçoive(nt). Du point de vue des spécifications, cela revient à dire que toute grandeur émise qui satisfait à la spécification de l'émetteur, doit satisfaire à la (aux) spécification(s) du(es) récepteur(s).

Cette définition est une forme adaptée de celle que nous avons donné à la fin de l'introduction, soit :

> la non-contradiction de spécifications qui, attachées à des entités distinctes, contraignent des grandeurs communes

Elle a pour objet de ramener la notion de compatibilité aux échanges entre les éléments du système. En effet, la définition de compatibilité que nous avons donné initialement couvrait toutes les sortes de partage de grandeurs entre différents éléments du système et pas seulement les échanges.

Une forme d'évaluation de la compatibilité

Comme nous l'avons vu dans le paragraphe précédent, les spécifications modulaires contraignent ce que les éléments d'architecture s'échangent d'un point de vue local à chacun, aussi leur comparaison devrait donc permettre de prévoir si elles sont compatibles. Suivant les formalismes l'expression de la compatibilité de deux ou plusieurs spécifications peut prendre différentes formes qui peuvent être évaluées de différentes manières. Dans notre exemple, si nous considérons les spécifications du comportement

des composants <Engine> et <CruiseCtrl>, ces deux composants seront compatibles si l'expression :

$$((?csp.setThrottle; !fdk.notifyAction) * \sqcap (?sns.on; !csp.setThrottle*; ?sns.off)*)$$

interprétée sous la forme évaluable suivante :

```
parallelCheck ((?csp.setThrottle; !fdk.notifyAction)*,  // <Engine>
     (?sns.on; !csp.setThrottle*; ?sns.off)*) // <CruiseCtrl>
```

dans laquelle parallelCheck dénote la vérification par model checking de la composition parallèle des spécifications, est évaluée à vrai. <CruiseCtrl> envoie des messages setThrottle à <Engine>. L'évaluation vérifie que chaque fois que le message est émis il est bien attendu. Si chaque fois que le message est émis il est attendu, alors chaque fois que la spécification de <CruiseCtrl> est vraie (pour le message setThrottle) la spécification de <Engine> l'est aussi, et alors les deux spécifications sont compatibles. Si nous considérons maintenant les spécifications en CCLJ, S3 et S4 (cf 3.2.2.1), l'interprétation en CCLJ de leur compatibilité est la suivante :

```
On <Engine>
  context float csp.getRTime (float throttle)
    post : (retour<5) => (retour < 10)
```

La valeur retour, retour de la méthode getRTime, est la grandeur échangée entre les composants. Elle exprime que les composants sont compatibles si pour chaque retour de la méthode getRTime, la post condition du client (récepteur de la grandeur) retour < 10 est vraie tant que la post condition du serveur (émetteur de la grandeur) retour < 5 l'est. L'évaluation de cet exemple est triviale et ne requiert pas d'observation de la grandeur échangée, mais ce n'est pas le cas général.

Nous pouvons donc voir que les observations nécessaires à l'évaluation de l'expression de compatibilité différeront suivant l'expression (cf l'exemple), le formalisme et l'outil d'évaluation disponible. Par exemple, la compatibilité de spécifications comportementales en Behavior Protocol pourra être évaluée à l'aide d'un modelchecker, donc sans observation de l'exécution, tandis que celle de pré et post conditions de CCLJ le sera en observant les valeurs des paramètres et valeurs de retour fournis par l'exécution du système.

Réification

A nouveau, il ne s'agit pas pour nous d'établir une taxonomie de toutes les formes de compatibilité que le contrat devra prendre en compte mais d'identifier ce qu'elles ont en commun. C'est cette partie commune qui devra être modélisée dans le contrat car indépendante du formalisme et de l'architecture contrainte.

Ainsi ce qui est commun aux différentes expressions de compatibilité est qu'elles reflètent les relations contenues dans l'architecture contrainte par le contrat. En effet les échanges entre éléments de l'architecture sont définis par les relations entre ceux-ci.

Toute grandeur exposée ou émise par un élément et consommée ou requise par un autre l'est via une relation établie entre ces deux derniers. Il convient donc pour chaque relation d'explorer les spécifications de ses extrêmités pour former l'expression exprimant que chaque spécification contraignant une grandeur échangée d'un côté de la relation est compatible avec celle(s) contraignant cette même grandeur de l'autre côté de la relation. De ce fait, il faut noter qu'une expression de compatibilité, établie pour un ensemble d'éléments d'architecture, peut ne concerner qu'un sous ensemble des relations les unissant, et/ou un sous ensemble de leurs spécifications. Ainsi toute expression de compatibilité ne vaut que si elle accompagnée des hypothèses qui décrivent sa portée aussi bien en termes de relations architecturales que de spécifications. Le contrat devra donc expliciter :

- l'expression de la compatibilité à évaluer ,
- l'ensemble des relations dont l'expression de la compatibilité garantit la validité,
- les observations éventuellement nécessaires à l'évaluation,
- l'ensemble des spécifications prises en compte pour établir l'expression de la compatibilité

3.3.1.3 Responsabilité

Définition

Nous définissons la responsabilité comme une propriété qui lie une entité (logicielle ou non) au résultat de l'évaluation de sa conformité à sa spécification. La sémantique de cette propriété est que l'entité responsable est celle à blamer en cas de non-satisfaction de la spécification, c'est à dire celle qui est considérée comme à l'origine de la non-satisfaction. Toutefois, pour être déclarée responsable, une entité doit être causalement liée au résultat de l'évaluation de la spécification, en effet l'objectif de cette propriété est de guider les actions de rétablissement de la conformité.

Sémantique

Dans le cadre de cette définition, nous pouvons distinguer deux grandes approches de la responsabilité suivant la nature de la cause :

- le responsable est à l'origine du résultat de l'évaluation : une modification du responsable entrainera alors une modification du résultat de l'évaluation. Dans cette approche, les composants ou services contraints sont les responsables de leur conformité à leurs spécifications, puisqu'ils sont à l'origine directe des observations qui sont utilisées pour l'évaluation de la conformité. Modifier un composant ou un service modifiera le résultat de l'évaluation à la conformité. Cette approche privilégie une causalité directe entre les paramètres et le résultat de l'évaluation. Par exemple, une méthode peut être considérée comme responsable du résultat de l'évaluation d'une post-condition sur sa valeur de retour,

- le responsable est celui dont l'action peut changer le résultat de l'évaluation : dans ce cadre sont considérées les entités capables d'influer sur l'origine du résultat. Cette approche considère comme responsables non plus les composants ou les services, mais leurs développeurs, ou les assembleurs de l'application qui les ont choisi. Une causalité indirecte est dans ce cas considérée, puisque ce n'est plus l'origine (composant, service) des paramètres de l'évaluation qui est la cause, mais ce qui a produit cette origine (développeur, assembleur, etc...).

Nous considérerons la responsabilité associée à la cause directe du résultat de l'évaluation, c'est à dire les composants et services comme responsables. En effet si nous considérions une causalité indirecte, c'est à dire par exemple les développeurs comme responsables, ce n'est plus les échanges entre composants qu'il faudrait étudier mais ceux entre les développeurs. Un développeur ne garantit pas la conformité de son composant à un autre composant avec lequel le sien interagit, il garantit la conformité à un autre développeur producteur du composant qui interagit avec le sien. Or cette garantie peut alors se teinter de considérations complètement externes à la nature de l'échange entre les composants, comme des considérations commerciales etc...

Une forme d'évaluation de la responsabilité

Si nous considérons la spécification d'un élément d'architecture, celle-ci est évaluée sur la base d'un ensemble d'observations. Nous avons considéré jusqu'ici que ces observations mentionnaient explicitement les éléments ou parties d'éléments d'architectures desquels elles étaient issues. Ainsi il est possible de distinguer les observations qui portent sur ce qu'un élément d'architecture expose, émet ou fournit de celles sur ce qu'il reçoit, consomme ou requiert. Or un élément d'architecture, ne peut en temps que cause, n'être responsable que des spécifications qui portent sur ce qu'il expose, émet ou fournit. Il est donc nécessaire d'interpréter les spécifications, par rapport à leur élément porteur, pour en distinguer la garantie ou l'obligation, c'est à dire la partie dont le porteur est responsable.

Réification

Comme nous l'avons vu, la définition de la responsabilité relève de l'interprétation de la spécification pour en extraire une garantie telle que définie au paragraphe précédent. Nous pouvons remarquer que dans le cadre de l'approche hypothèse-garantie la spécification fait bien apparaître une garantie telle que nous la souhaitons. Mais elle fait aussi apparaître une hypothèse qui a alors pour objet de contraindre ce que le porteur de la spécification consomme, requiert ou reçoit de son environnement. Dans la sémantique hypothèse-garantie, la satisfaction de l'hypothèse conditionne la responsabilité de son porteur vis à vis de la garantie contraignant alors ce qu'il fournit ou produit. Or cette sémantique est tout à fait celle que nous avons utilisé pour définir la validité d'un assemblage, dans les termes de l'échange, utilisés dans la conformité et la compatibilité, et dans ceux de l'engagement, utilisés dans la responsabilité : chaque participant s'engageant à fournir ce que ses pairs attendent pourvu que ceux-ci lui fournissent ce dont il a besoin. L'approche hypothèse-garantie représente donc le modèle dans lequel il faudra interpréter les spécifications pour les faire entrer dans le contrat tel que nous l'avons défini. Conformité, compatibilité, responsabilité vis à vis de spécifications

hypothèse-garantie, permettent d'exprimer la validité de l'assemblage telle que nous le souhaitons.

3.3.2 Architectures acceptées par le modèle de contrat

Les contrats qui suivent les principes que nous avons donnés ne sont pas explicitement associés à une architecture spécifique. Toutefois la mise en oeuvre d'un contrat suivant notre modèle requiert de l'architecture contrainte un certain nombre de propriétés :

- Système collaboratif : comme notre définition de la compatiblité repose sur le principe d'échange entre les participants de l'architecture, nous requiérons que le fonctionnement des éléments d'architecture (composants, services, ...) repose sur l'échange avec leur environnement, ou de manière plus générale que le système soit collaboratif,
- Observation, désignation : les éléments de l'architecture doivent être observables et désignables : c'est à dire que chaque élément de l'architecture intervenant dans un contrat, doit pouvoir être observé, pendant les moments du cycle de vie auquel on souhaitera appliquer le contrat, au moins pour les grandeurs contraintes par le contrat. L'architecture devra donc proposer une réflexivité au moins structurelle, si ce n'est comportementale. Pour ce faire chaque élément de l'architecture doit pouvoir être désigné par une expression dont la résolution par un mécanisme adhoc retourne une référence sur l'entité logicielle correspondante dans le système contraint,
- Relations : les relations entre éléments d'architecture doivent être observables même si elles ne sont pas réifiées, afin de pouvoir découvrir la réalisation d'un motif d'architecture,

Pour s'assurer que notre modèle de contrat peut s'appliquer à différentes architectures, et bien que cela ne remplace bien sur pas la mise en oeuvre du contrat sur des systèmes concrets variés, nous allons ici dans un premier temps évaluer la compatibilité conceptuelle des architectures et de nos hypothèses. Nous citerons au passage quelques exemples de plateformes qui se conforment à nos hypothèses.

3.3.2.1 Objets

Conceptuellement le fonctionnement d'une application composée d'objets repose sur leur collaboration. Les librairies de programmation par aspect, comme les capacités de réflexion des langages à objets, ou l'instrumentation de machines virtuelles, permettent aux architectures objets de satisfaire aux hypothèses conditionnant la mise en oeuvre de notre modèle :

- désignation : les librairies d'aspect, ou la réflexivité structurelle du langage, permettent de désigner des objets à des niveaux d'abstractions plus ou moins élevés (pointcut de aspectJ par exemple),
- observation :
  - configuration, relation : pourvu que le langage du programme instrumenté soit réflexif il est possible d'explorer le graphe d'objets et leurs relations,

  - exécution : les librairies d'aspect, ou la réflexivité comportementale permettent d'intercepter l'exécution sur les appels de méthodes entrants et sortants des objets et de récupérer les paramètres passés, ainsi que les références à l'objet appelant et appelé. Il est ainsi possible d'observer l'exécution des objets,

Les objets répondent à la plupart des hypothèses posées pour l'application du modèle de contrat. Toutefois il leur manque la capacité de désigner une instance d'objet dans le graphe d'objets (à l'aide par exemple d'un chemin). Néanmoins pour les langages réflexifs comme java ou SmallTalk, un tel outil ne semble pas insurmontable à développer. Ainsi le modèle de contrat pourrait être appliqué à des architectures d'objets.

3.3.2.2 Composants

Tout comme pour les objets, le fonctionnement d'une application formée de composants repose sur leur collaboration le plus souvent explicite. Conceptuellement la désignation de composants dans un système est favorisée par une architecture classiquement explicitée à l'aide d'ADL (Architecture Description Language). Enfin l'observation des interactions est facilitée par le fait que les composants exposent, en général, ce qu'ils s'échangent. Différents exemples de modèles de composants confirment que les hypothèses nécessaires au modèle de contrat se rencontrent dans le cadre des plateformes de composants :

- désignation : un exemple de désignation est fourni par le langage FPath ([26]) développé dans le cadre du modèle de composant Fractal. Ce langage, inspiré de XPath, permet d'interpréter des chemins pour désigner des composants au sein d'une architecture donnée.

- observation :

  - configuration, relation : certains modèles de composants comme Fractal sont réflexifs et permettent à partir d'une instance de composant de naviguer de proche en proche dans celles qui l'entourent,
  - exécution : plusieurs modèles de composants, comme Fractal et CCM, permettent de poser des intercepteurs sur les interfaces exposés par les composants, cela permet de suivre l'exécution du système.

L'approche par composant est ainsi conceptuellement compatibles avec nos hypothèse. D'un point de vue plus pratique un modèle de composant tel que Fractal répond à toutes les demandes du modèle de contrat. D'autres modèles de composants fournissent certaines hypothèses et pourraient être étendus pour satisfaire à toutes.

3.3.2.3 Services

A la différence des objets et des composants, les services se concentrent plus sur l'interaction avec leur client direct. En effet comme les services privilégient un fort découplage entre eux, ils ne désignent pas ceux avec lesquels ils pourraient interagir, et cette interaction ne fait donc pas partie de leur principe de fonctionnement. Toutefois, comme nous avons vu dans l'état de l'art il existe différents langages d'organisation de l'activité d'un ensemble de services destiner à les faire collaborer (BPEL, WSCI, BPML

...). Par contre l'observation du fonctionnement des services est plus développée que dans le cadre des objets et des composants, qu'il s'agisse de grandeurs fonctionnelles ou non fonctionnelles.

- désignation : les langages décrivant des collaborations de services explicitent leur organisation. Ainsi même si nous ne connaissons pas de langage permettant de désigner un service au sein d'une telle organisation, une implémentation réflexive de l'organisation (comme celle d'ActiveBPEL[1]) permettrait d'en produire un.

- observation :

  – configuration, relation : certains outils de mise en oeuvre de collaboration de services offrent des outils d'exploration de la configuration de services qu'ils produisent. Par exemple ActiveBPEL propose une interface d'interaction et d'exploration des orchestrations de services qu'elle permet d'instancier,

  – exécution : de nombreuses librairies existent de monitoring du fonctionnement des services,

Conceptuellement l'approche par services et orchestration de services semble donc compatibles avec les hypothèses de mise en oeuvre de notre modèle de contrat. D'un point de vue plus pratique le fait que différents outils organisent les services dans le cadre d'orchestrations explicites et que certains fournissent un accès réflexif à celles-ci laisse supposer que la désignation des services dans le cadre d'orchestration est possible. Par ailleurs, l'observation, tant de la configuration (au sein d'une organisation) que de l'exécution des services, est aussi possible.

3.3.3 Formalismes acceptés par le modèle de contrat

Notre modèle de contrat n'est pas dédié à un formalisme spécifique comme CCLJ ou les Behavior Protocols. Néanmoins, il fait certaines hypothèses sur les formalismes des spécifications qu'il est susceptible de mettre en oeuvre. Pour que les spécifications soient organisées par notre modèle de contrat il faut que leur formalisme :

- soit modulaire, c'est à dire que chaque spécification doit porter sur une seule entité logicielle (à la concrétisation par hypothèse indépendante des autres),

- permette d'extraire mécaniquement, de chaque spécification, les observations (leurs descriptions) qu'elle contraint sur l'entité logicielle à laquelle elle est attachée,

- permette d'interpréter mécaniquement chaque spécification en un couple de contraintes, sur son entité logicielle associée, répondant à la sémantique de l'approche hypothèse-garantie,

- dispose d'outils pour évaluer la conformité d'une spécification à l'entité logicielle à laquelle on l'associe, et pour évaluer la compatibilité de deux ou plus spécifications,

[1] http ://www.active-endpoints.com/index.htm

Tout comme pour l'applicabilité de notre modèle de contrat à différents types d'architectures, nous devrons implémenter l'intégration de différents formalismes pour valider qu'il en accepte bien de variés. Cependant, il nous est ici possible d'envisager d'un point de vue conceptuel les formalismes qui répondent à nos critères et d'extraire de notre état de l'art quelques exemples qui répondent ou non à nos hypothèses.

3.3.3.1 Modularité

Nous pouvons constater que le génie logiciel tend au découplage, et à la modularisation, des entités logicielles mises en oeuvre dans les applications. En particulier, les objets, composants et services expriment dans leur définition cette évolution. Il n'est donc pas surprenant que même 'il n 'est pas exhaustif, notre état de l'art des formalismes de spécification des objets, composants et services n'ait rencontré que des formalismes modulaires. Nous en concluons que la majorité des formalismes de spécification des objets, composants et services sont modulaires.

3.3.3.2 Extraction des observations

Si nous considérons une spécification, nous pouvons constater que plus son formalisme est de haut niveau, c'est à dire éloigné de la réalisation concrète de l'application qu'elle contraint, plus il est délicat d'extraire de son expression les observations sur lesquelles elle s'appuie. Ainsi il est aisé d'extraire les observations contraintes par un formalisme assertionnel comme nous l'avons réalisé dans le cas du système de contrat Confract avec l'interprétation du langage d'assertion adapté aux composants nommé CCLJ. Par contre comme nous l'avons vu dans le cas de la logique temporelle TLA [22], les observations contraintes par une spécification doivent être fournies par le spécificateur au moment de l'application de la spécification à un système concret. Il existe bien sûr une gradation dans la facilité ou difficulté à extraire les observations contraintes par une spécification et il n'y a pas de frontière nette entre ceux pour lesquels cela est possible mécaniquement et les autres. Toutefois plus un formalisme autorise la désignation d'élément architecturaux d'implémentation (composants, interfaces, méthodes, etc...) plus facile en sera l'extraction automatique des observations de son expression.

3.3.3.3 Interprétation sous forme d'hypothèse et garantie

Notre approche suppose de pouvoir interpréter une spécification sous forme d'une hypothèse et ou d'une garantie (certaines pouvant être de pures hypothèses, d'autres de pures garanties), contraintes équivalentes à la spécification. A nouveau les observations contraintes par la spécification jouent un rôle déterminant dans son interprétation. Ainsi une contrainte sur une observation du porteur de la spécification est une garantie que le porteur doit satisfaire, une contrainte sur une observation de l'environnement du porteur est une hypothèse dont le porteur attend la satisfaction. Deux grands cas de figures se présentent alors :

Hypothèse et garantie sous forme d'expressions distinctes

L'expression de la spécification peut être divisée en deux expressions contraignant chacune un ensemble distinct d'observations, d'un côté les observations sur le porteur, d'un autre les observations sur son environnement. La sémantique de ces deux expressions doit être telle que la satisfaction de la spécification originale soit équivalente à la

satisfaction de la garantie tant que l'hypothèse l'est. Par exemple si nous considérons sur notre exemple la spécification suivante :

```
On <Engine>
  context float csp.getRTime (float throttle)
    pre : throttle < 10
    post : retour < 5
```

Elle spécifie que pour autant que l'accélération demandée au moteur est inférieure à 10, son temps de réponse sera inférieur à 5. Elle peut se diviser en deux expressions respectant l'approche hypothèse-garantie :

- une hypothèse : `throttle < 10` qui porte sur la valeur de `throttle` fournie par son environnement à `<Engine>`,
- une garantie : `retour < 5` qui porte sur la valeur de retour de la méthode `getRTime` fournie par le composant à son environnement,

telles que la garantie doit être satisfaite tant que l'hypothèse l'est.

Ces conditions peuvent paraître contraignantes, néanmoins de nombreux formalismes de description d'éléments d'architecture reposent sur, ou peuvent se ramener à cette approche hypothèse-garantie. En effet celle-ci met en oeuvre des contraintes interdépendantes entre un élément et son environnement. Nous avons vu que nous retenions les architectures dont le fonctionnement reposait sur la collaboration et donc les échanges entre leurs participants. Or dans ce cadre les éléments d'architecture sont définis par ce qu'ils fournissent et requièrent vis à vis de leur environnement. Ainsi la distinction entre l'élément d'architecture et son environnement, ainsi que la potentielle dépendance entre ce que l'élément fournit et requiert, font que l'approche hypothèse-garantie est particulièrement adaptée à la description des propriétés des éléments d'architectures collaboratives.

Hypothèse et garantie sous forme d'une seule expression

L'expression de la spécification ne peut être divisée en deux contraintes bien qu'il soit possible de distinguer les observations qui contraignent le porteur de la spécification de celles sur son environnement. Il se peut dans ce cas que l'évaluation de la contrainte ne fasse pas intervenir toutes les observations en même temps, ce qui nous permet alors de savoir, en fonction de l'observation contrainte à un instant donné, si la spécification doit être interprétée comme une hypothèse ou une garantie. Par exemple si nous considérons la spécification suivante en Behavior Protocol du `<CruiseCtrl>` :

```
        (?sns.on; !csp.setThrottle*; ?sns.off)*
```

pour garantir sa non-contradiction par rapport à l'exécution concrète du système, cette contrainte est évaluée pour un appel à la fois[2]. Ainsi cette expression est une garantie de `<CruiseCtrl>` par rapport à l'appel de `setThrottle` car il doit l'émettre, par contre c'est une hypothèse par rapport aux appels de `on` et `off` car il attend ces derniers de son

[2]nous faisons l'hypothèse que notre système est mono thread

environnement. La sémantique "hypothèse-garantie" de cette expression est que tant que l'environnement appelle `on` et `off` aux bons moments, le `<CruiseCtrl>` garantit qu'il émet des appels à `setThrottle` aux bons moments. Concrètement cela se traduit par l'affectation de la responsabilité en cas de contradiction entre la spécification et l'éxécution du système. En effet suivant que l'appel est entrant ou sortant, c'est respectivement l'environnement ou le composant qui est responsable. Nous pouvons donc constater qu'une hypothèse ou une garantie ne sont pas seulement de simples expressions mais l'association d'une contrainte avec les grandeurs sur lesquelles elle est évaluée.

3.3.3.4 Outils associés

Les outils associés aux différents formalismes sont variés. Certains permettent de vérifier la conformité des spécifications à leur porteur comme souvent dans le cas des assertions, dans ce dernier cas ils peuvent aussi être utilisés pour s'assurer de leur compatibilité. D'autres se focalisent plus exclusivement sur l'évaluation de la compatibilité de différentes spécifications comme dans le cas de formalismes logiques. Enfin certains formalismes comportementaux (Sofa) et de qualité de services (CQML) proposent à la fois l'évaluation de la conformité et de la compatibilité des spécifications.

3.3.4 Synthèse

Cette analyse nous a conduit a concevoir un framework de contrat en trois parties, sur la base des propriétés qu'il fournit et qu'il requiert. La première partie, constitue le noyau du framework et contient les éléments de modélisation des contrats. Ces éléments permettent à nos contrats de présenter sous forme réifiée les propriétés de conformité, compatibilité et responsabilité. Par ailleurs nos contrats sont indépendants de la nature concrète de l'architecture contrainte et des formalismes des spécifications qu'ils organisent. Nous adjoignons donc à ce noyau deux autres parties qui lui fournissent l'accès à une architecture et aux formalismes de spécification concrets qu'il requiert. Il s'agit d'une part d'un adaptateur d'architecture qui fournit au noyau les observations du système concret. Ces observations serviront à construire les contrats et évaluer la conformité, compatibilité et responsabilité des participants. Et d'autre part des adaptateurs de formalisme traduisent les spécifications en hypothèse-garantie, pour exprimer les propriétés de conformité compatibilité et responsabilité, tout en fournissant leurs outils d'évaluation. Ces deux parties modélisent architecture et formalismes dans les termes génériques définis par le noyau. La figure 3.4 illustre cette organisation.
De plus, il faut noter que cette approche permet à chacun des intervenants dans la mise en oeuvre des contrats sur un système donné de ne se préoccuper que de la partie de notre framework le concernant. Nous avons ainsi distingué cinq intervenants :

l'administrateur du système contractualisé : il ne se préoccupe que des messages émis par le noyau du modèle. Ils lui permettent sans être un expert des formalismes ou de l'implémentation de l'architecture de distinguer les erreurs de conformité, de compatibilité et d'obtenir un diagnostic désignant l'entité à l'origine de l'erreur. C'est-à-dire qu'il dispose de premiers éléments pour cerner le problème pour faire éventuellement appel à l'expert concerné (d'un formalisme, de la configuration architecturale, du type d'élément architectural...),

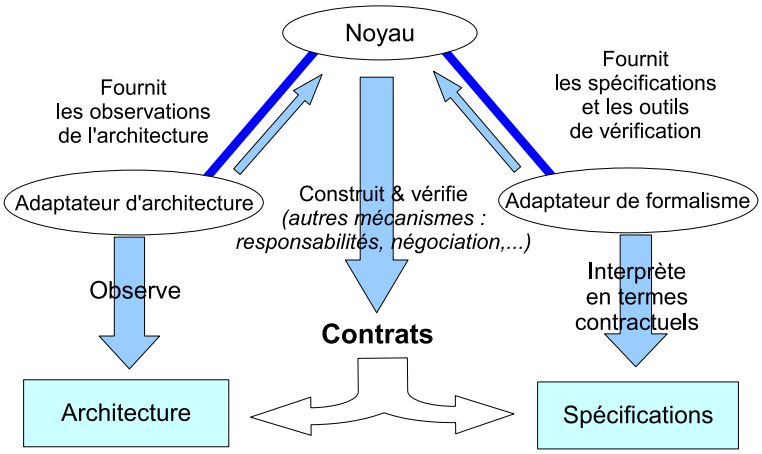


FIGURE 3.4 – Vue d'ensemble du système de contrat

l'intégrateur d'architecture : il produit l'adaptateur d'architecture dédié à un type d'architecture donnée (par exemple Fractal, CCM etc...). Il ne se préoccupe que des interfaces de description générique d'architecture fournies par le noyau. Il n'a pas à connaître les adaptateurs de formalismes qui sont susceptibles d'être mis en oeuvre, ni l'implémentation des outils associés à ces formalismes et celle du noyau,

l'intégrateur de formalisme : il produit l'adaptateur de formalisme pour un formalisme donné dans les termes génériques définis par le noyau. Il utilise les observations que l'adaptateur d'architecture met à sa disposition mais il n'a pas besoin de connaître l'implémentation de celui-ci, ni celle de l'architecture contrainte et du noyau,

le fournisseur de spécifications : il peut se concentrer sur la rédaction des spécifications sans avoir à connaître précisément l'implémentation du système auquel elles sont destinées et celle des adaptateurs (pourvu que leurs hypothèses de fonctionnement aient été clairement fournies),

l'assembleur de l'architecture : il peut aussi se concentrer sur sa tâche sans avoir besoin de connaître les adaptateurs de formalismes et d'architecture. Dans certains cas il est possible qu'il lui faille utiliser des éléments d'architecture dont l'adaptateur d'architecture a modifié l'utilisation mais alors il suit les consignes de l'intégrateur d'architecture sans rentrer dans le détail de l'adaptateur. Par exemple, si l'observation de l'architecture impose l'introduction de nouveaux éléments solidaires de ceux existants, comme des proxys ou des intercepteurs, l'assembleur peut être amené à les manipuler mais ne devra pas avoir à connaître leurs spécificités et implémentations,

Une même personne peut avoir la charge de toutes ces tâches mais notre approche lui permet de les cloisonner. Cela a l'avantage d'autoriser des tests et des certifications des différents éléments nécessaires à la mise en oeuvre des contrats, de manières distinctes.

Ce découpage des tâches leur permet aussi d'être traitées chacune par un expert de leur domaine et surtout de faire partager cette expertise aux autres intervenants. Par exemple, l'intégration du modèle de composants Fractal va permettre à des non spécialistes de ce système de lui appliquer, par exemple, leur formalisme. Ainsi notre approche a l'avantage de permettre des mises en oeuvre d'architecture, de formalismes et de spécifications qui n'auraient pas eu lieu du fait de la formation qu'aurait du entreprendre le spécialiste d'un domaine pour prendre en charge les tâches autres que celles de son domaine de compétence.

3.4 Conception du modèle générique de contrat

D'après les propriétés que nous avons décrites et que nous souhaitons réifier dans le contrat, nous allons faire reposer celui-ci sur quelques concepts principaux. Il faut noter que nous souhaitons que le contrat soit une entité de premier ordre, dont l'état reflète la validité de l'architecture.

3.4.1 Les constituants du contrat

3.4.1.1 Les participants

Les participants du contrat sont les éléments d'architecture dont nous souhaitons contraindre la collaboration afin de valider l'organisation architecturale. Dans un système à composants, il s'agit des composants (comme montré sur la figure 3.5), dans une orchestration de services, ce sont les services.

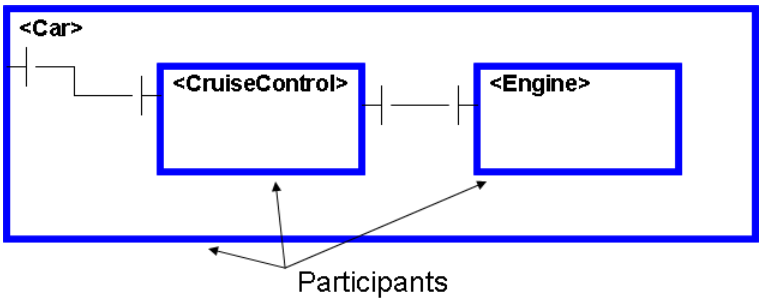


FIGURE 3.5 – Participants du contrat

3.4.1.2 Les spécifications contractuelles

Une spécification contractuelle est la spécification d'un élément de l'architecture, interprétée en termes d'hypothèse et garantie. L'hypothèse et la garantie sont définies chacunes par une contrainte et les grandeurs sur lesquelles elle est évaluée. Ainsi une contrainte évaluée pour des grandeurs émises ou exposées par le porteur de la spécification est une garantie fournie par ce dernier. Une contrainte évaluée pour des grandeurs reçues ou requises par le porteur est une hypothèse du porteur sur son environnement. Sa sémantique est que tant que l'hypothèse est satisfaite par l'environnement de l'élément, alors celui s'engage à satisfaire à la garantie dont il est responsable (figure 3.6). **Il faut noter que l'hypothèse et la garantie liées par la spécification ne portent pas**

nécessairement sur la même interface. Elles peuvent chacune porter sur une, voire plusieurs interfaces différentes.

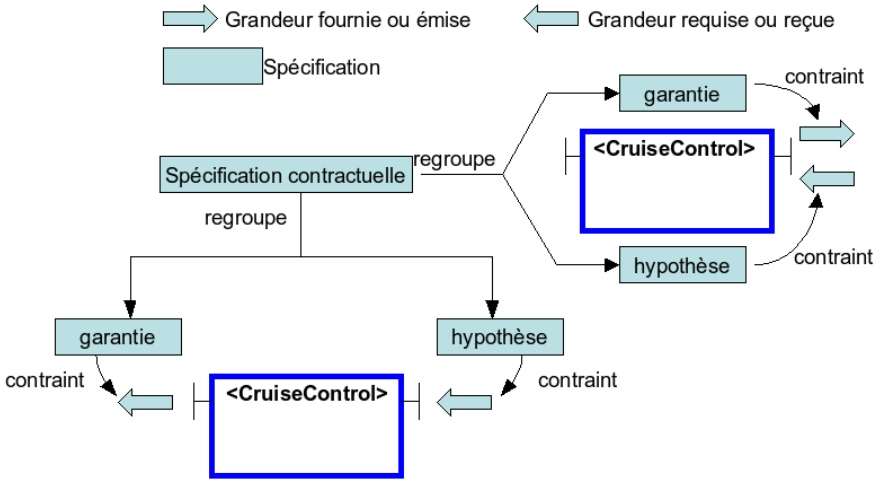


FIGURE 3.6 – Spécification contractuelle

Dans le cadre de notre exemple nous avons vu que la spécification du `<CruiseCtrl>` :

```
(a)        (?sns.on; !csp.setThrottle*; ?sns.off)*
```

pouvait être interprétée de la manière suivante :

- hypothèse = expression (a) évaluée pour les appels reçus à `on` et `off`
- garantie = expression (a) évaluée pour les appels émis à `setThrottle`

3.4.1.3 Les clauses

Les clauses réifient la conformité des participants à leurs spécifications en associant, chacune, un participant à une de ses spécifications contractuelles, comme illustré sur la figure 3.7. Chaque clause fait du participant, sur lequel elle porte, le responsable de la spécification contractuelle qu'elle lui associe. Les clauses sont des objets évaluables qui peuvent être satisfaits ou non suivant la conformité du participant à sa spécification. Ainsi chaque clause exprime l'obligation (et l'attente) de chaque participant vis à vis des autres et vérifie qu'il s'y conforme bien du moment que son environnement lui fournit ce qu'il requiert.
Par exemple une clause sera formée de l'association d'une instance du composant `<CruiseCtrl>` à sa spécification contractuelle vue dans le paragraphe précédent. De cette manière il sera possible d'évaluer la conformité de cette instance de composant à sa spécification, en observant chacun des appels entrants ou sortants.

3.4.1.4 L'accord

L'accord réifie que chacune des attentes des participants du contrat est satisfaite par les obligations des autres. Comme ces attentes et obligations sont exprimées par les clauses du contrat, l'accord représente la compatibilité de ces dernières. Son expression est celle de la compatibilité des spécifications contenues dans les clauses. Ce sont les

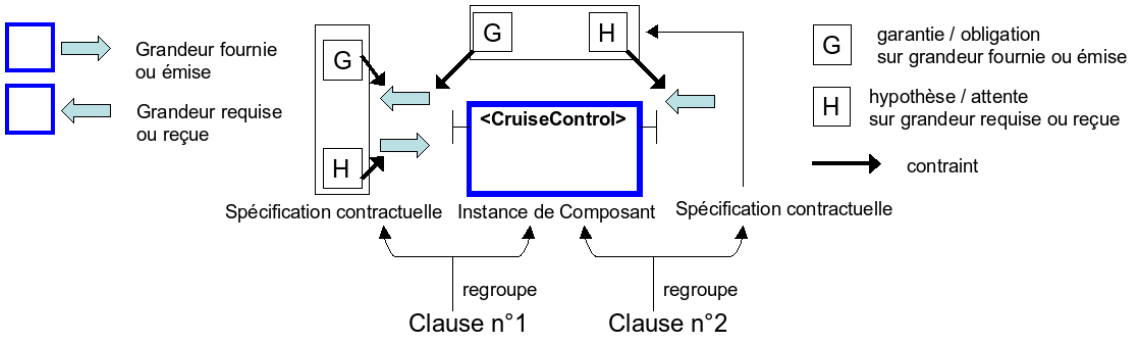


FIGURE 3.7 – Clauses

relations qui connectent les participants au contrat qui déterminent les clauses dont la compatibilité est à étudier. En effet, comme nous avons vu, ces relations définissent les grandeurs que les participants vont s'échanger et qu'ils vont de ce fait avoir en commun. Ainsi les clauses des différents participants contraignant une même grandeur (celle qu'ils s'échangent) doivent donc être compatibles (cf figure 3.8). L'accord rassemble l'ensemble de ces expressions de compatibilité (variables suivant le formalisme). L'accord est évaluable, sa valeur reflète la compatibilité des spécifications. Nous avons vu que celle-ci ne valait que dans la mesure ou était exprimé sa portée tant du point de vue "géographique", c'est à dire des relations qu'elle contraignait, que du point de vue "logique", des spécifications qu'elle prenait en compte. Ces hypothèses seront associées au "type du contrat" présenté plus bas.

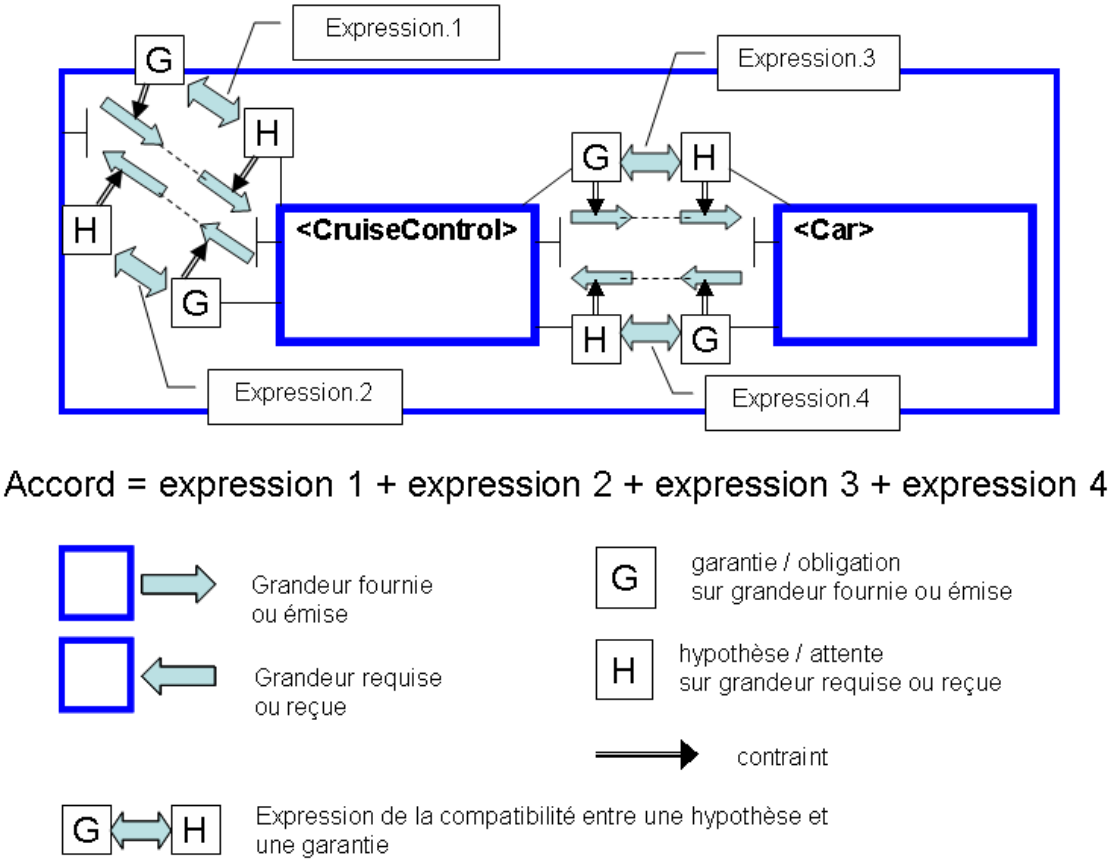


FIGURE 3.8 – Accord

Deux expressions simples de compatibilité ont été données dans lors de l'analyse de la compatibilité, nous en rappellerons juste celle de `<CruiseCtrl>` et `<Engine>` :

$$((?csp.setThrottle; !fdk.notifyAction) * \sqcap (?sns.on; !csp.setThrottle*; ?sns.off)*)$$

interprétée sous la forme évaluable suivante :

```
parallelCheck ((?csp.setThrottle; !fdk.notifyAction)*,  // <Engine>
     (?sns.on; !csp.setThrottle*; ?sns.off)*) // <CruiseCtrl>
```

Cette formule exprime que le moment auquel le message `csp.setThrottle` est émis (par `<CruiseCtrl>`) doit correspondre au moment auquel il est attendu (par `<Engine>`). Dans notre exemple `parallelCheck` effectue une vérification par modelChecking qui est dans ce cas exhaustive. Mais l'accord aurait tout aussi bien pu être évalué par l'observation des messages `csp.setThrottle` échangés et la vérification que les deux spécifications sont satisfaites (ou plus précisément que la garantie implique l'hypothèse).

3.4.1.5 Exemple de contrat

Un exemple de contrat très simple mettant en oeuvre les éléments présentés est le suivant (3.9). Il est passé entre `<Engine>` et `<CruiseCtrl>` et son accord et ses clauses utilisent les expressions dont nous avons discuté précédemment.

```
Contract
{
  Participants : <Engine>, <CruiseCtrl> ;

  Clause :
   responsible : <Engine>;
   assumption : check((?csp.setThrottle; !fdk.notifyAction)*, {
       setThrottle})
   guarantee : check((?csp.setThrottle; !fdk.notifyAction)*, {
       notifyAction})

  Clause :
   responsible : <CruiseCtrl>
   assumption : check((?sns.on; !csp.setThrottle*; ?sns.off)*, {on,off})
   guarantee : check((?sns.on; !csp.setThrottle*; ?sns.off)*, {
       setThrottle})

  Agreement :
    parallelCheck ((?csp.setThrottle; !fdk.notifyAction)*,
    (?sns.on; !csp.setThrottle*; ?sns.off)*)
}
```

FIGURE 3.9 – Exemple de contrat

L'expression `check(protocole, messages)` dénote la vérification que les messages en question sont détectés à des instants compatibles avec le protocole. Nous pouvons par ailleurs remarquer que conformément à notre approche le message `setThrottle` apparait dans l'hypothèse d'`<Engine>` qui le reçoit, et dans la garantie de `<CruiseCtrl>` qui l'émet.

3.4.2 Vue d'ensemble

Une vue d'ensemble du modèle de contrat est donnée dans la figure 3.10. Chaque contrat est associé à la réalisation d'un motif d'architecture (ArchitecturalPatternInstance) qui référence un ensemble d'entités architecturales (composants ou services), les *Participants* du contrat. Un exemple très simple de motif d'architecture est celui du client-serveur décrivant que les participants du contrat sont un composant client et un serveur, liés par la connexion d'une interface requise exposée par le client à une autre fournie exposée par le serveur.

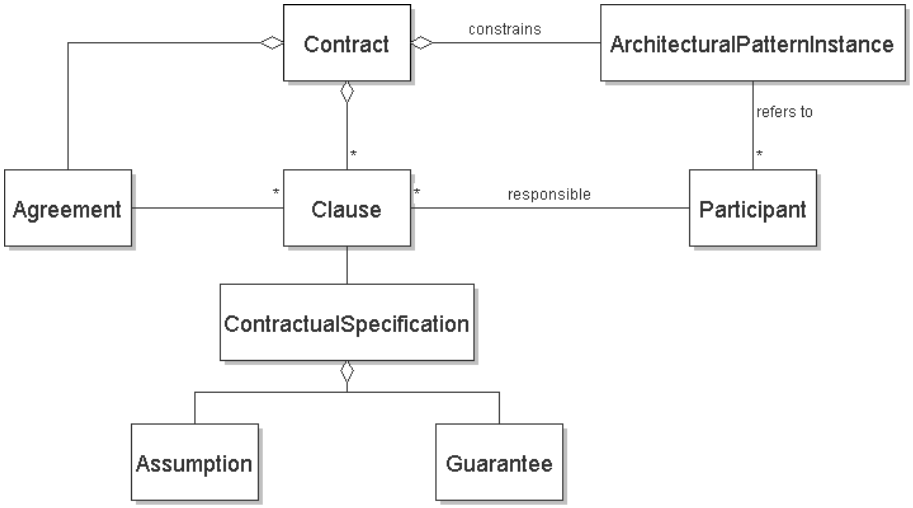


FIGURE 3.10 – Diagramme de classes du contrat

Un exemple d'instanciation du modèle de contrat est illustré par la figure 3.11. Il s'agit d'un simple contrat "client-serveur" passé entre le `<CruiseCtrl>` et l'`<Engine>`. Il est obtenu sur la base des spécifications S3 et S4 associées au système exemple, et a ainsi pour objet d'assurer que l'échange de la valeur de retour de la méthode `getRTime` est correct.

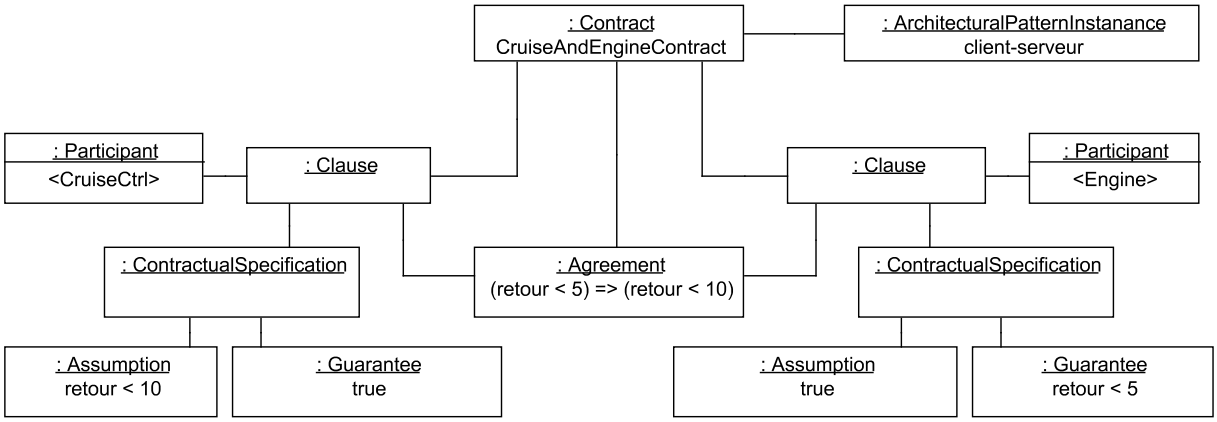


FIGURE 3.11 – Instance de contrat

La *Guarantee* et l'*Assumption* modélisent la garantie et l'hypothèse de la spécification contractuelle. Ils contiennent chacun un prédicat et la description des observations

qu'il contraint. Pour un composant donné, la garantie contraint ce que le composant fournit (messages émis, signaux, etc...), l'hypothèse contraint ce qu'il requiert (messages et signaux reçus etc...). Suivant le formalisme des prédicats, leur satisfaction est évaluée à la configuration ou à l'exécution du système.
Une *ContractualSpecification* (modélisation de la spécification contractuelle) est un prédicat qui associe une *Assumption* et une *Guarantee* pour un composant donné. En tant que spécification contractuelle elle suit le principe de l'approche hypothèse-garantie : la garantie doit être vraie aussi longtemps que l'hypothèse l'est. Ainsi comme nous l'avons vu dans l'étude de la portée du modèle de contrat, les formalismes des spécifications doivent être modulaires et tels qu'ils puissent être interprétés en terme d'hypothèses et garanties. Une *Clause* (modélisation de la clause) est un objet associant une spécification contractuelle avec un participant du contrat, qui est alors responsable de sa satisfaction. L'*Agreement*, modélisation de l'accord, exprime la compatibilité des clauses du contrat. Plus précisément il exprime que les hypothèses des parties collaborantes sont remplies par les garanties qu'elles se fournissent mutuellement.
Ainsi ce modèle de contrat est indépendant du formalisme des spécifications, pourvu qu'elles puissent s'interpréter en termes d'hypothèses et garanties (propriété n°4). Il réifie via les *Clause* la conformité des participants à leurs spécifications (propriété n°1), et via l'*Agreement* la compatibilité de celles-ci (propriété n°2). Enfin les responsabilités sont explicitées par l'association d'un participant à chaque clause (propriété n°3).

3.4.3 Le motif architectural (`ArchitecturalPatternInstance`)

Tout comme les spécifications, la validité d'une architecture gagne à être exprimée et établie de manière modulaire. Il est intéressant de considérer l'architecture non pas prise dans son ensemble (et pour toutes ses spécifications), mais via des sous ensembles d'éléments interconnectés. Limiter la portée architecturale des contrats (tant en étendue que dans la nature des relations contraintes) permet :

- en cas d'une modification locale de l'architecture, de ne re-valider qu'un contrat local, et non pas un contrat global remettant en cause tout le système,
- d'évaluer la validité de sous ensembles pertinents et lisibles de l'architecture, comme par exemple des design patterns ou des styles architecturaux,
- de limiter la portion d'architecture remise en cause par la violation du contrat, et donc potentiellement de laisser le reste de l'architecture s'exécuter,

Un cas de figure typique est celui des systèmes à composants hiérarchiques : certaines parties d'une architecture ne sont pas toujours visibles depuis d'autres, et on peut donc déjà distinguer les contrats qui vont s'appliquer entre les composants qui se voient mutuellement car ils sont indépendants de ceux qu'ils ne voient pas.

Ainsi dans le cadre d'une architecture, il faut pouvoir distinguer les éléments d'architecture qui vont être des participants du contrat. Il faut aussi distingeur les relations que ce dernier va contraindre car elles vont guider la production des clauses et de l'accord, en permettant d'identifier les échanges à considérer. Ceci est l'objet du motif d'architecture qui définit de manière générique un ensemble d'éléments d'architectures et de relations les connectant à l'aide d'un ensemble de conditions portant sur

les premiers et les seconds. Chaque ensemble d'éléments et relations d'un système satisfaisant à un motif d'architecture est nommé "instance" du motif d'architecture. Un exemple de motif d'architecture est donné sur la figure 3.12.

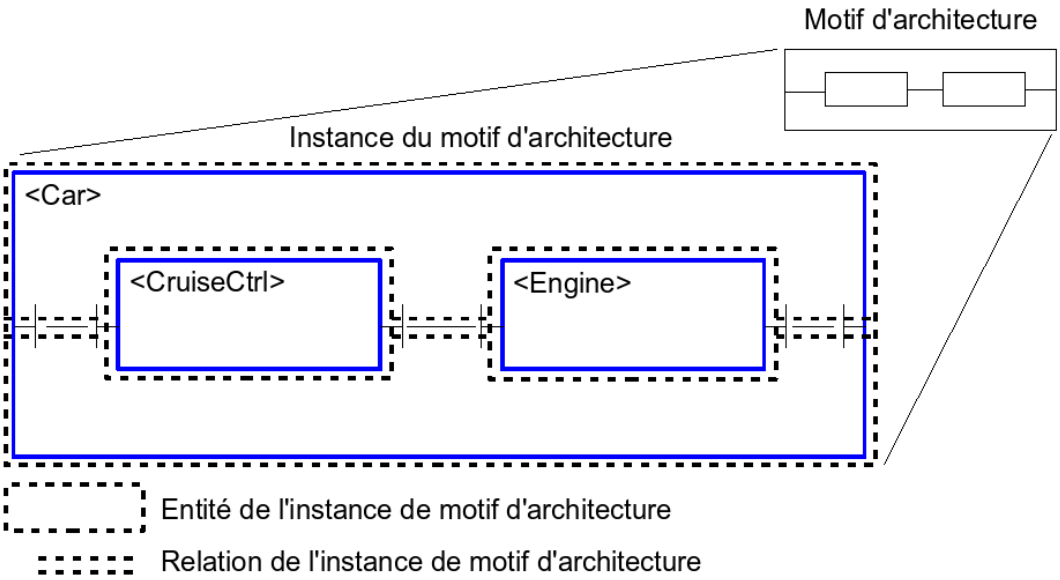


FIGURE 3.12 – Un motif d'architecture et son instance

Un exemple de motif d'architecture basique (ne contenant qu'une seule relation) est celui du client serveur pour les composants :

- relation : une relation de connexion d'interface entre deux composants,
- composants : les composants client et serveur liés par cette relation,

Un autre exemple de motif basique est celui du design pattern "sujet - observateur", qui désigne seulement la connexion d'une unique interface à plusieurs autres. Dans le cadre de notre exemple, d'autres éléments que le moteur pourraient être intéressés par les décisions du `<CruiseCtrl>`. Ce dernier leur donne donc la possibilité de s'enregistrer comme "écouteurs" de ses actions dont il les notifie alors, comme illustré sur la figure 3.13 qui montre une instance du motif "sujet observateur".
Les interfaces serveur `SubjectObserver` des écouteurs viennent se connecter à l'unique interface cliente `Observer` du `CruiseCtrl` pour former une instance de ce motif d'architecture. Il est défini de manière générique par un ensemble de relations de connexion liant une unique interface cliente, `Observer`, à un ensemble d'interfaces serveur `Subject-`
`Observer`. L'application d'un contrat à ce motif permettra d'exprimer et vérifier des propriétés d'interaction qui lui sont propres comme par exemple que chaque notification est effectuée une seule fois, que chaque message est bien reçu, etc...

3.4.3.1 Principe

Un motif d'architecture (ArchitecturalPattern) est la description générique, c'est à dire indépendante d'une instance architecturale particulière, des relations sur lesquelles porte le contrat. La description de ce motif repose sur trois types de propriétés :

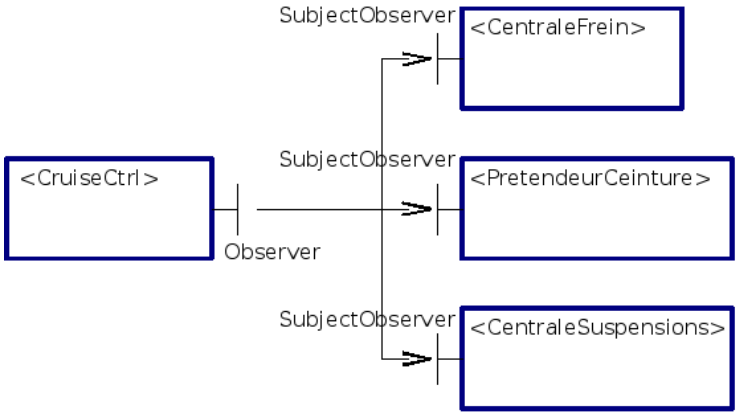


FIGURE 3.13 – instance du motif d'architecture sujet observateur

- sa géométrie : composée d'un ensemble de relations, dont on peut décrire l'organisation de manière simple à l'aide d'expressions du type : début(relation1)=fin(relation2),
- les propriétés des relations qu'il accepte. Par exemple, on peut avoir pour "relation1" une relation d'inclusion, pour "relation2" une relation de connexion,
- les propriétés des entités qu'il accepte aux extrêmités des relations, par exemple : debut(relation1)=composant composite,

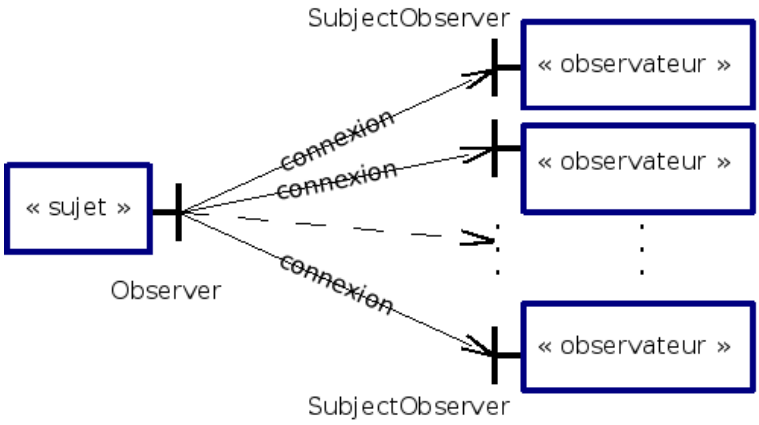


FIGURE 3.14 – illustration du motif d'architecture sujet observateur

Par exemple le motif architectural "sujet-observateur" dont nous avons donné un exemple d'instance précédemment pourra se définir de la manière suivante (illustrée par la figure 3.14) :

- géométrie : début($relation_i$)=début($relation_j$) quelques soient i et j,
- propriété des relations : $relation_i$ sont des relations de connexion d'interfaces,
- extrêmités des relations : quelque soit i : debut($relation_i$)=interface client `Observer`, fin($relation_i$)=interface serveur `SujectObserver`,

Au sein du motif d'architecture les positions peuvent occuper les éléments d'architecture, extrêmités des relations, peuvent être associées à des noms. Ces noms définissent des rôles au sein du motif d'architecture. Il faut noter que dans le cadre d'un motif d'artchitecture donné : un rôle peut s'appliquer à plusieurs positions mais qu'une position ne possède qu'un seul rôle. Par exemple dans le cas d'un contrat d'interface, une entité à l'extrêmité de la relation de connexion sera à la place du "serveur", l'entité à l'autre extrêmité sera à la place du "client". Dans le cas du motif sujet observateur, le porteur de l'interface `Observer` (interface offerte aux observateurs) sera le "sujet", les autres composants sont les "observateur"s.

Par ailleurs le motif d'architecture ne traverse jamais une frontière de visibilité dans l'architecture à laquelle il s'applique. Nous nommons porteur du contrat, le plus petit élément d'architecture englobant les relations sur lesquelles porte le contrat. Nous disons alors que le contrat s'applique dans la portée de cet élément. Ainsi par exemple dans le cas d'un système d'éléments d'architecture hiérarchiques, le porteur d'un contrat est toujours un composite (éventuellement la racine de l'architecture). En effet le plus petit englobant de toute relation dans ce type d'architecture, est le composant contenant les composants qui sont reliés.

3.4.3.2 Fonctionnement

Une instance de motif d'architecture, ou ArchitecturalPatternInstance, désigne un ensemble de relations architecturales entre des entités qui occupent des rôles. Pour savoir quelle relation peut entrer dans une instance de motif d'architecture, l'ArchitecturalPattern fonctionne de la manière suivante :

- pour chaque relation architecturale qu'on lui soumet, définie par deux éléments d'architecture et la description de leur lien, il retourne si elle peut faire partie d'une de ses instances,
- si une relation soumise peut faire partie d'une instance, l'ArchitecturalPattern retourne en plus le(s) couple(s) de rôles que peuvent occuper ses extrêmités,

Une fois que le couple de rôles acceptable pour une relation est connu, il faut soumettre cette relation, pour ces rôles, aux instances existantes du motif d'architecture qui ne sont pas complètes. Si une instance accepte la relation, alors elle lui est ajoutée et le processus se termine pour cette relation. Si aucune instance de motif d'architecture n'accepte, pour ces rôles, la relation alors il faut en créer une nouvelle instance qui contiendra la relation, dont les extrêmités auront les rôles déterminés par l'ArchitecturalPattern. La figure 3.15 illustre ce processus.

Les règles concernant le placement d'une relation architecturale dans une instance de motif d'architecture sont simples, et évitent la problématique complexe du "*pattern matching*" :

- 1) une entité, extrêmité d'une relation architecturale, ne peut occuper qu'une seule position au sein d'un ArchitecturalPatternInstance, par extension une entité ne peut avoir qu'un seul rôle dans une instance donnée de motif d'architecture,
- 2) deux entités différentes, extrêmités de deux relations architecturales, ne peuvent occuper la même position au sein d'un ArchitecturalPatternInstance,

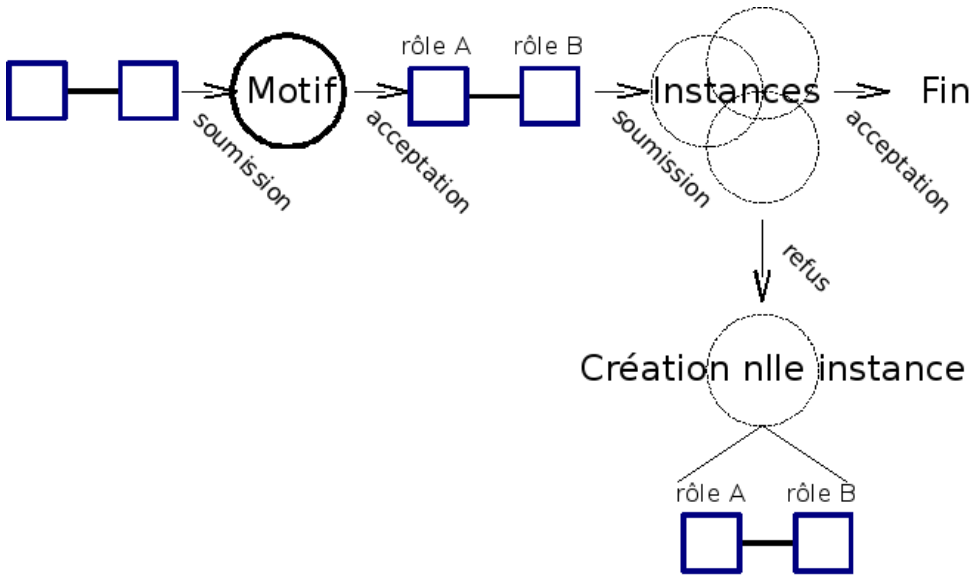


FIGURE 3.15 – Processus de création des instances de motif architectural

- 3) une relation ne peut voir ses deux extrêmités avoir les mêmes rôles dans deux instances d'un même motif d'architecture. Ainsi une relation peut appartenir à plusieurs instance d'un motif d'architecture mais de sorte qu'une au moins de ses extrêmités occupent des rôles différents.

Ces règles permettent de produire les ArchitecturalPatternInstance mécaniquement sur la base de l'énumération des relations et de leur soumission à l'ArchitecturalPattern pris en compte. En effet, en considérant la construction incrémentale des instances d'un motif d'architecture par ajout de nouvelles relations, pour chaque nouvelle relation, soit :

- il existe une instance dans laquelle un couple de rôles acceptables pour la nouvelle relation définit une position libre, on ajoute alors la relation à cette instance pour ces rôles. La définition de liberté d'une position étant donnée par les règles 1 et 2.

- il n'existe pas d'instance dans laquelle un couple de rôles acceptables pour la nouvelle relation définit une position libre, on crée une instance de motif d'architecture à laquelle on ajoute la relation pour un des couples de rôles que ses extrêmités peuvent occuper,

Finalement, la règle 3 permet de stopper le processus de création des instances de motifs d'architecture pour une relation donnée et un motif d'architecture donné. Par contre elle contraint fortement la couverture de l'architecture par les motifs architecturaux. En effet deux instances d'un même motif architectural ne peuvent en leur sein donner la même position à une relation de l'architecture. Toutefois cette limitation ne nous a pas semblé un problème majeur face au vaste problème de détermination de la couverture d'un graphe par un autre plus petit.

3.4.4 Le type de contrat (`ContractType`)

Nous choisissons de définir et d'utiliser un type de contrat qui nous permettra d'instancier une forme de contrat donnée en différents points d'une architecture pour en contraindre des motifs potentiellement récurrents. Cela nous permet de décomposer la validation de l'architecture. Nous choisissons de définir le type du contrat sur la base :

- d'un motif d'architecture tel que défini précédemment,
- d'un type de spécifications : pour chaque élément satisfaisant au motif d'architecture, donc susceptible d'être un participant de l'instance de contrat, on ne considère qu'un sous ensemble de ses spécifications, défini par des conditions sur celles-ci (par exemple sur leur formalisme, leurs portées, etc...),

Restreindre l'ensemble de spécifications considéré dans un contrat permet tout comme dans le cas de la portée architecturale du contrat de faciliter l'analyse de l'état du contrat vis à vis de l'architecture. Ainsi il est possible de distinguer les problèmes de comportements de ceux de QoS et même d'être plus fin dans le cas par exemple où plusieurs ensembles de spécifications contraignent une même propriété : par exemple si on a à la fois des pre et post conditions et des contraintes sur les traces d'appels comme les Behavior Protocol. Par ailleurs il faut noter que généralement les spécifications exprimées dans deux formalismes différents ne sont pas comparables, et donc leur compatibilité est exprimée de manière séparée dans deux contrats distincts. D'autre part certains motif architecturaux n'ont de sens que pour certaines spécifications.

Par exemple, si nous considérons le motif d'architecture décrivant le design pattern "sujet-observateur" que nous avons vu, le type de contrat associé à ce motif ne retiendra que :

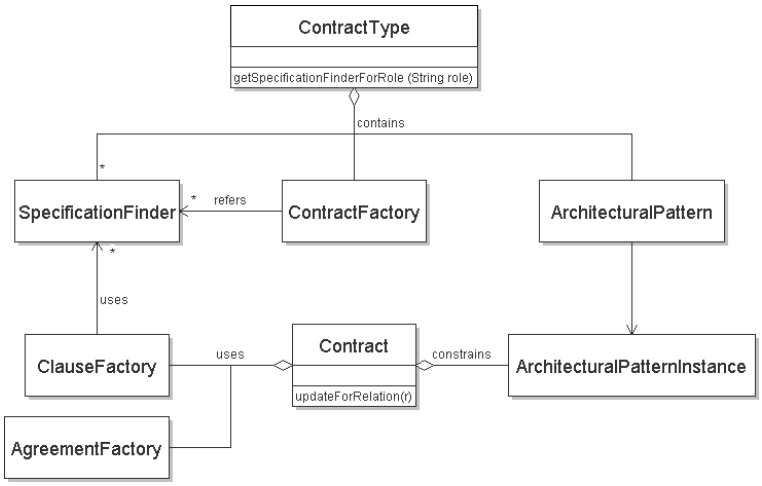
- pour le composant "sujet" les spécifications portant sur l'interface `Observer` (par exemple les assertions sur les méthodes de cette interface)
- pour les composants "observateurs" les spécifications portant sur l'interface `Subject-Observer`

Dans ce cas nous voyons en plus que les conditions qui portent sur les spécifications acceptables peuvent dépendre de la position de l'élément dans le motif d'architecture.

3.4.4.1 Principe

Le type de contrat via le motif architectural qu'il inclut détermine les éléments d'architecture impliqués dans le contrat. Mais nous avons vu qu'en plus, associé à un formalisme donné, le type de contrat spécifie à l'aide du `SpecificationFilter` :

- quelles spécifications d'un participant sont éligibles pour devenir des clauses d'un contrat donné : par exemple, dans le cas d'un formalisme assertionnel on peut sélectionner les assertions suivant leur objet ou portée (par exemple dans un contrat entre deux interfaces, les spécifications retenues ne porteront que sur les méthodes de celles-ci), etc... Par ailleurs suivant la position du participant dans le



motif architectural les spécifications éligibles le concernant peuvent varier : classiquement dans le cadre d'un simple contrat entre le fournisseur et le consommateur d'un service, un type de contrat pourra, suivant des taux de fiabilité, être par exemple plus sévère avec l'un des participants qu'avec l'autre, en retenant plus de spécifications le contraignant,

- comment synchroniser la spécification contractuelle avec l'architecture. En effet dans certains cas, suivant l'endroit auquel est appliqué la spécification sur le système concret il est nécessaire d'inverser les prédicats, l'hypothèse devient la garantie, et vice versa. L'exemple le plus simple consiste à traduire un couple d'assertions "pre-post" en un couple "hypothèse-garantie" associé à une méthode. Si nous considérons qu'elles s'appliquent sur la méthode serveur, la précondition sera l'hypothèse et la post condition la garantie. Mais au moment d'appliquer cette spécification contractuelle, si il s'avère que la méthode contrainte est cliente il faudra inverser les prédicats d'hypothèse et de garantie. De manière plus générale si la spécification n'explicite pas si l'objet qu'elle contraint est requis ou fourni, il faut choisir une convention de transformation en hypothèse-garantie, tout en étant prêt à mettre à jour la spécification suivant que son objet se révèle concrètement requis ou fourni dans l'architecture.

La compatibilité des clauses, qui est l'objet de l'accord, peut s'écrire de diverses manières. Il s'agit toutefois toujours d'exprimer la compatibilité entre les hypothèses et les garanties des différents participants. Le type du contrat, qui est dédié à un formalisme, fournit la fonction qui permet de construire un accord sur la base des clauses. Suivant la forme que prend l'expression de compatibilité, il est plus ou moins possible de distinguer l'origine de son non respect. La forme la plus fine de l'expression de la compatibilité consiste à expliciter, pour chaque relation du motif d'architecture, la compatibilité des clauses portant sur chacune de ses extrêmités. Ainsi quand cette expression n'est plus valide, il est possible de remettre en cause finement la relation architecturale qui lui est associée.

3.4.5 Cycle de vie du contrat

La sémantique du contrat réside dans l'explication des différents états que lui et ses constituants peuvent prendre. Nous avons vu que l'objet du contrat était d'exprimer la

validité d'une instance de motif architectural sur la base des spécifications de ses participants en vérifiant conformité et compatibilité de celles-ci. Ainsi de manière globale nous distinguerons deux états principaux du contrat :

- fermeture : un contrat est dit "fermé" si :
  - il dispose des références sur tous les éléments d'architectures (composants, services...) formant l'instance du motif architectural défini par le type du contrat, et qui sont alors ses participants,
  - il dispose des spécifications des participants au contrat nécessaires pour établir leur compatibilité (entre eux), et telles qu'on puisse vérifier leur conformité à leur porteur,
- ouverture : un contrat est dit "ouvert" si certains participants définis par le motif architectural de son type sont absents de l'architecture qu'il contraint, ou bien si il manque des spécifications pour établir la compatibilité des participants entre eux. Les spécifications peuvent en effet ne pas être disponibles, ou bien n'être plus représentatives de leur porteur comme nous verrons dans la partie sur les contrats hiérarchiques, ou dans le cas de négociations qui les modifient.

Dans le cas où le contrat est fermé son évaluation va refléter la validité de l'assemblage telle que nous avons défini cette dernière. Nous distinguerons alors deux états suivant le résultat de l'évaluation du contrat fermé :

- contrat "satisfait" : cet état est obtenu quand d'une part les participants se conforment à leurs spécifications, c'est à dire si les clauses du contrat sont satisfaites et d'autre part que les clauses sont compatibles, c'est à dire que l'accord est vérifié,
- contrat "insatisfait" : cet état est obtenu quand : soit une clause au moins n'est pas vérifiée, c'est à dire qu'un porteur n'est pas conforme à sa spécification, soit l'accord n'est pas vérifié, c'est à dire qu'il y a une incompatibilité entre deux spécifications,
- contrat "inconnu" : cet état est celui du contrat "fermé" qui n'a pas encore été évalué,

L'assemblage est donc valide suivant la définition que nous en avons donné tant que le contrat est "fermé" et "satisfait".

Pour chaque ajout d'un élément d'architecture appartenant à son instance de motif d'architecture, le contrat obtient les spécifications relatives aux échanges que ce participant effectue dans le motif architectural. Ces spécifications sont transformées en clauses, dont le participant est alors responsable, et ajoutées au contrat. L'accord du contrat est aussi mis à jour suivant l'ajout de clauses des participants. Une fois le contrat fermé, alors que les clauses et l'accord n'ont pas encore été évalués, le contrat est dans un état "inconnu". Puis suivant que clauses et accord évalués sont satisfaits ou non, le contrat passe de l'état "satisfait" à "insatisfait". Quand un élément d'architecture du motif architectural est retiré de celle-ci, le contrat perd un de ses participants et devient "ouvert" alors que les clauses de ce dernier sont supprimées et l'expression de

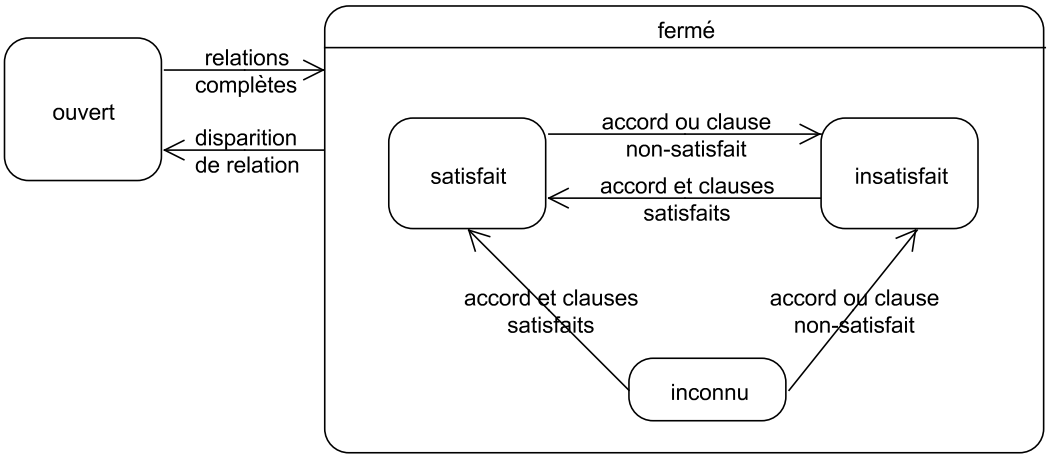


FIGURE 3.16 – Cycle de vie du contrat

l'accord mise à jour. Quand une spécification est invalidée, la clause correspondante est supprimée et l'accord aussi mis à jour.
Le contrat suit donc les évolutions de l'architecture et des spécifications avec lesquels il doit rester synchronisé pour que son état en soit un reflet pertinent.

3.4.6 Evaluation du contrat

L'évaluation des différents constituants du contrat (hypothèse, garantie, clause, accord) repose sur l'observation du système de composants. Toutefois il y a une importante différence entre les observations nécessaires à l'évaluation de l'accord et celles nécessaires aux clauses. Par principe l'accord évalue la compatibilité de spécifications portant sur les mêmes observations. Les clauses suivent par contre la sémantique de l'approche hypothèse-garantie : la garantie doit être vraie tant que l'hypothèse l'est, c'est à dire que les observations sur lesquelles on évalue la garantie doivent être concomittantes ou postérieures à celles d'évaluation de l'hypothèse. Par ailleurs, suivant les formalismes, les observations peuvent être déclenchées par l'activité du système contraint ou effectuées à la demande du système de contrat. Afin de prendre en charge ces différents besoins, nous associons à chaque élément évaluable (clause, accord, contrat) du contrat un gestionnaire d'observation et un gestionnaire d'évaluation. Le gestionnaire d'évaluation prend la décision de déclencher l'évaluation quand le gestionnaire d'observation le notifie qu'il a reçu un ensemble d'observations cohérentes. Pour obtenir ces observations le gestionnaire d'observations peut soit attendre, soit demander des observations. Ainsi le gestionnaire d'observation reçoit les observations déclenchées par le système contraint, ou est à l'origine des observation effectuées par le système de contrat. Il faut encore noter que l'évaluation d'un élément évaluable peut nécessiter ou être déclenchée par celle d'un autre, classiquement après la réévaluation d'une clause, le contrat doit être à nouveau évalué. La définition du moment auquel est effectuée l'évaluation d'un élément évaluable est laissé à la charge de l'intégrateur de formalisme. Ce dernier peut ainsi adapter la politique d'évaluation du contrat aux propriétés de son formalisme. Par exemple, il pourra suivant le coût de l'évaluation de la conformité et de la compatibilité les recalculer plus ou moins souvent, voire à la demande

seulement...

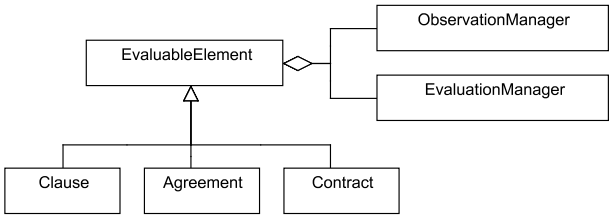


FIGURE 3.17 – Eléments évaluables du contrat

3.5 Modèle hiérarchique

Comme nous l'avons vu dans l'état de l'art, le paradigme hypothèse-garantie favorise une approche compositionnelle des spécifications. Différents travaux ont tendu à lier la spécification d'un ensemble d'éléments à l'ensemble de celles de chacun d'entre eux. Ceci est en particulier intéressant dans le cas des architectures hiérarchiques (comme celle du composant <Car> englobant les composants <CruiseCtrl> et <Engine>, figure 3.18) dans le cadre desquelles il est important de lier la spécification d'un élément composite à celles de ses sous-éléments, puisque le composite est censé les représenter.

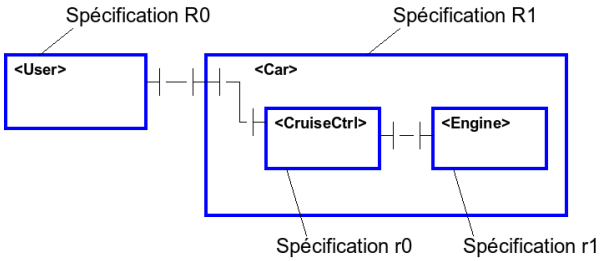


FIGURE 3.18 – exemple de modèle hiérarchique

La liaison entre la spécification du composite et celles de ses sous-éléments peut intéresser les contrats de deux points de vue :

- ouverture et fermeture du contrat : valider ou invalider la spécification du composite, c'est à dire s'assurer que la spécification du composite le représente bien : cela revient à s'assurer que la spécification du composite est bien la composition des spécifications des sous-éléments. Or si la spécification (R1) du composite est invalidée alors le contrat entre le composite <Car> et d'autres éléments (<User>) devient "ouvert",

- responsabilités : rechercher la responsabilité de la violation de la spécification du composite parmi ses sous-éléments : les résultats sur la composition des spécifications font souvent apparaître que si les spécifications entre sous-éléments sont compatibles alors une violation de la spécification composite (R1) est indissociable d'une violation d'une spécification d'un sous-élément (r0, r1),

La plupart des théorèmes de composition que nous avons rencontrés montrent comment une spécification de l'élément composite se déduit des spécifications des sous élements pourvu qu'une expression "prémisse" soit satisfaite. Cette prémisse exprime dans les différents cas la compatibilité des spécifications des sous-composants. Elle peut donc servir d'expression d'accord au contrat entre les sous-éléments. Dans ce cas la spécification de l'élément composite dépend du contrat entre les sous-éléments. Il est donc intéressant dans le cadre de spécifications dont le formalisme autorise la composition hiérarchique de modéliser le lien entre la spécification du composite et le contrat entre les sous-composants, ce que nous faisons à l'aide d'une `CompositeContractualSpecification`. Cette dernière est une `ContractualSpecification` associée à l'élément composite mais qui observe l'accord du contrat entre ses sous-éléments, comme représenté sur la figure 3.19.

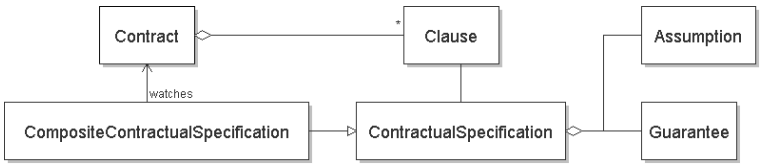


FIGURE 3.19 – CompositeContractualSpecification

Dans le cas de notre modèle, nous pouvons considérer une `CompositeContractualSpecification`, basée sur R1, comme une ContractualSpecification associée à <Car>, et dont la validité dépend du contrat entre les sous-éléments de ce dernier ("contrat <CruiseCtrl>-<Engine>", figure 3.20). Ce type de spécification permet ainsi d'établir un lien entre contrats qui suit le lien de hiérarchie réalisé par l'architecture comme le montre la figure 3.20.

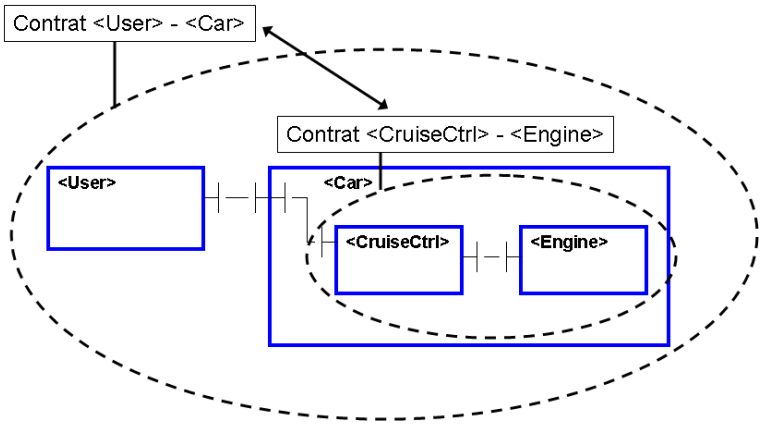


FIGURE 3.20 – Lien entre des contrats sur les éléments d'une hiérarchie

Toutefois un point essentiel ne doit pas être omis, il s'agit du respect des hypothèses du théorème de composition dont on utilise le prémisse comme expression d'accord pour le contrat entre les sous-composants. Nous verrons plus en détail certaines de ces hypothèses dans le cadre de la mise en oeuvre du modèle de contrat pour le théorème d'Abadi et Lamport (cf 4.3.4.4).

3.5.1 Exploitation des dépendances entre contrats

Pour étudier les dépendances entre des contrats à des niveaux hiérarchiques différents, deux approches sont possibles : la première part d'une observation au niveau composite et en infère des résultats au niveau des sous-composants, la seconde part d'une observation au niveau des sous-composants pour en déduire des résultats au niveau composite. Nous nommerons la première "approche descendante", la seconde "approche montante".

3.5.1.1 Approche descendante

Dans ce cas nous détectons que l'élément du niveau composite viole sa spécification. Certains théorèmes de composition, comme celui d'Abadi et Lamport (cf 2.5.3), permettent de déduire que si l'accord du contrat entre les sous-composants n'est pas violé alors c'est qu'un (au moins) des sous-éléments viole lui même sa spécification. En effet pour un formalisme satisfaisant aux hypothèses d'Abadi et Lamport, il est possible de choisir pour expression de l'accord entre les sous-composants la prémisse de la règle d'inférence du théorème. Or la conséquence de cette règle est que le respect des spécifications des sous-composants implique celui de celle du composite. Ainsi si la spécification du composite est violée c'est qu'une spécification au moins d'un sous-composant l'est aussi.
Il est donc possible d'adapter la politique de vérification (comme discuté à la fin de 3.4.6) des contrats afin de stocker les observations sur les sous-composants et de ne lancer de vérification des clauses que lorsqu'un sous-composant est soupçonné d'avoir violé sa spécification (NB : l'accord doit lui par contre être toujours vérifié que ce soit par observation ou preuve). Par ailleurs ce raisonnement peut se répéter en descendant dans les niveaux de composition si les sous-composants sont eux-mêmes des composites etc....

3.5.1.2 Approche montante

Dans le cadre de cette approche nous observons le contrat entre les sous-composants et tirons des conclusions au niveau composite. Deux cas de figure se présentent : un élément viole sa spécification, l'accord entre les éléments n'est plus valide.

Violation de spécification

Quand un sous-élément viole sa spécification, sans que l'accord soit violé, il n'est pas possible d'affirmer que la spécification composite n'est plus valide. Il est en effet tout à fait possible qu'un sous-composant viole sa spécification sans que cela se répercute sur le composite qui l'englobe.

Violation de l'accord

Si l'expression de l'accord est la prémisse du théorème de composition qui fournit la spécification composite, alors la violation de cette expression remet en cause la validité de cette dernière. Ainsi, de la même manière qu'un élément peut ne plus satisfaire à sa spécification (au sens de la réaliser), une spécification composite peut ne plus représenter un élément composé. Par exemple, si nous choisissons pour prémisse du théorème qui produit la spécification de `<Car>`, l'accord du contrat entre `<CruiseCtrl>` et `<Engine>`,

alors la violation de l'accord de ce contrat remet en cause la validité de l'inférence fournissant la spécification de `<Car>`. La spécification de `<Car>` n'est donc plus légitime par rapport à celles de ses sous-composants, si leur contrat est violé via son accord.

Dans ce cas, le contrat auquel participe un élément en déphasage avec sa spécification, comme alors `<Car>` passe dans un état inconnu. En effet la spécification associée au participant `<Car>` ne le représente plus et donc l'accord qui la fait intervenir n'est plus représentatif de la bonne collaboration du composant `<Car>` avec `<User>`. Comme précédemment cette conséquence peut être propagée dans les niveaux de hiérarchie, en remontant cette fois-ci. Les accords des contrats à des niveaux supérieurs de hiérarchie peuvent cesser d'être pertinents ou représentatifs du système et donc les contrats passés évoluer vers un état inconnu.

3.6 Conclusion

Le modèle de contrat dont nous avons présenté la conception et la constitution satisfait au positionnement et aux propriétés que nous lui avons fixés. Il présente par ailleurs l'avantage de pouvoir être appréhendé via la métaphore du contrat de la vie courante. En effet, il en utilise les principaux éléments : participants, clauses et accord. La constitution de ces derniers est guidée par la contrainte des échanges entre participants. Comme ces échanges sont eux mêmes définis par les relations architecturales, le contrat contraint par contre coup la configuration architecturale du système. La valeur de retour du contrat, envisagé comme une fonction de l'architecture et de ses spécifications, reflète donc bien la validité de la configuration du système. De plus notre approche considère les rôles des différents intervenants autour de la contractualisation d'une architecture logicielle. Notre modèle est ainsi un outil sur lequel peuvent s'appuyer l'administrateur du système, son assembleur et le fournisseur de ses spécifications, ainsi que les intégrateurs d'architecture et de formalisme.

CHAPITRE 4

Mise en oeuvre

4.1 Introduction

Dans le chapitre précédent nous avons présenté les principes de notre modèle de contrat ainsi que les hypothèses qu'il faisait sur l'architecture et les formalismes auquel il s'appliquait. Sur la base de ces hypothèses et des besoins du contrat nous avons défini une modélisation générique minimale des architectures et des formalismes associés par le contrat. Cette modélisation nous a ainsi permis de découpler le contrat des particularités des différents formalismes de spécification (propriété n° 4) et de l'implémentation de l'architecture (propriété n° 5). Dans cette partie nous présenterons cette modélisation (4.2 et 4.3), ainsi que la mise en oeuvre du contrat, conçue pour être effectuée par le système de contrat, sur des exemples concrets tirés de l'application décrite au chapitre précédent (4.3.4).

4.2 Architectures

Nous souhaitons que le modèle de contrat s'applique potentiellement à différentes architectures. Il est en prise avec celles-ci via les objets suivants :

- l'instance de motif architectural : cet objet désigne dans l'architecture concrète les éléments et les relations, qui répondant aux conditions du motif architectural, forment le schéma que le contrat contraint,
- les participants au contrat : ces objets doivent désigner dans l'architecture concrète les éléments, responsables des clauses du contrat, qui sont issus de l'instance de motif architectural,
- les observations du système contraint à partir desquelles sont évaluées les hypothèses et les garanties,

Ce découplage du modèle de contrat et de l'architecture repose sur deux APIs génériques, l'une de modélisation de l'architecture, utilisée par une seconde de modélisation des observations. Pour chaque architecture concrète à laquelle le modèle de contrat est appliqué une spécialisation de chacune de ces deux APIs est nécessaire. Cette spécialisation incombe à l'intégrateur de l'architecture qui la met en oeuvre dans l'"adaptateur d'architecture" (cf. 3.3.4). Nous allonc donc décrire chacune de ces APIs et montrer à chaque fois comment elle s'applique à un modèle d'architecture donné : le modèle de composants Fractal.

4.2.1 Modélisation de l'architecture

4.2.1.1 Eléments et relations

Les motif d'architecture et participants du contrat ont besoin de désigner les éléments et relations de l'architecture. Mais surtout cette désignation doit être indépendante de l'implémentation concrète de ces derniers. Pour ce faire les éléments d'architecture, composants ou services, sont modélisés par une unique classe `Reference`. La spécialisation de celle-ci pour une architecture concrète détient la référence réelle vers l'élément d'architecture. Les attributs des éléments d'architecture, qui sont aussi des éléments d'architecture, par exemple les interfaces et les méthodes des composants, peuvent être aussi désignés par des classes `Reference`. Les relations entre éléments désignés par des `Reference` sont modélisées par des classes `Relation`. Ces classes `Relation` représentent des relations aussi bien de configuration, comme des connexions entre interfaces de composants ou l'inclusion d'un composant dans un autre, que des relations plus statiques par rapport à l'exécution, comme le lien entre un composant et une de ses interfaces. Le lien avec les constituants du contrat est représenté sur la figure4.1.

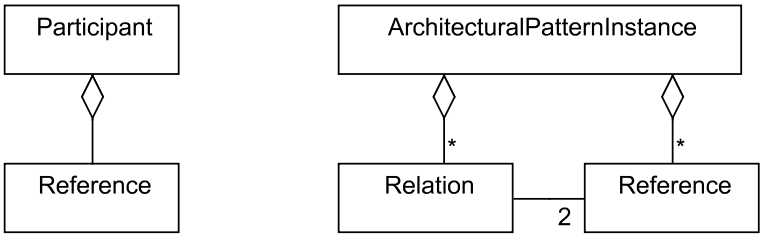


FIGURE 4.1 – Liens entre contrat et architecture.

Ainsi pour appliquer le modèle de contrat à la plateforme Fractal, il faut spécialiser les classes `Reference` et `Relation` comme illustré dans la figure 4.2.

4.2.1.2 Chemins

Les `Reference` et les `Relation` permettent de fournir au système de contrat une représentation générique des architectures mais ils ne permettent pas de désigner un ou plusieurs de ses éléments. Or le système de contrat nécessite un moyen d'obtenir de telles références. Pour répondre à ce besoin nous nous sommes inspirés de l'approche utilisée par XPath [106] pour désigner des noeuds dans un document XML.

XPath. XPath permet de désigner un ensemble de noeuds dans un document XML, comme étant les extrêmités d'un chemin composé d'une suite de relations parmi celles qui existent entre les noeuds d'un document XML. La sémantique de XPath repose sur une suite de sélections de noeuds : par exemple "/child : :A/child : :B/child : :C" sélectionne les noeuds enfants de la racine du document qui sont nommés "A", puis les noeuds enfants des noeuds nommés "A" et qui sont nommés "B", puis les noeuds enfants de ces derniers qui sont nommés "C", etc.... De manière plus générale il permet d'appliquer trois contraintes décrites pour chaque étape de la sélection de la manière suivante "...//axe : :nodetest[predicate]/..." :

- "axe" : est une contrainte portant sur la relation entre les noeuds recherchés et les

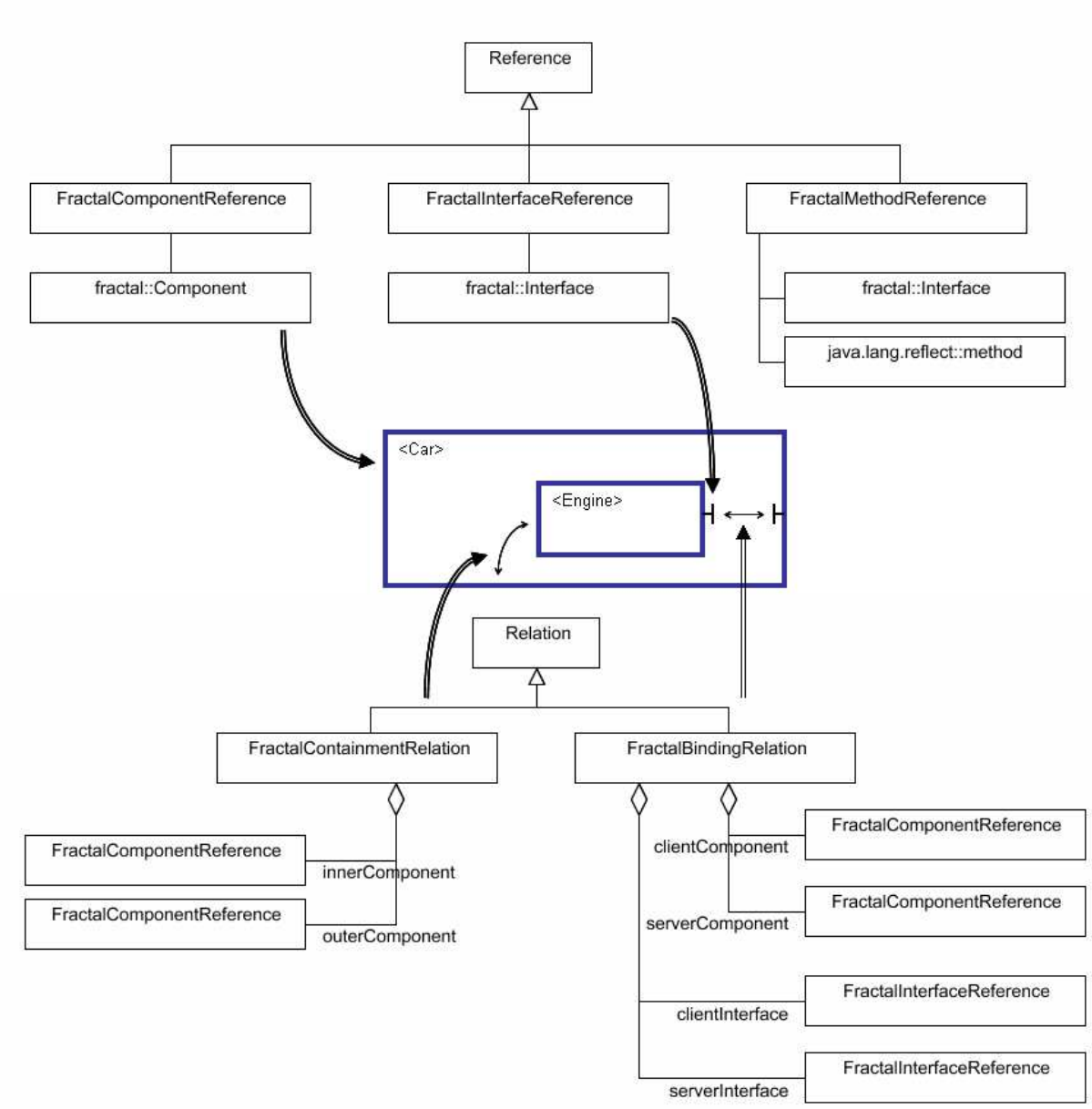


FIGURE 4.2 – Spécialisation des apis pour Fractal

noeuds de la sélection précédente, un axe simple est "child : :" qui sélectionne les noeuds enfants de ceux de la précédente sélection,

- "nodetest" : s'applique à la sélection courante et filtre les noeuds sélectionnés par l'axe, par défaut il est possible de remplacer "nodetest" par le nom du noeud recherché, mais il est aussi possible de placer à cet endroit un test sur le type du noeud (commentaire ...),

- "predicate" : il s'agit d'un prédicat agissant comme un filtre sur la sélection courante et contraignant les propriétés du noeud, par exemple quand la sélection courante comporte plusieurs noeuds le prédicat permet d'en extraire un en spécificant un numéro d'ordre dans la sélection,

APath. Nous avons donc défini une modélisation simplifiée des chemins répondant à notre besoin de désignation d'une référence. Cette solution comme XPath repose sur le parcours d'un chemin de relations architecturales et d'un filtrage des références obtenues à chaque étape, servant de point de départ à la suivante. Toutefois elle doit être indépendante de l'architecture tout en étant adaptable au plus grand nombre. Ainsi nous n'avons retenu que deux éléments essentiels :

- le `Link` : équivalent de l'axe, il permet de déduire d'une ou plusieurs références, une ou plusieurs références atteintes via une relation architecturale donnée (méthodes `resolve*Reference` de la figure 4.3). Ainsi pour une relation de connexion, il retournera l'élément d'architecture connecté à un autre, pour une relation d'inclusion il retournera les éléments d'architecture inclus dans un autre, etc,

- le `Node` : équivalent du prédicat, il permet de filtrer un ensemble de références pour ne conserver que celles qui satisfont à une condition donnée (méthode `architectureMatch` de la figure 4.3). Basiquement il peut simplement s'agir du nom de l'élément d'architecture, comme le nom d'une interface ou d'un composant, ou de contraintes sur d'autres propriétés...

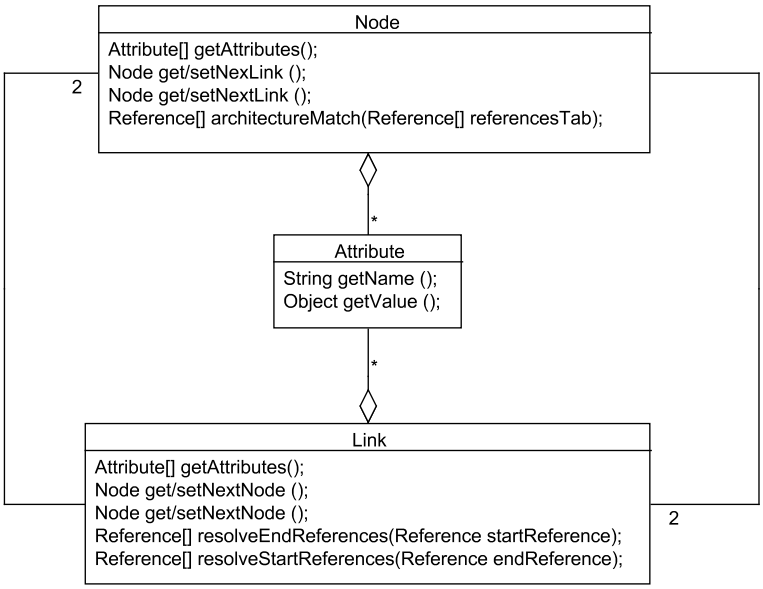


FIGURE 4.3 – Architecture générique des Links et Nodes

Les chemins de désignation sont ainsi constitués d'une suite alternée de `Link` et de `Node`. Pour plus de généricité ces classes disposent chacune d'un ensemble, non limité, d'attributs génériques, stockés sous forme de paires nom-valeur. Ces attributs peuvent notamment être utilisés pour stocker les paramètres de filtrage des `Node` (comme un nom, ou une autre caractéristique de l'élément d'architecture recherché). Ils permettent ainsi aux `Node` et `Link` de prendre en compte des caractéristiques de filtrage variées suivant les propriétés propres à chaque type d'architecture sans modifier leurs interfaces. Les organisations du principe générique sont décrites dans la figure 4.3.

Application à Fractal

La mise en oeuvre dans le cas de Fractal des chemins est décrite dans la figure 4.4. Dans le cadre de l'architecture Fractal les différents types de relations décrivent l'architecture de la manière suivante :

- le `FractalSubComponentLink` lie un composite (`FractalComponentNode`) à un de ses sous composants (`FractalComponentNode`),
- le `FractalBindingLink` lie une interface cliente (`FractalInterfaceNode`) à une interface serveur (`FractalInterfaceNode`),
- le `FractalConstitueOfLink` lie une interface (`FractalInterfaceNode`) au composant qui l'expose (`FractalComponentNode`), ou une méthode (`FractalMethodNode`) à l'interface à laquelle elle appartient (`FractalInterfaceNode`),

Ainsi les trois types de `Node` et `Link` associés à Fractal suffisent à décrire complètement n'importe quel système bati sur ce modèle de composant. Par exemple pour obtenir une référence sur une interface, nous utiliserons un chemin du type suivant (avec une syntaxe paraphrasant celle de XPath, et en partant du noeud racine du système) : `/FractalSubComponentLink::[FractalComponentNode(name="Car")] /FractalConstituteOfLink::[FractalInterfaceNode(name="sns")]`.

4.2.2 Modélisation des observations

4.2.2.1 Pincipe général

Nous définissons les observations de manière abstraite par rapport à l'architecture afin que cette approche soit réutilisable dans différents systèmes. Une observation se définit par une donnée à récupérer dans un contexte lorsqu'un déclencheur donné est actionné. Dans notre approche nous distinguons la description d'une observation, de l'observation effective (cf. figure 4.5). La description d'une observation (ObservationDescription) revient à décrire la donnée à observer (DataDescription) ainsi que le déclencheur (TriggerDescription). Une fois le déclencheur actionné, un objet observation est produit qui contient la valeur observée (Value), la(es) donnée(s) de déclenchement (Trigger), ainsi qu'une référence à sa description (ObservationDescription) au cas où un raisonnement serait nécessaire sur la description de départ.

4.2.2.2 Modèle générique

L'ObservationDescription. La description d'observation, `ObservationDescription`, est constituée de l'ensemble d'objets présentés dans la figure 4.6. Elle contient

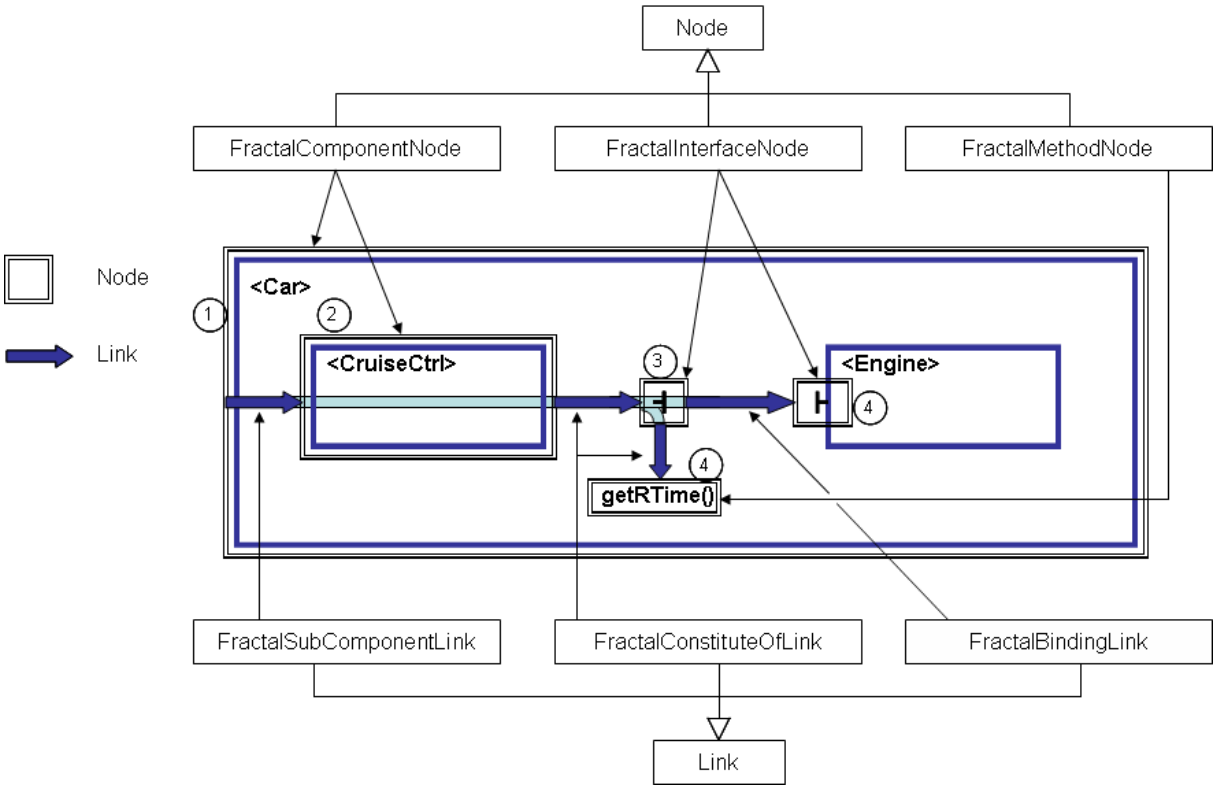


FIGURE 4.4 – Application des Links et Nodes à l'architecture Fractal.

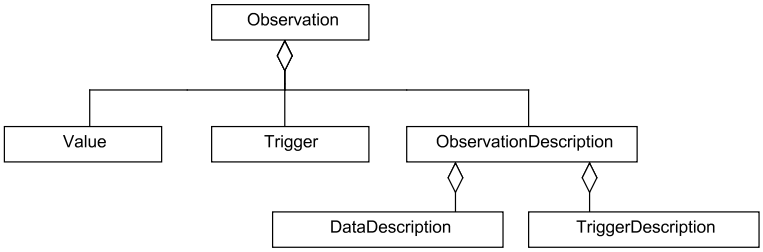


FIGURE 4.5 – diagramme de classes de l'observation

aussi le nom associé à l'observation qu'elle décrit. Elle possède encore la référence de l'élément d'architecture sur lequel porte les observations qu'elle décrit (`base` sur la figure 4.6). En effet l'objet des observations est de permettre à une clause de contraindre son responsable dans le système contractualisé. Ainsi chaque description d'observation, toujours associée à la spécification d'une clause, désigne l'élément du système à observer. Cette référence, stockée dans la description, sert de base à la résolution des chemins utilisés dans les `DataDescription` et `TriggerDescription` pour désigner les éléments du système sur lesquels ils portent. Le `DataDescription` est une description de la donnée que doit contenir l'observation réalisée, et qui par défaut ne propose qu'une méthode permettant d'extraire la donnée de la référence de base et du `Trigger`. Dans l'implémentation spécifique à une plateforme donnée, cette classe sera spécialisée avec le code nécessaire à l'obtention de la donnée pour cette plateforme. Le `TriggerDescription` est une description du déclencheur de l'observation. Dans son implémentation pour une plateforme donnée, cette classe sera spécialisée pour s'interfacer avec les outils de monitoring du système contraint (de plus bas niveau, par exemple des outils d'interception des appels de méthodes, etc). Enfin, les `Watcher` sont les écouteurs de l'observation qui sont notifiés quand celle-ci a lieu et qui présentent cette interface.

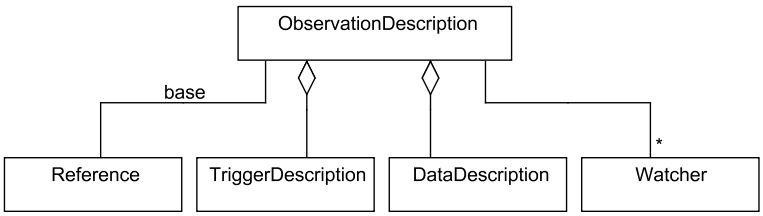


FIGURE 4.6 – diagramme de classes de description de l'observation

Fonctionnement. Quand la description du `TriggerDescription` se réalise, alors un objet `Trigger` est instancié par cette dernière et transmis à l'`ObservationDescription` qui l'utilise pour obtenir du `DataDescription` la valeur qui lui sert à produire l'`Observation`. Une fois celle-ci obtenue elle est envoyée aux `Watcher` qui se sont enregistrés auprès de l'`ObservationDescription`.

TriggerDescription. La description du déclencheur `TriggerDescription` définit la nature de son critère de déclenchement : détection d'événement, etc. Elle contient aussi les paramètres configurant ce critère de déclenchement : propriétés de l'événement, etc. Par ailleurs le `Trigger` contient les données ayant provoquées le déclenchement, c'est à dire satisfaisant à la nature et aux paramètres du critère du déclencheur (`TriggerDescription`). Par exemple une nature de déclenchement sera l'interception de l'exécution, configurée par le nom de la méthode à intercepter. Dans ce cas le `Trigger` contiendra alors la pile d'exécution ayant provoqué le déclenchement. Nous pouvons remarquer que la cause du déclenchement peut être variée dans le temps et la géographie du système contraint. Nous pouvons distinguer différentes origines géographiqes du déclenchement comme illustré sur la figure 4.7 :

- a) déclenchement à origine interne au déclencheur : par exemple le déclencheur

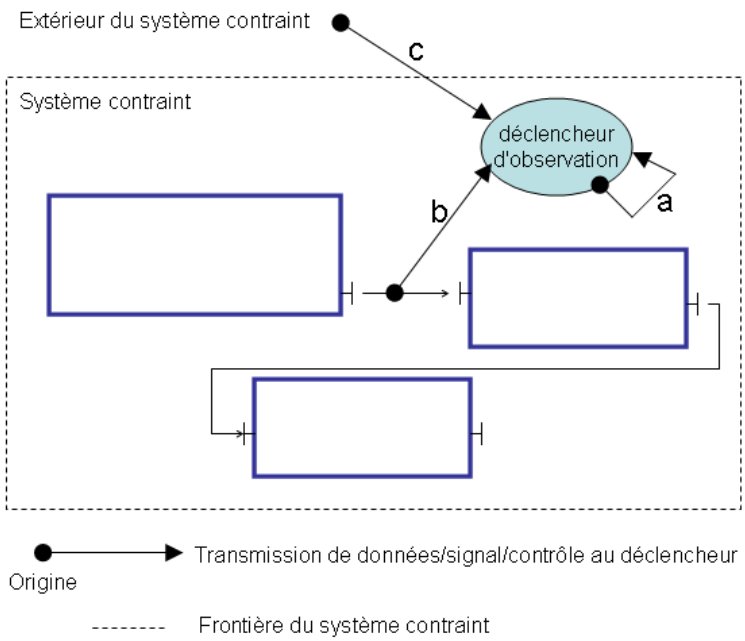


FIGURE 4.7 – Origines du déclenchement de l'observation

contient un timer et déclenche une observation à intervalles fixes, il s'agit alors d'un événement mais qui ne sort pas du déclencheur,

- b) déclenchement à origine externe au déclencheur et interne au système : nous considérons que le déclencheur peut réagir à un événement provoqué par le système contraint par le contrat. L'exemple typique est l'interception d'un flot d'exécution du système contraint, ou l'entrée du système contraint dans un état donné,
- c) déclenchement à origine externe au déclencheur et externe au système : le déclencheur réagit dans ce cas à un événement issu de l'extérieur du système contraint, par exemple un système d'observation peut solliciter des observations à différents instants qu'il définit. Ainsi si un système de validation externe au système contraint est déclenché par l'utilisateur, il peut demander que des observations soient effectuées pour évaluer des fonctions reflétant des propriétés du système contraint. Typiquement, il s'agit de tests lancés par l'utilisateur à un moment choisi par lui,

Il faut noter que suivant la sémantique des formalismes de spécifications, les contraintes que ces dernières expriment peuvent porter sur des durées plutôt que sur des instants. L'exemple le plus courant est celui d'OCL [82] dans lequel les invariants sont à satisfaire sur des durées. Dans ce cas, c'est dans le cadre de l'interprétation du formalisme en termes d'hypothèses et garanties que la traduction de ces durées en événements discrets doit être opérée. Toutefois, il est possible que les observations développées pour une architecture doivent être enrichies pour un formalisme particulier. Par exemple les formalismes de QoS peuvent requérir la mise en oeuvre d'observations diverses ainsi que de sondes.

Pour obtenir une base générique de modélisation des déclencheurs nous avons

défini trois classes de `Trigger` de spécialisation croissante présentées dans la figure 4.8 qui couvrent différentes sources de déclenchement :

- `TriggerDescription` : décrit un déclencheur sans besoin de communication particulier, ce pourra être le déclencheur à origine interne,
- `EventTriggerDescription` : décrit un déclencheur qui réagit à des événements dont l'origine peut aussi bien être externe que interne au système contraint,
- `InterceptionEventTriggerDescription` : décrit un déclencheur qui réagit à des événements issus de l'interception de l'exécution du système contraint (déclencheur à origine interne au système) : il s'agit de l'exécution au sens large, c'est à dire de fonctionnalités aussi bien fonctionnelles (exécution d'interactions métier) que non fonctionnelles (action de configuration etc...) du système contraint (pourvu qu'on puisse les observer),

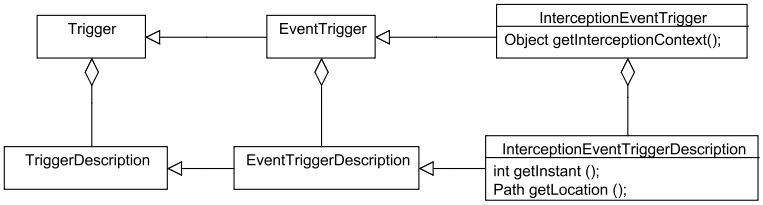


FIGURE 4.8 – Trigger générique

L'InterceptionEventTriggerDescription décrit un déclencheur qui réagit à l'entrée ou à la sortie (`getInstant()`) de l'exécution d'un élément d'architecture. Celui-ci est désigné, vis à vis de l'élément d'architecture auquel est attachée l'observation, par la résolution du chemin retourné par `getLocation()`. Le `Trigger` qui contient les données de déclenchement permet d'accéder au contexte d'exécution, modélisé de manière générique par un `Object`.
Dans le système Fractal il peut être ainsi spécialisé de deux manières (cf figure 4.9) :

- non fonctionnelle : les déclencheurs `ComponentStartTriggerDescription` et `ComponentStopTriggerDescription` réagissent au démarrage ou à l'arrêt d'un composant donné, ils sont de la catégorie de déclencheurs sensibles à la configuration du système de composants. L'observation qu'ils décrivent ne contient pas d'autres informations que le composant dont ils notifient le démarrage ou l'arrêt.
- fonctionnelle : le déclencheur `FractalInterceptionTriggerDescription` réagit à l'interception d'échanges de messages entre les composants, c'est un déclencheur sensible à l'activité métier du système de composants. L'observation qu'il décrit contient les données relatives à l'exécution qu'il a intercepté (paramètres de méthode, référence de l'appelant, de l'appelé etc...).

DataDescription Dans le cadre de Fractal nous avons défini différents types de données à observer. Il faut toutefois noter que certains peuvent requérir des types de déclencheurs spécifiques, car ces derniers peuvent ne pas fournir tous les mêmes informations :

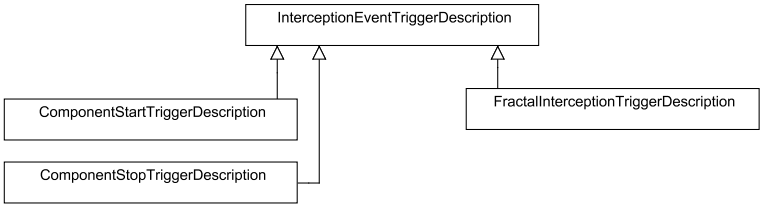


FIGURE 4.9 – Déclencheurs dédiés à la plateforme Fractal

- `ArchitecturalDataDescription` : désigne la référence d'un élément d'architecture par un chemin qui est résolu par rapport à l'élément de référence de l'observation,
- `ParameterInterceptionDataDescription` : désigne la valeur prise par le paramètre (défini par son index) d'un appel de méthode intercepté, cette description est à utiliser conjointement avec une `FractalInterceptionTriggerDescription` en entrée de méthode,
- `ReturnInterceptionDataDescription` : désigne la valeur de retour d'un appel de méthode intercepté, cette description est à utiliser conjointement avec une `FractalInterceptionTriggerDescription` en sortie de méthode,
- `CalleeInterceptionDataDescription` : désigne la référence de l'élément d'architecture émetteur de l'appel de méthode intercepté, cette description est à utiliser conjointement avec une `FractalInterceptionTriggerDescription` en entrée ou sortie de méthode,
- `PreInterceptionDataDescription` : désigne la valeur d'un paramètre d'entrée fournie au moment de l'interception de la sortie de la méthode, à utiliser avec une `FractalInterceptionTriggerDescription` en sortie de méthode,
- `FormulaInterceptionDataDescription` : désigne le résultat de l'évaluation d'une formule qu'elle contient, la formule peut réutiliser les résultats d'autres observations associées au même déclencheur, elle est à utiliser avec une `FractalInterceptionTriggerDescription`,

Organisation des observations. Les observations sont organisées dans le cadre d'un `ObservationNotificationManager` décrit dans la figure 4.10. L'objet du `ObservationNotificationManager` est de maintenir synchronisés les observateurs (`Watcher`), les requêtes d'observations (`ObservationDescription`) et les déclencheurs. En effet les contrats comme l'architecture qu'ils contraignent peuvent évoluer dynamiquement et indépendamment. Le contrat peut évoluer du fait d'une renégociation (c'est à dire la modification d'une spécification), ou du fait de la modification de la configuration architecturale qu'il contraint. La disparition d'un observateur (c'est à dire d'une clause) doit être retransmise jusqu'à l'observation qui le désabonne et la disparition d'une observation, qui par exemple n'a plus d'observateur, doit être propagée aux outils de déclenchement de l'observation (`TriggerDescription`) et de monitoring (etc). Comme pour des raisons de ressources il est possible que les ressources de déclenchement et de monitoring soient partagées par les observations, l' `ObservationNotificationManager`

intervient comme un mediateur entre eux, afin de garantir qu'une ressource de déclenchement ou de monitoring inutilisée est bien finalement relachée et qu'une description d'observation nouvellement appliquée au système partage des ressources (déclencheur etc...) potentiellement déjà existantes.

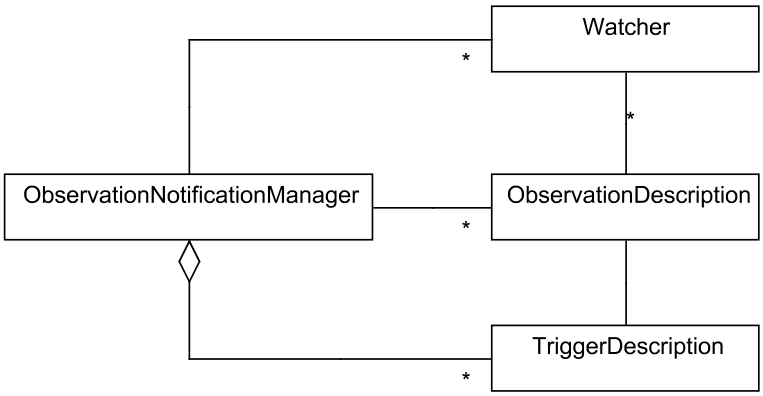


FIGURE 4.10 – Organisation des observations

4.3 Formalismes

Dans cette partie nous allons montrer comment le modèle de contrat est découplé de la nature du formalisme des spécifications qu'il organise. Ces dernières sont utilisées dans les clauses et l'accord du contrat, mais nous commencerons par montrer comment elles peuvent être décrites génériquement pour être ensuite interprétées en terme d'hypothèses et garanties. Ce dernier point nous permettra de discuter la production des clauses et de l'accord, sachant que notre système devra effectuer mécaniquement ces opérations, depuis la production de la description générique jusqu'à la construction et vérification du contrat, sur la base des spécifications dans leur formalisme original.

4.3.1 Modélisation des contraintes sur un système

4.3.1.1 Approche générique

Nous considérons dans notre approche les spécifications comme des contraintes dont on peut évaluer la satisfaction vis à vis du système architectural sur lequel elles portent. Une première étape a donc consisté à modéliser de manière indépendante de leur formalisme ces contraintes. Pour ce faire nous avons remarqué que basiquement chacune était composée d'un prédicat et des observations du système que ce dernier contraignait. Ceci nous a permis de définir un Domain Specific Language [22], qui a pour objet de décrire ces contraintes de manière générique et lisible par un humain [1]. La traduction automatique d'une spécification de son formalisme original dans ce DSL incombe à l'intégrateur du formalisme qui la fournit dans le cadre de l'"adaptateur de formalisme" (cf. 3.3.4) :

[1]NB : nous rappelons que dans nos travaux nous privilégions malgré tout une production automatique des contrats

```
On <un element d architecture>
  Observe {
        ( val: <valeur> at: <instants>; )+
  }
  Verify <propriete>
```

La sémantique de cette expression est relativement simple. Le bloc "On" définit le domaine spatial du système que la règle peut voir, il s'agit de la portée d'un élément d'architecture, composant ou service. Le bloc "Observe" décrit les observations effectuées dans ce domaine. Le bloc "Verify" décrit la partie d'évaluation de la propriété passée en argument, qui s'appuie sur les observations décrites précédemment. Au delà de ces parties communes à tous les formalismes, l'interprétation des observations du bloc `Observe` et de la propriété du bloc `Verify` doivent être spécialisés pour chacun des formalismes qu'on souhaite manipuler à l'aide de notre langage. Cette tâche incombe à l'intégrateur de formalisme dans le système de contrat, sachant que par défaut le système prévoit que l'outil d'évaluation de la propriété du bloc `Verify` soit notifié de l'occurence des observations à l'aide d'événements dits d'observation.

Instants d'observation. Les instants d'observation sont définis par les événements que l'architecture ou ses systèmes de monitoring peuvent émettre. Comme dans le cadre de l'exécution d'un système il est courant de disposer d'une manière ou d'une autre d'interception de l'exécution du système, nous proposons dans notre approche deux mots clés : `entry` et `exit` qui désignent l'entrée de l'exécution dans un élément d'architecture et sa sortie de celui-ci. Ainsi, dans le cas de l'exemple du système contraint de CruiseControl :

```
On <Engine>
  Observe {
        val : throttle at : entry setThrottle (float throttle,
    float TS);
}
```

cette expression observe la valeur du paramètre (throttle) au moment où l'exécution entre dans la méthode `setThrottle` offerte par le composant `<Engine>`. Par ailleurs il est possible que des formalismes requiert des observations s'appliquant à des durées, dans ce cas il revient à la charge du traducteur de spécification (humain ou informatique) de discrétiser cet intervalle en instants discrets.

Cohérence temporelle. Nous n'abordons pas ici explicitement le problème de l'organisation temporelle des observations. En effet, il est possible que la spécification du bloc `Verify` requiert que les observations soient effectuées dans un certain ordre, ou fasse de manière générale porter une contrainte sur leur organisation temporelle. La description de l'organisation temporelle générique des observations nous a en effet semblé un problème trop vaste pour être traité dans le cadre de cette thèse. Nous proprosons que dans la description des instants puissent être incluses des conditions temporelles faisant éventuellement intervenir les autres observations du bloc en désignant leurs événements déclencheurs. Toutefois la syntaxe, la sémantique de ces expressions et leur interprétation sont laissés à la charge de l'intégrateur de formalisme, c'est à dire la personne fournissant l'outil de traduction du formalisme dans notre langage. Il faut noter que cette solution autorise la mise en oeuvre d'événements composites pour déclencher les observations ce qui est d'un intérêt certain. Par contre, il faut prendre

garde aux cycles d'événements qui pourraient apparaître si par exemple une observation était déclenchée par une condition externe (par exemple une interception sur le système) ou par une autre observation, alors que cette dernière se produit consécutivement à la première. Ainsi l'événement relatif à une observation est noté `e_val` où `val` est le nom de la valeur observée. Le mot clé `with` désigne une condition temporelle sur les événements, il est à noter que cette condition peut faire intervenir l'événement de déclenchement de l'observation à laquelle elle appartient (noté `e_this`). Par exemple, si le ";" dénote une séquence et "*" une répétition indéfinie d'un ensemble d'événements :

```
On <Engine>
  Observe {
        val : evt.getSpeed at : exit sns.getSpeed ();
        val : throttle at : entry csp.setThrottle (float throttle,
     float TS) with (e_getSpeed;e_this)* ;
}
```

Dans cette expression, la première valeur observée est simplement un événement nommé `getSpeed` associé à la sortie de l'exécution de la méthode `getSpeed()`. Ensuite l'observation porte sur le paramètre `throttle` lu à l'entrée de l'exécution dans la méthode `setThrottle`. Toutefois le bloc `with` vient contraindre l'instant de la lecture du paramètre en spécifiant que cet instant doit toujours se produire juste après une occurence de l'événement `getSpeed` (soit la lecture de la vitesse courante).

Moment d'évaluation de la spécification. Le moment d'évaluation de la spécification n'est pas décrit explicitement dans notre langage car il dépend de l'outil d'évaluation utilisé ou de la configuration du système. Classiquement une observation peut aussi bien être utilisée au moment de son obtention pour évaluer au plus tôt sa spécification afin de pouvoir réagir en quasi temps réel à un problème, ou bien elle peut être stockée pour une évaluation différée du comportement du système. Un compromis peut être atteint dans le cas d'une évaluation qui tire partie de moments de faible charge du système pour s'exécuter sur des données sauvegardées etc. Dans ces différents cas l'observation et la spécification sont identiques, seul diffère le moment d'évaluation.

4.3.1.2 Application à des formalismes concrets

Nous allons dans cette partie étudier comment deux formalismes, CCLJ et les Behaviour Protocol peuvent être interprétés dans le langage simple que nous venons de décrire. Pour ce faire nous nous placerons dans le cas du système contraint décrit dans la partie 3.2.

CCLJ

La traduction de spécifications écrites en CCLJ est relativement immédiate, ainsi une spécification de la forme :

```
        on <Composant> context f(float valeur) :
          pre : condition (valeur)
```

se réécrira sous la forme :

```
        On <Composant>
        Observe {
```

```
            val: valeur at : entry f(float valeur);
        }
        Verify : condition(valeur)
```

L'interprète de notre langage comportera alors un interprète du formalisme du prédicat `condition` pour calculer sa valeur à l'aide de la valeur observée `valeur` dont il est notifié.

Behavior Protocol

Si nous considérons la spécification du composant `<CruiseCtrl>` suivante :

```
        (?sns.on; !csp.setThrottle*; ?sns.off)*
```

Elle traduit qu'après avoir reçu un appel sur la méthode `sns.on()`, le `<CruiseCtrl>` va émettre un nombre non limité d'appels à `csp.setThrottle`, avant de recevoir un appel sur sa méthode `sns.off()`.
Elle peut se réécrire de la manière suivante :

```
        On <CruiseCtrl>
        Observe {
                val : evt.on at : entry sns.on();
                val : evt.on at : entry sns.off();
                val : evt.setThrottle at : entry csp.setThrottle();
                }
        Verify : (?on; !setThrottle*; ?off)*
```

Dans ce cas l'interprète de notre langage doit être spécialisé pour qu'un "runtime checker" vérifie l'expression du Behaviour Protocol du bloc `Verify` au fur et à mesure qu'il est notifié des événements d'observations spécifiés dans le bloc `Observe`.

4.3.1.3 Modélisation

Du point de vue du modèle de contrat, les spécifications appliquées aux éléments d'architecture peuvent être modélisées en suivant les concepts de notre langage spécialisé, comme montré sur la figure 4.11. Nous nommons description de règle (RuleDescription) la traduction d'une spécification d'un formalisme quelconque dans notre langage. Nous distinguerons cette forme descriptive d'une forme évaluable que nous nommerons `EvaluableRule` que nous décrirons dans la partie suivante. L'objectif de la description est de contenir des expressions (spécifications et observations) dont l'interprétation mécanique produit, pour chacun des outils d'évaluation potentiel, une forme évaluable de la contrainte.

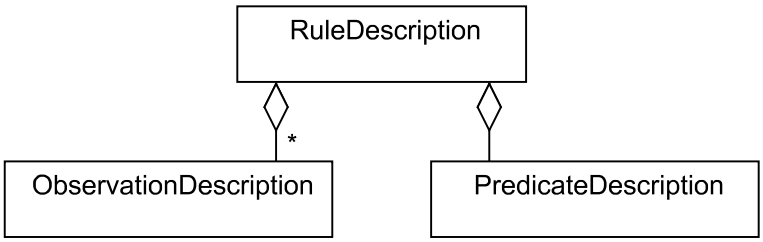


FIGURE 4.11 – Description de la traduction d'une spécification

4.3.2 Modélisation de contraintes sous une forme évaluable

4.3.2.1 Evaluation d'une contrainte

Les contraintes que nous considérons se présentent sous la forme de prédicats. Par défaut ceux-ci sont destinés à être évalués dans un environnement initialisé avec les ressources qu'ils requièrent, aussi bien en fonctions (par exemple via des références à un analyseur statique ou dynamique (model ou runtime checker), etc) qu'en variables (les valeurs observées, etc), comme illustré figure 4.12. Pour ce faire dans le cadre de l'implémentation générique nous considérons que les contraintes sont interprétées par l'interprète de script Beanshell dans un environnement correctement initialisé. Toutefois il est bien sur possible de mettre en oeuvre un autre outil d'évaluation des contraintes du moment qu'il s'intègre dans la modélisation que nous allons présenter. En effet, nous n'ignorerons pas qu'il peut être difficile, voire impossible de traduire une contrainte en expression "java/Beanshell". Dans ce cas l'interprète appelle l'outil d'évaluation adapté à la contrainte. C'est par exemple le cas avec les Behavior Protocol qui ne sont pas traduits en expression java, mais dont les outils d'évaluation, "*runtimeChecker*"et "*modelChecker*", sont appelés par l'interprète.

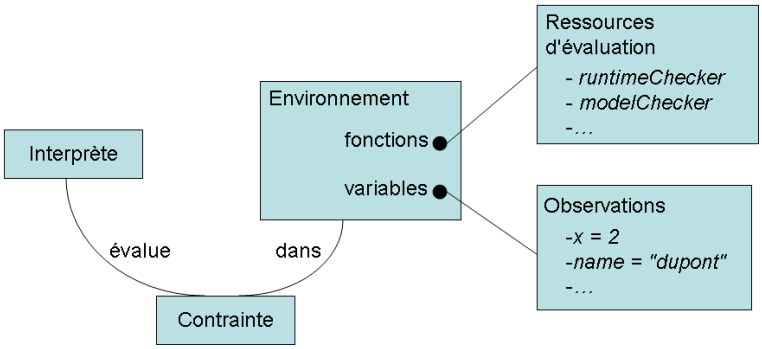


FIGURE 4.12 – Interprétation d'une contrainte

Le Beanshell[2]**.** Le Beanshell est un interprète d'expressions écrites avec la syntaxe de java. En outil de script classique il permet de définir des variables et des fonctions dans son environnement d'évaluation. Ces définitions peuvent être effectuées depuis un programme java manipulant une instance de l'interprète. Cette instance de l'interprète peut évaluer des expressions et en retourner le résultat au programme qui l'utilise. Enfin il est possible de définir dans l'environnement du Beanshell des variables qui pointent vers des objets de l'environnement du programme qui manipule l'interprète. Classiquement il est possible d'utiliser dans une expression évaluée par le beanshell, une ressource de calcul externe au Beanshell mais référencée sous forme d'un objet par une variable dans l'environnement du Beanshell. Par exemple, il est possible de définir dans le Beanshell une fonction "runtimecheck(string x)" qui prend en entrée l'expression à tester à l'exécution et qui dans son code utilise une variable "evt" contenant une référence au dernier événement détecté. Le Beanshell donne alors la possibilité d'initialiser une variable statique à la fonction à l'aide de l'expression "x", de mettre à jour la variable "evt" et d'appeler un runtimechecker référencé dans l'environnement d'évaluation de la fonction "runtimecheck".

[2]http ://www.beanshell.org/

4.3.2.2 Modélisation d'une contrainte évaluable

La contrainte évaluable obtenue à partir d'un `RuleDescription` est une `EvaluableRule` qui est un `EvaluableElement` contenant un prédicat. Comme les autres `EvaluableElement`, l'`EvaluableRule` dispose d'un gestionnaire d'observation qui recueille (et éventuellement requiert) les observations nécessaires à son évaluation. Elle dispose aussi d'un gestionnaire d'évaluation qui est notifié quand les observations nécessaires sont présentes et procède alors à l'évaluation. Elle est décrite dans la figure 4.13.

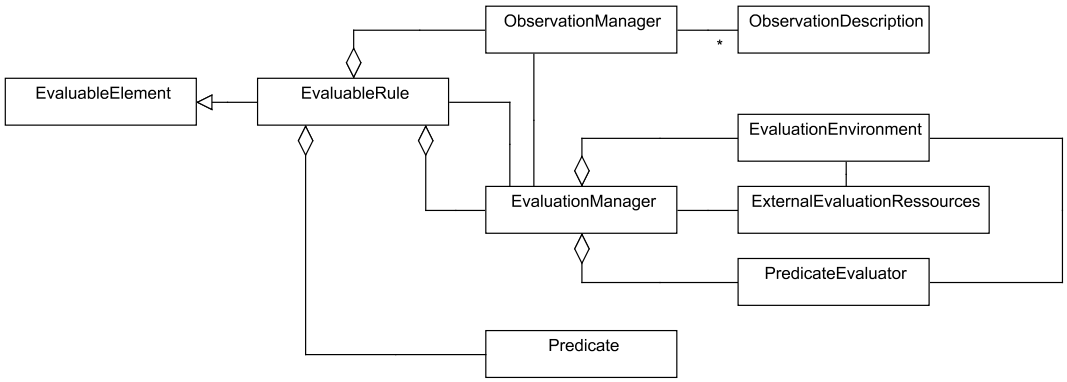


FIGURE 4.13 – Règle évaluable

La modélisation d'une contrainte sous une forme évaluable nécessite plusieurs mécanismes :

- *la traduction* : il s'agit de traduire (automatiquement) l'expression descriptive de la contrainte, ainsi que les noms des observations en expressions interprétables par l'outil d'évaluation choisi. Par exemple dans le cas du Beanshell la syntaxe des identifiants doit suivre les règles de celle de java. Les expressions faisant intervenir ces identifiants doivent par la suite aussi être mises à jour dans celle de la contrainte.
- *l'initialisation de l'environnement d'évaluation* : en particulier les ressources d'évaluation : au moment de la création de l'`EvaluableRule`, l'`EvaluableEnvironment` doit être initialisé avec les références aux objets fournis par les `EvaluationRessources`,
- *l'interprétation des contraintes d'organisation des observations* : l'`ObservationManager` doit être initialisé avec les descriptions des observations qu'il va devoir observer et les contraintes sur ces dernières qui les rendent acceptables. Il s'agit de prendre en compte les conditions exprimées dans la clause "with" du langage de description des règles.

4.3.3 Interprétation des spécifications en hypothèses et garanties

Comme nous l'avons présenté dans l'état de l'art sur les approches "hypothèse garanties", celles-ci mettent en oeuvre des spécifications qui contraignent l'état de l'élément d'architecture, ou celui de son environnement. Dans cette mesure, un élément est considéré responsable d'une spécification lorsque celle-ci contraint exclusivement son état

propre. Cette spécification est alors une garantie qui peut être conditionnée par une hypothèse, c'est à dire une spécification contraignant exclusivement l'état de l'environnement de l'élément. Dans notre approche nous ne considérons plus l'état des éléments, car nous manipulons des éléments d'architecture "boite noire". Nous remplaçons donc l'état d'un élément par ce qu'il émet ou expose, et l'état de son environnement par ce que l'élément en reçoit ou peut en percevoir. Ainsi, parallèlement à l'approche classique par état, un élément d'architecture ne peut agir que sur les grandeurs qu'il expose ou émet. Il ne peut modifier les grandeurs qu'il référence ou reçoit de son environnement, et qu'il utilise pour produire celles qu'il expose. L'élément d'architecture n'est donc responsable, au sens donné dans le chapitre "Modèle de contrat" que des spécifications qui contraignent des grandeurs qu'il émet. Par contre il peut requérir la satisfaction de spécifications sur des grandeurs qu'il reçoit et utilise pour produire celles qu'il garantit. Ainsi distinguer une hypothèse ou une garantie associée à un élément d'architecture ne peut se faire que sur la distinction des grandeurs qu'il émet ou expose de celles qu'il reçoit ou référence, et qui sont définies dans sa description. Notre objectif est de traduire (mécaniquement et automatiquement) une spécification d'un élément d'architecture en un couple de `RuleDescription` dont l'une est l'hypothèse et l'autre la garantie avec la sémantique de l'approche hypothèse-garantie.

4.3.3.1 Catégorisation des observations

Nous pouvons distinguer les observations impliquées dans les spécifications d'après leurs objets que nous rangeons dans deux catégories qui correspondent à deux types d'échanges entre éléments d'architecture :

- un élément d'architecture est l'objet d'un échange entre des éléments d'architecture, et l'objet de l'observation : cela se produit dans le cas d'un élément d'architecture référencé par un autre, par exemple quand une interface requise référence ou désigne une interface fournie, un composant fournit une interface à un autre. Dire que l'élément d'architecture est l'objet de l'observation signifie qu'une de ses propriétés est observée. Cet échange a pour caractéristique d'être d'une durée plus courte que la durée de vie des éléments d'architecture mais plus longue que les interactions d'exécution du système,

- une entité logicielle qui n'est pas un élément d'architecture est l'objet de l'échange entre éléments d'architecture et de l'observation : par exemple un message ou un signal peut être échangé entre une méthode requise et une méthode fournie, et il est possible d'observer des propriétés de ce message. Dire qu'il ne s'agit pas d'un élément d'architecture signifie qu'il n'apparaît pas dans la description ou l'introspection architecturale du système contraint. Cet échange est de durée de vie plus courte que celui d'éléments d'architecture.

L'objectif de cette partie est de montrer comment il est possible de distinguer de manière automatique les objets reçus ou requis des objets émis ou fournis par les éléments du système contraint.

Objet de l'observation : élément d'architecture

Dans cette partie nous considérerons les termes qui se retrouvent à la fois dans l'expression de la spécification et dans la description de l'élément d'architecture auquel

la spécification est associée, par exemple un nom de méthode, d'interface etc.... Il faut noter que cette description peut être obtenue par introspection d'une instance de l'architecture et n'est pas nécessairement fournie à l'avance. Dans ce cas "terme" désigne la part d'expression de la spécification dont l'interprétation sur le système correspond à l'élément d'architecture qui y est découvert. Par exemple : il est possible en java d'introspecter une instance d'un objet (ne possédant pas de description architecturale explicite) pour y découvrir une certaine donnée membre sur laquelle une spécification pourra porter pourvu que son formalisme soit assez expressif. Dans ce cas le mot terme représente la désignation de l'élément d'architecture qui nous intéresse dans le formalisme de la spécification. Il faut aussi noter que l'inverse est possible, c'est à dire que le formalisme de spécification peut ne pas explicitement faire référence à l'architecture qui elle est explicite, comme dans le cas de TLA [22] appliqué au modèle de composant Fractal. Dans ce cas on peut utiliser la désignation fournie par le formalisme de description de l'architecture. Le système de contrat doit découvrir si les éléments d'architectures désignés par ces termes sont exposés, émis, possédés par l'élément d'architecture auquel est associé la spécification ou bien s'il les requiert, reçoit ou référence. En effet, si le reste de la spécification décrit les propriétés qu'ils doivent satisfaire , la notion de garantie et d'hypothèse dépend de leur origine.

Maintenant nous considérons l'élément d'architecture auquel est associé la spécification, un composant, service ou objet. Nous considérons aussi les éléments d'architecture qui entrent dans leur composition : opérations, méthodes, interfaces etc... Nous pouvons constater que classiquement chaque élément en aggrège d'autres, nous ne retenons que ceux qu'il expose. Ceux-ci sont pour nous définis par le fait qu'ils sont associés à leur propriétaire pour la durée de vie de celui-ci. Toutefois nous remarquons que ce dernier aggrège aussi des besoins d'éléments d'architectures, qu'il expose aussi dans certains cas, classiquement les composants possèdent des interfaces requises. Pour parler plus simplement nous nommerons éléments d'architecture fournis ceux qui sont aggrégés sans être des besoins et éléments d'architectures requis ceux dont le besoin est aggrégé.

Le point important est de définir quand un "sous" élément d'architecture est fourni ou requis par rapport à l'élément d'architecture, composant, service ou objet, dont il entre dans la composition. En effet suivant qu'il est requis ou fourni le sous élément d'architecture pourra être l'objet d'une propriété garantie ou attendue par son élément d'architecture le plus élevé. C'est à l'intégrateur d'architecture de fournir la fonction qui définit si un élément est requis ou fourni par un autre dans le cadre de l'adaptateur d'architecture (cf. 3.3.4). Nous proposons, pour guider la réalisation d'une telle fonction, une représentation des relations d'aggrégation qui fait apparaître les éléments fournis et requis sur un exemple de composant de la figure 4.14 :

- sur les flèches apparaît si l'élément d'architecture enfant est requis ou fourni par son parent
- sur l'arc de cercle apparaît si l'élément d'architecture enfant est requis ou fourni par l'élément d'architecture à la racine de l'arbre

Transitivité de l'aggrégation. Afin de déterminer si un élément d'architecture enfant est requis ou fourni par l'élément d'architecture racine de l'arbre nous devons poser

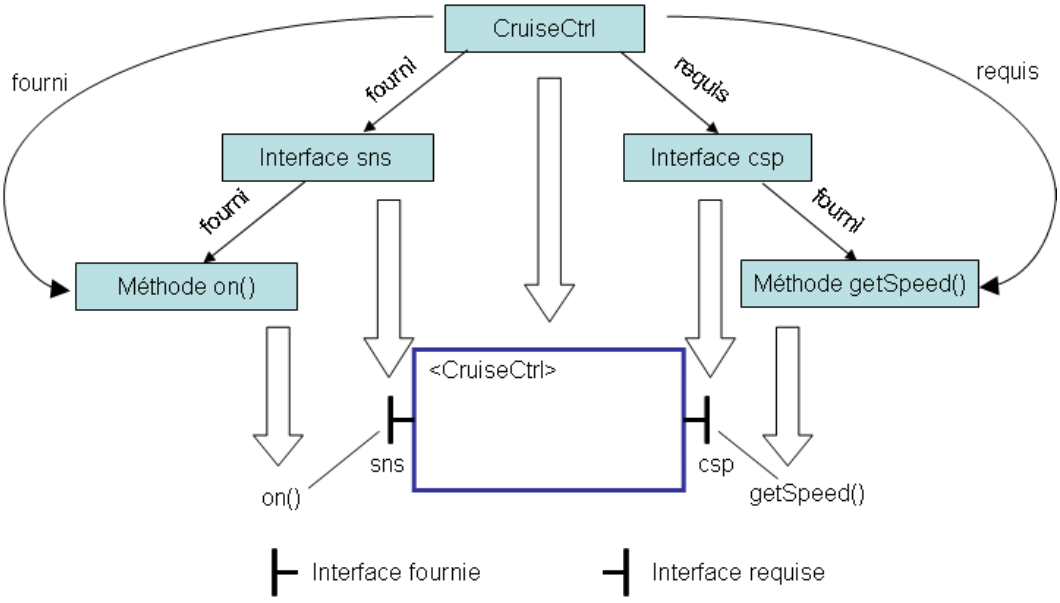


FIGURE 4.14 – Exemple d'aggrégations d'éléments d'architecture

des règles de transitivité de ces propriétés. Ainsi si le noeud A est le père du noeud B, lui même le père du noeud C, nous noterons "(X,Y)=fourni" que "Y est fourni par X" :

- (A,B)=fourni, (B,C)=fourni alors (A,C)=fourni,
- (A,B)=fourni, (B,C)=requis alors (A,C)=requis,
- (A,B)=requis, (B,C)=fourni alors (A,C)=requis,
- (A,B)=requis, (B,C)=requis alors (A,C)=fourni,

Ainsi par introspection de l'architecture et de ses relations, la fonction de l'adaptateur d'architecture détermine le statut requis ou fourni d'un élément par rapport à un autre.

Objet de l'observation : entité échangée

Quand une entité logicielle n'est pas un élément d'architecture, elle n'apparait pas dans la description de l'architecture du système. Toutefois elle est désignée dans la spécification, que nous souhaitons interpréter en hypothèse ou garantie, associée à l'élément d'architecture qui l'émet ou la reçoit. Les paramètres et valeurs de retour des méthodes sont une exception à cette approche car ils peuvent apparaître dans la description architecturale des systèmes contraints. Mais il ne sont pas pour autant des éléments d'architecture au sens de l'aggrégation que nous avons défini précédemment. Comme nous considérons les entités échangées entre deux éléments d'architecture, l'un d'eux s'en trouve être l'émetteur tandis que l'autre est le récepteur. La nature technique de l'échange peut être très variée : copie de valeur pour des paramètres, passage du contrôle dans le cas de signaux etc. Ce qui est important, c'est que l'entité soit émise et reçue par des éléments d'architecture. Pour simplifier le discours nous dirons qu'une entité échangée est fournie par un élément d'architecture quand il l'émet et qu'elle est requise par celui-ci quand il la reçoit.
Cette approche permet d'interpréter les émissions ou réceptions dans les systèmes :

- les paramètres d'une méthode : ils sont requis par la méthode,

- la valeur de retour d'une méthode : elle est fournie par la méthode,
- un appel de méthode : il est requis par la méthode,
- etc : cette énumération pourrait être complétée suivant les différents types d'interaction existant entre éléments, dont une idée de la variété est fournie dans l'état de l'art sur les composants et les services,

Le système de contrat doit maintenant distinguer si l'entité, émise ou reçue par l'élément d'architecture qui la produit ou la consomme, est fournie ou requise par l'élément d'architecture responsable : composant, service ou objet. Nous pouvons à nouveau faire jouer la transitivité que nous avons vue précédemment, en l'étendant aux grandeurs échangées. Un exemple est donné sur la figure 4.15. L'adaptateur d'architecture fournit la fonction qui détermine le statut, requis ou fourni, de l'entité échangée.

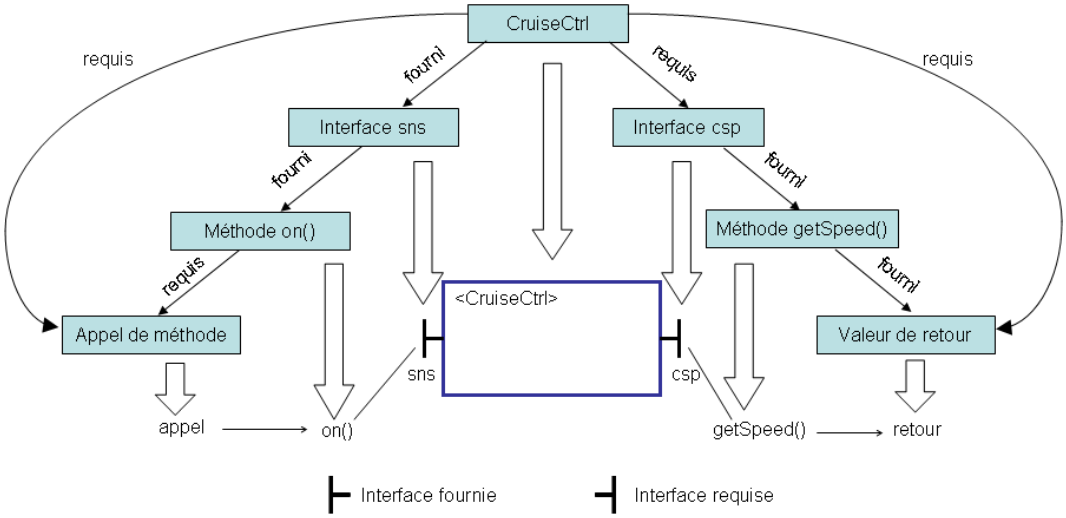


FIGURE 4.15 – Echanges d'entités non architecturales

Conclusion

Ainsi que les observations portent sur des éléments d'architectures ou d'autres entités qu'ils s'échangent, il est possible de déterminer automatiquement celles dont l'élément d'architecture porteur de la spécification est responsable et celles dont son environnement est responsable. Ceci nous permet de partitionner en deux parties l'ensemble des observations, pour distinguer celles sur lesquelles portera l'hypothèse de celles de la garantie, d'une entité architecturale donnée.

4.3.3.2 Interprétation des contraintes

Les règles (`RuleDescription`) que nous avons présentées comportent chacune un prédicat qui contraint un ensemble d'observations. Nous avons vu que chaque règle recherchée, hypothèse et garantie, portait sur un ensemble différent d'observations. Il faut maintenant déduire de la spécification de départ, les prédicats qui vont contraindre chacun de ces ensembles d'observations, de sorte à obtenir une règle hypothèse et une règle garantie, liées par la sémantique de l'approche hypothèse garantie.

Causalité. Si nous considérons P et Q, deux propriétés, l'une sur l'environnement d'un élément d'architecture, l'autre sur l'élément lui même, la sémantique de l'approche hypothèse garantie exprime que *"tant que P est vraie alors Q est vraie"*. Ceci peut s'interpréter par le fait que P est la, ou une des, cause(s) de Q. Il s'agit donc de déduire de la spécification deux contraintes, telle que la satisfaction de la première soit la cause de la satisfaction de la seconde. Nous avons ainsi un premier critère d'interprétation des spécifications.

Séparation des observations. Les grandeurs échangées, entre l'élément d'architecture et son environnement, contraintes par l'hypothèse P doivent être fournies par l'environnement c'est à dire requises par l'élément d'architecture auquel elle est associée. Les grandeurs échangées contraintes par la garantie Q doivent être fournies par l'élément d'architecture associé et requises par son environnement. En dehors de ces contraintes, hypothèse et garantie peuvent faire intervenir des paramètres, éventuellement des appels à d'autres méthodes présentées par les éléments d'architecture etc...

Temporalité. Nous pouvons constater de plus qu'une hypothèse temporelle est faite dans l'approche hypothèse garantie. En effet pour vérifier une telle expression, il faudrait effectuer les observations à des instants où les deux ensembles d'observations sont disponibles, car elle sous entend que P et Q sont vraies au même moment. Toutefois ce n'est pas toujours possible, ne serait ce que parce que P et Q peuvent porter sur des événements discrets séparés dans le temps. Nous serons donc amener à considérer une interprétation de cette règle : *"tant que P est vraie pour un ensemble d'observations données, alors Q est vraie pour des observations faites alors que P n'a pas reçu de nouvelles observations"*. Il s'agit du second critère de définition des prédicats : ils doivent porter sur des observations qui si elles ne peuvent être simultanées doivent être consécutives (on pourrait en effet imaginer une règle faisant porter l'hypothèse sur une première observation, la garantie sur une autre arrivant bien plus tard etc...).

4.3.3.3 Exemples d'interprétation de spécifications

CCL-J

Nous considérons la méthode "`float getRTime (float throttle)`" de l'interface `CarSpeed` du composant `<Engine>`. Cette méthode retourne le temps de réponse nécessaire au moteur pour atteindre l'accélération requise par le paramètre `throttle`. Si nous en interprétons la spécification suivante :

```
on <Engine>
 context float csp.getRTime (float throttle) :
  pre : throttle < 10;
  post : return > 2;
```

Les observations portent sur les valeurs reçues et émises par la méthode `float getRTime`. Le paramètre `throttle` est fourni à la méthode par l'environnement, la valeur de retour est fournie à l'environnement par la méthode. Par rapport au composant `<Engine>`, comme il fournit l'interface `CarSpeed` qui fournit la méthode qui nous intéresse, le paramètre `throttle` peut être considéré comme fourni au composant et la valeur de retour émise par celui-ci. L'hypothèse va donc contraindre le paramètre et la garantie la valeur de retour comme illustré par la figure 4.16.

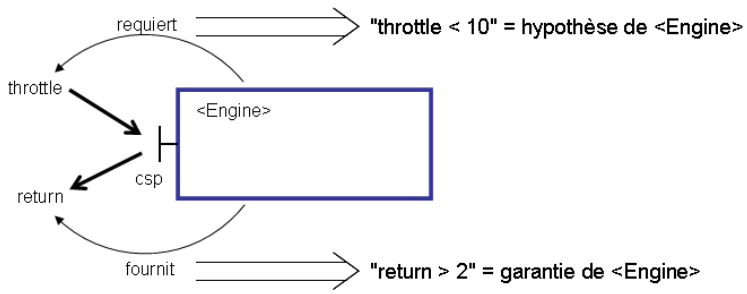


FIGURE 4.16 – Interpretation de spécifications CCLJ

C'est bien ce qui se passe avec l'interprétation classique des pré et post conditions dans le cas où l'interface est fournie (serveur), la précondition est l'hypothèse et la post condition la garantie. Nous obtenons donc deux règles décrites de la manière suivante :

l'hypothèse :

```
On <Engine>
 Observe {
   val : throttle at : entry csp.getRTime (float throttle);
  }
 Verify : throttle < 10
```

la garantie :

```
On <Engine>
  Observe {
    val : return at : exit csp.getRTime (float throttle);
  }
  Verify : return > 2
```

Dans le cas ou la postcondition aurait comporté des valeurs "@pre" nous choisissons de considérer celles ci comme des paramètres de la post condition et non pas comme des grandeurs échangées contraintes par la post condition.

Behavior Protocol

Si nous considérons la spécification du composant `<CruiseCtrl>` suivante :

```
        (?sns.on; !csp.setThrottle*; ?sns.off)*
```

Elle contraint des échanges d'entités non architecturales qui sont des appels de méthode. Dans ce formalisme les "?" et "!" dénotent les messages respectivement reçus et émis par le composant. Nous pouvons d'ailleurs remarquer que leur explicitation (sous forme de "?" et "!") est redondante avec l'interprétation des messages effectuée à l'aide de la règle de transitivité vue précédemment si les appels de méthodes sont considérés comme reçus par les méthodes. Ainsi les appels à"on" et "off" sont reçus par le composant qui émet les appels à "setThrottle".

Cette spécification est évaluée pour chacun des appels à `on`, `off` et `setThrottle` pris séquentiellement, son statut d'hypothèse ou de garantie dépend alors du fait que l'appel est entrant ou sortant. Comme illustré sur la figure 4.17 et comme nous l'avons

défini précédemment, il s'agit d'une hypothèse vis à vis des appels reçus par le composant et d'une garantie vis à vis de ceux qu'il émet.

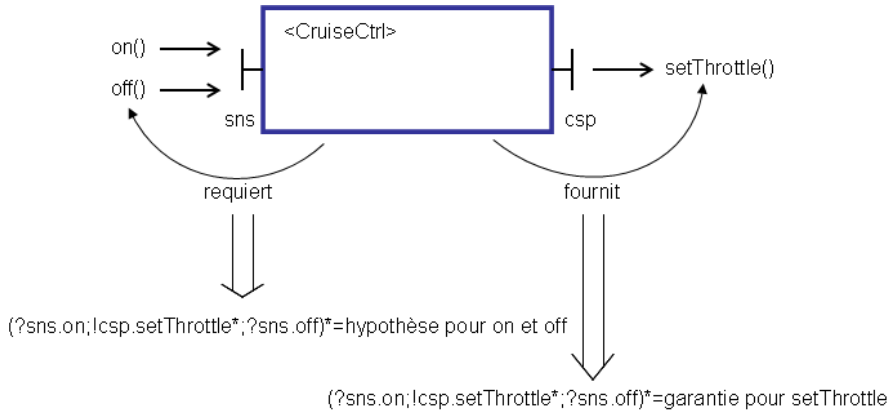


FIGURE 4.17 – Interpretation de spécifications Behavior Protocol

Les prédicats des règles sont mécaniquement produits par le système de contrat sur la base de l'expression de la spécification en décrivant comment elle est interprétée. Nous faisons appel à un runtime checker commun à l'hypothèse et à la garantie. Simplement dans le cas de l'hypothèse il écoutera les messages reçus et dans le cas de la garantie les messages émis. La syntaxe de la contrainte est la suivante : "runtimecheck ("(?sns.on ; !csp.setThrottle* ; ?sns.off)*", "on", "off") ;", ce qui sera interprété au moment de la traduction en règle évaluable de sorte que le runtime checker, ressource externe d'évaluation, notifie cette règle quand un appel à "on" et "off" ne satisfait pas le protocole passé en argument. Le sens de cette approche est d'exprimer que tant que les messages reçus satisfont au runtime checker alors les messages émis doivent y satisfaire aussi. Nous obtenons ainsi deux règles :

l'hypothèse :

```
On <CruiseCtrl>
  Observe {
   val : evt.on at : entry sns.on();
   val : evt.off at : entry sns.off();
  }
  Verify : runtimeCheck ("(?on;!setThrottle*;?off)*", {"on", "off"})
```

la garantie :

```
On <CruiseCtrl>
  Observe {
   val : evt.setThrottle at : entry csp.setThrottle();
  }
  Verify : runtimeCheck ("(?on;!setThrottle*;?off)*", "setThrottle")
```

4.3.4 Mise en oeuvre dans un contrat

Dans cette partie nous envisageons les mécanismes contractuels qui s'appuient sur la modélisation des spécifications. Dans un premier temps, nous considérons leurs artic-

ulations génériques, puis nous verrons comment elles sont spécialisées pour différents formalismes.

4.3.4.1 Production et évaluation des clauses

Création de la clause

La création d'une clause est la conséquence de la découverte pour un participant au contrat d'une spécification contraignant une (ou plusieurs) grandeur(s) qu'il échange avec un autre participant. La spécification, éventuellement déjà traduite en `RuleDescription`, est traduite en un couple de `EvaluableRule`, l'hypothèse et la garantie. Il peut n'y avoir qu'une seule des deux expressions, l'autre est alors considérée comme satisfaite. L'intérêt de la forme `RuleDescription` est quelle peut être stockée dans un référentiel de spécifications (qui les associent à des types d'éléments d'architecture) en attendant d'être appliquée à un système concret et traduite alors en `EvaluableRule`. En effet il est nécessaire de disposer d'un système concret afin d'initialiser :

- l'environnement d'évaluation avec les références aux entités architecturales qui peuvent intervenir dans cette dernière,
- les mécanismes d'obtention des observations requises pour l'évaluation de la spécification,

Une fois les spécifications mises sous la forme d'`EvaluableRule` nous obtenons la configuration de la figure 4.18.

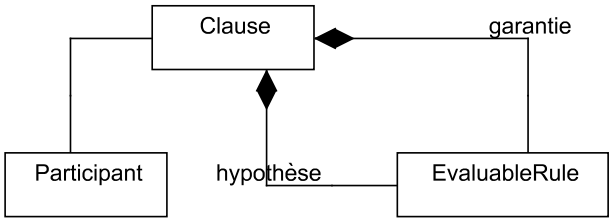


FIGURE 4.18 – Clause

Evaluation de la clause

L'évaluation de la clause prend pour argument le résultat de l'évaluation de l'hypothèse et de la garantie. Le gestionnaire d'observation de la clause observe donc les gestionnaires d'évaluation de l'hypothèse et de la garantie. Il doit notifier le gestionnaire d'évaluation de la clause chaque fois qu'il dispose des observations nécessaires de l'hypothèse et de la garantie, c'est à dire quand la garantie a été évaluée après l'hypothèse. Cette condition peut prendre deux formes :

- l'hypothèse est évaluée puis la garantie,
- la garantie a été déjà évaluée après l'hypothèse mais de nouvelles observations nécessitent la réévaluation de la garantie et pas celle de l'hypothèse,

Ces conditions n'organisent pas les évaluations de l'hypothèses et de la garantie, elles se contentent de les observer. Or nous avons vu que les observations recevables pour l'évaluation de la garantie étaient liées à celles recevables pour l'évaluation de l'hypothèse, au sens où elles devaient avoir lieu après. Le mot "après" peut recouvrir

différentes réalités techniques dont l'approche générique devra autoriser la prise en compte : par exemple, il devra être possible de requérir deux observations soient certes consécutives mais encore provoquées par le même thread etc. Pour ce faire le gestionnaire d'observation de la garantie va observer celui de l'hypothèse. Il sera notifié au même titre que le gestionnaire d'évaluation de l'hypothèse quand celle-ci dispose des observations nécessaires à son évaluation et il pourra accéder à ces observations. Ainsi quand le gestionnaire d'observation de la garantie recevra les observations pour l'évaluation de celle-ci il pourra les comparer à celles de l'hypothèse. Les gestionnaires d'évaluation de l'hypothèse et de la garantie peuvent partager des ressources d'évaluation mais ils ne sont pas pour autant associés. La figure 4.19 présente un diagramme objet de l'organisation générique d'une clause vis à vis de son évaluation.

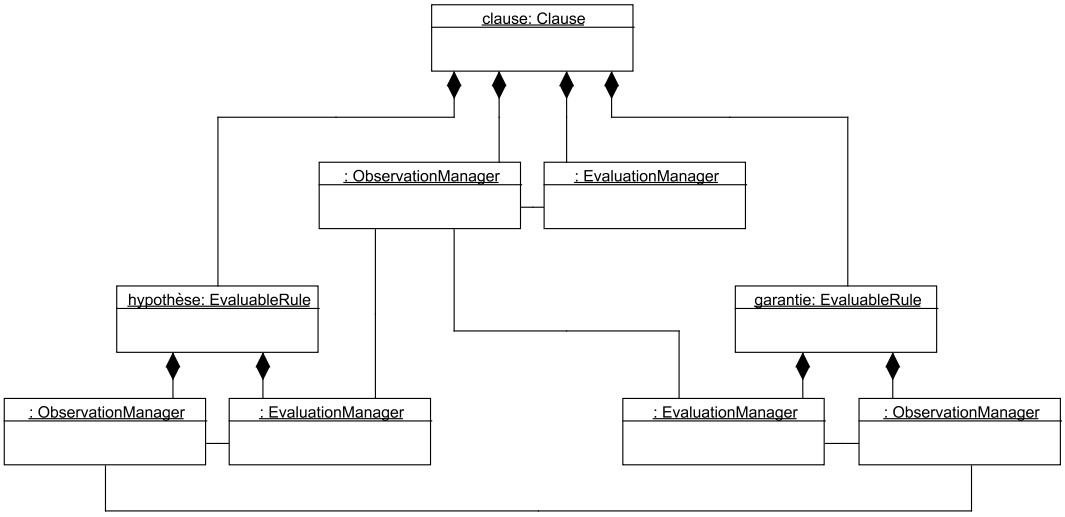


FIGURE 4.19 – Diagramme objet de l'organisation des mécanismes d'évaluation

4.3.4.2 Production et évaluation de l'accord

Constitution et production

L'accord doit regrouper sous la forme d'une ou plusieurs contraintes les conditions sous lesquelles les échanges entre les participants du contrat, via les relations du motif architectural, sont valides. Ces conditions sont celles pour lesquelles les hypothèses sur les grandeurs échangées sont satisfaites par les garanties sur ces mêmes valeurs. Suivant le formalisme qui décrit ces échanges, ces conditions peuvent prendre des formes variées. Toutefois, il s'agit toujours de comparer des hypothèses à des garanties sur des grandeurs échangées. Ainsi suivant le formalisme, il sera possible :

- de comparer des couples de spécifications quand celles-ci ne contraignent pas les échanges qu'entre deux participants : l'une contraignant des grandeurs du côté de son émetteur, l'autre ces mêmes grandeurs du côté de son récepteur. Ainsi pour chaque échange, nous aurons une condition à vérifier. L'accord sera alors la conjonction de ces expressions de compatibilité. Cette approche permettra une étude fine de la validité de l'assemblage car l'invalidation de l'accord pourra être rapprochée d'un échange précis,

- de comparer des ensembles de spécifications : lorsque les spécifications contraignent chacune des échanges, avec plus d'un participant, en les liant dans leurs

expressions, il n'est plus possible de les comparer deux à deux. Il se peut que l'accord se réduise alors à une seule condition bâtie sur l'ensemble des spécifications à comparer.

Si on considère la validation de l'assemblage formé par les composants `<CruiseCtl>` `<Engine>` et `<Car>`, nous rencontrons les deux cas de figure cités pour des spécifications respectivement en CCLJ et au format des Behavior Protocol.

CCLJ Comme le formalisme CCLJ est assertionnel, il contraint dans chacune de ses spécifications un échange entre une fonction cliente et une autre serveur, soit deux éléments d'architecture. Pour que l'échange soit valide nous avons vu qu'il fallait que les attentes du récepteur soient comblées par les garanties de l'émetteur. Dans le cas des assertions cela revient à vérfier que chaque hypothèse est impliquée par la garantie correspondant à la grandeur reçue. Si nous considérons les couples de spécifications que nous avons déjà présentés dans le chapitre "Modèle de contrat" :

```
S1 :
On <Car>
  context int sns.setSecurityLevel (int level)
    pre : level > 3

S2 :
On <CruiseCtrl>
  context int sns.setSecurityLevel (int level)
    pre : level < 5
```

et d'autre part :

```
S3 :
On <Engine>
  context float getRTime (float throttle)
    post : retour < 5

S4 :
On <CruiseCtrl>
  context float getRTime (float throttle)
    post : retour < 10
```

Si nous interprétons en termes d'hypothèses et de garanties ces spécifications il en ressort que :

- S1 est une garantie de `<Car>` sur `level`
- S2 est une hypothèse de `<CruiseCtrl>` sur `level`
- S3 est une garantie de `<Engine>` sur la valeur de retour de `getRTime`
- S4 est une hypothèse de `<CruiseCtrl>` sur la valeur de retour de `getRTime`

De cette interprétation nous pouvons déduire que l'accord regroupera deux conditions :

```
S1 => S2
S3 => S4
```

Ces deux expressions pourront être décrites sous forme de règles `RuleDescription`, simplement les observations associées dépendront de la méthode d'évaluation. Les prédicats à vérifier sont les suivants :

```
(1)     (level > 3) => (level < 5)
(2)     (retour < 5) => (retour < 10)
```

Il faut remarquer que l'accord est invalidé au cas où un de ces prédicats n'est pas satisfait. Il est alors possible de désigner précisément l'échange, ou la paire de composant en cause ainsi que la paire de spécifications incompatibles.

Behavior Protocol Dans le cas des Behavior Protocol il nous suffit d'observer la spécification (déjà présentée dans le chapitre "Modèle de contrat") :

- `<CruiseCtrl>`:

```
(1)  (?sns.on; !csp.setThrottle*; ?sns.off)*
```

du composant `<CruiseCtrl>`, pour constater qu'une spécification peut contraindre et rendre dépendants des échanges entre un composant et plusieurs qui l'entourent, dans notre cas les échanges entre `<CruiseCtrl>` et les composants `<Car>` et `<Engine>`. Il en va de même pour les spécifications de ces deux derniers composants :

- `<Engine>`:

```
(2)  (?csp.setThrottle; !fdk.notifyAction)*
```

- `<Car>`:

```
(3)  (?sns.on; !fdk.notifyAction*; ?sns.off)*
```

Dans le cadre des Behavior Protocol l'expression de la compatibilité peut prendre deux formes :

- compatibilité entre des composants de même niveau de composition, via l'opérateur de composition parallèle $\sqcap$,
- compatibilité entre un composite et la composition de ses sous composants, via l'opérateur de composition verticale $compl$,

Pour vérifier l'assemblage des trois composants, il faut donc vérifier la satisfaction du prédicat suivant :

$$(3)\ compl\ ((1)\ \sqcap\ (2)) \tag{4.1}$$

qui exprime non seulement que `<Engine>` (spécification n°2) et `<CruiseCtrl>` (spécification n°1) sont compatibles, mais que leur composition est compatible avec `<Car>` (spécification n°3). Il n'est pas valable de comparer seulement `<Engine>` et `<Car>`, puis seulement `<CruiseCtrl>` et `<Car>`, car suivant comment `<Engine>` et `<CruiseCtrl>` interagissent le comportement de leur composition peut être ou non compatible avec celui de `<Car>`. En effet les messages qu'ils émettent vers ou reçoivent de `<Car>` peuvent dépendre de ceux qu'ils s'échangent. Par exemple, `<Engine>` n'appelle la méthode `fdk.notifyAction` qu'après avoir reçu un appel sur `csp.setThrottle`. Si `<Car>` requiérait au moins un appel à `fdk.notifyAction` il faudrait que `<CruiseCtrl>` appelle au moins une fois `<csp.setThrottle>` (ce qui a priori n'est pas garantie par la spécification que nous en donnons) etc...

Evaluation

Par ailleurs suivant le formalisme et la technique de vérification, l'évaluation de la satisfaction à une condition de compatibilité peut requérir ou non des observations du système. Il apparaît donc que ces expressions de compatibilité peuvent être modélisées comme des règles évaluables `EvaluableRule`. L'accord est alors une aggrégation de ces règles. Il est lui même un élément évaluable dont le gestionnaire d'observation va observer l'évaluation des conditions de compatibilité et notifier son gestionnaire d'évaluation qui, suivant le formalisme, combinera ces résultats. Cette organisation est rendue par le diagramme objet de la figure 4.20.

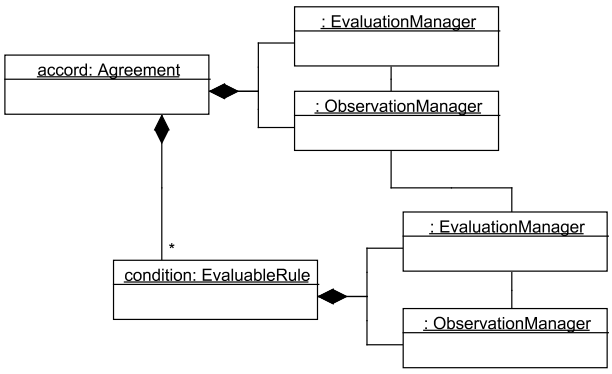


FIGURE 4.20 – Organisation de l'accord

L'accord et les clauses peuvent être évalués à des instants différents pourvu qu'ils le soient pour les mêmes observations. En effet l'assemblage n'est valide que lorsque la conformité et la compatibilité sont simultanément satisfaites.

CCLJ Si nous reprenons l'exemple d'accord précédent, il se présente comme la conjonction de deux règles :

```
On <CruiseCtrl>
  Observe {
    val: level at : entry sns.setSecurityLevel (int level);
}
Verify : (level > 3) => (level < 5)
```

et

```
On <Engine>
  Observe {
    val: retour at : exit getRTime (float throttle);
}
Verify : (retour < 5) => (retour < 10)
```

Nous avons volontairement placé ces règles sur les interfaces serveur car c'est à cet endroit que les observations sont en général les plus simples à réaliser. Par contre, nous pouvons constater qu'un outil d'évaluation statique permettrait d'éviter les observations et l'évaluation récurrente à l'exécution de la seconde règle car elle est trivialement toujours satisfaite. Ainsi si nous disposions de cet outil, il pourrait être intéressant de construire un gestionnaire d'évaluation (`EvaluationManager`) qui, combinant les deux outils d'évaluations, tente à la première évaluation une approche statique puis si elle échoue se reconfigure pour l'évaluer par la suite dynamiquement.

Behavior Protocol L'accord précédent prend la forme de la règle suivante :

```
On <Car>
  Observe {
    val: evt.start at : entry lc.start ()
}
Verify : verticalCheck (CarFrameProtocol,
            horizontalCheck(EngineFrameProtocol,
            CruiseCtrlFrameProtocol))
```

Dans laquelle les variables `CarFrameProtocol`, `EngineFrameProtocol` et `CruiseCtrl-FrameProtocol` sont initialisées avec les spécifications des composants. L'interface `lc` est celle du contrôleur du cycle de vie du composant qui commande le démarrage ou l'arrêt de sa communication avec son environnement. Les fonctions `verticalCheck` et `horizontalCheck` (qui vérifie la compatibilité et retourne la composition parallèle des spécifications) sont déclarées dans l'environnement d'évaluation et appellent un modelchecker dont une référence est passée à l'environnement d'évaluation. Pour ce formalisme l'accord est vérifié de manière statique, le gestionnaire d'observation observe les événements de configuration du système de composant et notifie le gestionnaire d'évaluation juste avant que le composant `<Car>` ne commence à interagir avec son environnement.

4.3.4.3 Contrat complet

Si nous considérons les spécifications en Behavior Protocol des trois composants :

- `<CruiseCtrl>`:

  ```
  CarFrameProtocol : (?sns.on; !csp.setThrottle*; ?sns.off)*
  ```

- `<Engine>`:

  ```
  EngineFrameProtocol : (?csp.setThrottle; !fdk.notifyAction)*
  ```

- `<Car>`:

  ```
  CarFrameProtocol : (?sns.on; !fdk.notifyAction*; ?sns.off)*
  ```

Nous pouvons donner une expression complète du contrat auquel ils doivent satisfaire pour que leur assemblage soit valide (d'après notre définition) :

```
Contract
{
  Participants : <Car>, <Engine>, <CruiseCtrl> ;

  Clause:
  {
   responsible : <Car>;

   guarantee :
     On <Car>
     Observe{
       val : evt.notifyAction at : entry fdk.notifyAction(int code);
       }
```

```
    Verify : runtimeCheck ("(?on;!fdk.notifyAction*;?off)*",
    {"notifyAction"});

  assumption :
    On <Car>
    Observe{
      val : evt.on at : entry sns.on ();
      val : evt.off at : entry sns.off ();
      }
    Verify : runtimeCheck ("(?on;!fdk.notifyAction*;?off)*",
    {"on", "off"});
  }

Clause :
{
 responsible : <Engine>;

 guarantee :
 On <Engine>
    Observe{
      val : evt.notifyAction at : entry fdk.notifyAction(int code);
      }
    Verify : runtimeCheck ("(?csp.setThrottle;!fdk.notifyAction)*",
    {"notifyAction"});

 assumption :
 On <Engine>
    Observe{
      val : evt.setThrottle at : entry csp.setThrottle(float throttle);
      }
    Verify : runtimeCheck ("(?csp.setThrottle;!fdk.notifyAction)*",
    {"setThrottle"});
 }

Clause :
{
    responsible : <CruiseCtrl>;

    guarantee :
    On <CruiseCtrl>
    Observe{
      val : evt.setThrottle at : entry csp.setThrottle(float throttle);
      }
    Verify : runtimeCheck ("(?on;!csp.setThrottle*;?off)*",
    {"setThrottle"});

    assumption :
    On <CruiseCtrl>
    Observe{
      val : evt.on at : entry sns.on ();
      val : evt.off at : entry sns.off ();
      }
    Verify : runtimeCheck ("(?on;!csp.setThrottle*;?off)*",
    {"on", "off"});
 }

Agreement :
{
```

```
    On <Car>
      Observe {
        val: evt.start at : entry lc.start ()
    }
    Verify : verticalCheck (CarFrameProtocol,
              horizontalCheck(EngineFrameProtocol,
                 CruiseCtrlFrameProtocol))
  }
}
```

4.3.4.4 Formalisme compositionnel

Nous ne nous sommes pas jusqu'ici intéressés à la dépendance entre la spécification d'un composant et celles de ses sous composants. Nous allons maintenant considérer des résultats concernant la composition verticale de spécifications de l'approche hypothèse garantie. Ainsi suivant la manière dont une spécification d'un composant dépend des spécifications de ses sous composants nous verrons comment un contrat à un niveau de composition donné peut dépendre d'un contrat entre les sous composants de ce niveau.

Spécification du système

Nous considérons un formalisme de spécification suivant le paradigme hypothèse-garantie. La Table 4.1 fournit des spécifications contractuelles simples s'appliquant aux composants `<CruiseCtrl>` et `<Engine>`. Nous faisons ici l'hypothèse que le système de contrôle de vitesse est actif, c'est à dire que les boutons *on* ou *resume* ont été pressés. Nous supposons par ailleurs que des intervalles de temps peuvent être observés sur le système durant son exécution.
La spécification du `<CruiseCtrl>` garantit alors qu'il émet périodiquement un appel à `setThrottle` sur son interface requise `csp` (soit "periodic(csp.setThrottle)"). La spécification d'`<Engine>` signifie que : si la vitesse cible (TS) est inchangée pendant r_{max} (c'est à dire "cst(TS, r_{max}")), et qu'il reçoit pendant ce temps périodiquement des appels à `setThrottle` sur son interface fournie `csp`, alors il garantit que la vitesse courante atteint la vitesse cible avant que le temps r_{max} soit écoulé (c'est à dire $eq(speed, TS, r_{max})$).

Participant	
<CruiseCtrl> offre	$TRUE \rightarrow periodic(csp.setThrottle)$
<Engine> offre	$periodic(csp.setThrottle) \wedge cst(TS, r_{max}) \rightarrow eq(speed, TS, r_{max})$
Environnement E	$wait(D_1)$ // ne fait rien pendant D_1
<User> requiert	$eq(speed, TS, D_2) \rightarrow TRUE$ // vitesse courante = vitesse cible avant D_2

TABLE 4.1 – Specifications (la notation TS dénote $< CruiseCtrl > .att.targetSpeed$).

Nous pouvons remarquer que les parties hypothèse des spécifications des composants ne peuvent être violée que par leurs environnements respectifs. Par ailleurs la partie garantie de ces spécifications ne peut être violée que par le composant qui la porte, et toutes les propriétés sont des propriétés de sûreté. Il est donc possible d'appliquer le théorème d'Abadi et Lamport pour déduire la spécification du composant `<Car>` de celles de ses sous composants. La mise en oeuvre du théorème suppose une prémisse

qui, si nous ignorons les termes présents de part et d'autres du signe d'implication, peut être écrite de la manière suivante :

$$E \cap M_{<CruiseCtrl>} \cap M_{<Engine>} \subseteq E_{<CruiseCtrl>} \cap E_{<Engine>}, \tag{4.2}$$

$$i.e.: \quad wait(D_1) \cap eq(speed, TS, r_{max}) \subseteq cst(TS, r_{max}). \tag{4.3}$$

Nous pouvons remarquer que :

$$wait(D_1) \subseteq cst(TS, r_{D_1}) \tag{4.4}$$

c'est à dire tant que l'environnement ne modifie pas la vitesse cible TS celle-ci reste constante. Donc nous pouvons réécrire la prémisse :

$$wait(D_1) \cap eq(speed, TS, r_{max}) \subseteq cst(TS, D_1) \cap eq(speed, TS, r_{max}) \tag{4.5}$$

Par ailleurs si $D_1 > r_{max}$ alors :

$$cst(TS, D_1) \subseteq cst(TS, r_{max}) \tag{4.6}$$

ce qui entraine que :

$$wait(D_1) \cap eq(speed, TS, r_{max}) \subseteq cst(TS, r_{max}) \cap eq(speed, TS, r_{max}) \; si \; D_1 > r_{max} \tag{4.7}$$

Mais si l'environnement respecte sa spécification et que $D_1 > r_{max}$ alors $cst(TS, r_{max})$ est vrai. Or, nous remarquons à la lecture de la table 4.1 que $eq(speed, TS, r_{max})$, étant garantie par `<Engine>`, ne peut être violée avant $cst(TS, r_{max})$, l'hypothèse de cette garantie, à moins que `<Engine>` ou `<CruiseCtrl>` (qui garantit $periodic(csp.setThrottle)$) ne violent leurs spécifications, ce que nous écartons pour l'instant. Ainsi si $cst(TS, r_{max})$ est vrai alors $eq(speed, TS, r_{max})$ est vrai. Finalement si l'environnement et les composants `<Engine>` et `<CruiseCtrl>` respectent leurs spécifications et que $D_1 > r_{max}$ alors la prémisse est vraie. Une condition supplémentaire au respect des spécifications par leurs porteurs pour que la prémisse soit vraie est donc :

$$D_1 > r_{max} \tag{4.8}$$

Si la prémisse, qui a, au départ, la forme d'une expression de compatibilité entre les composants `<Engine>`, `<CruiseCtrl>` et leur environnement, est vraie alors le théorème garantit que la conclusion de la règle d'inférence est vérifiée. Ceci implique que la composition de `<Engine>` et `<CruiseCtrl>` implémente la spécification contractuelle suivante $E \rightarrowtriangle M_{<CruiseCtrl>} \cap M_{<Engine>}$, c'est à dire :

$$wait(D_1) \rightarrowtriangle periodic(csp.setThrottle) \cap eq(speed, TS, r_{max}), \tag{4.9}$$

signifiant que tant que l'environnement (c'est à dire l'utilisateur) ne fait rien pendant la durée D_1, le système composé de `<Engine>` et `<CruiseCtrl>` garantit que `setThrottle` est appelé pédioquement de sorte la vitesse courante atteigne la vitesse cible en moins de r_{max}.
Par ailleurs, nous pouvons considérer que l'utilisateur a des attentes par rapport au système, comme le montre la table 4.1, exprimant qu'il est satisfait tant que la vitesse rattrape la vitesse cible en moins de D_2. Cette requête a la même forme que les offres des autres composants, et peut être ainsi inclue dans la composition. Dans ce cas une condition triviale pour que la composition soit valide est que $r_{max} < min(D_1, D_2)$.

Contrats

Contrat simple Nous pouvons mettre en oeuvre un contrat simple pour valider la collaboration des composants `<Engine>` et `<CruiseCtrl>` et de leur environnement qui intervient dans ce formalisme. Le contrat de la figure 4.21 est bâti sur la base des spécifications des deux composants et de leur environnement données dans la table 4.1. L'accord est constitué par une expression qui rend vraie celle de la prémisse du théorème de composition, dont nous avons vu qu'elle rendait compte d'une forme de la compatibilité des composants.

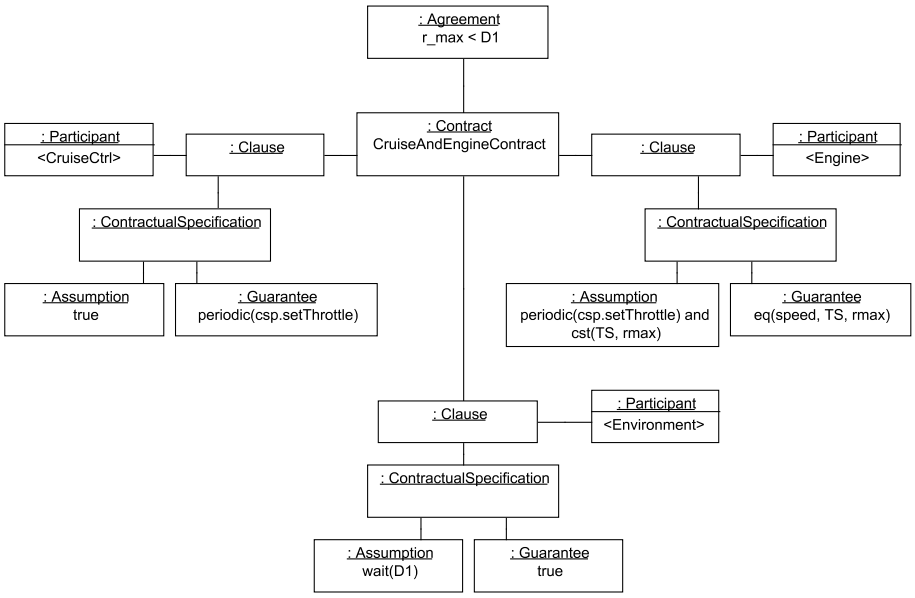


FIGURE 4.21 – Contrat simple

Contrat composite Si nous considérons que la spécification du composant `<Car>` est fournie par le théorème de composition sur la base des spécifications de ses sous composants, alors sa validité dépendra de celle de la règle d'inférence mise en oeuvre par le théorème. Or la prémisse de cette règle d'inférence constitue l'accord du contrat entre les sous composants de `<Car>`. Tout contrat, dans lequel `<Car>` dépendera alors du contrat entre ses sous composants, deviendra ce que nous nommerons un contrat composite. Ceci est dans notre modèle rendu possible par la satisfaction à un certain nombre de propriétés nécessaires à la mise en oeuvre de la règle d'inférence :

- *une garantie doit contraindre son fournisseur et son hypothèse associée doit porter sur l'environnement de ce dernier*. Cette condition fait partie de la définition de notre modèle de contrat,
- *une garantie doit être satisfaite aussi longtemps que son hypothèse l'est.* Cette condition fait partie aussi de la définition de notre modèle de contrat,
- *une spécification composite doit être implémentée par une composition valide de composants (au sens de la prémisse du théorème)*. L'objet `CompositeContractualSpecification` observe la contrat entre les sous composants de son porteur (figure 4.22), pour vérifier que son accord reste valide. Il surveille ainsi les conditions de sa propre validité.

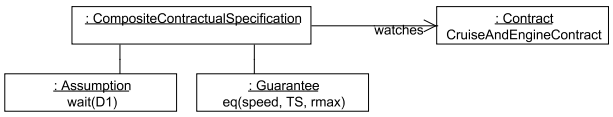


FIGURE 4.22 – Spécification composite

La figure 4.22 donne un schéma objet simplifié de la spécification composite qui peut être associée au composant `<Car>` d'après les spécifications et le théorème que nous avons vu dans la partie précédente. Son hypothèse et sa garantie sont le résultat de la règle d'inférence du théorème, mais nous avons toutefois caché l'événement `setThrottle` qui n'est pas visible de l'extérieur du composant `<Car>`. Elle surveille donc l'accord du contrat entre le `<Engine>` et `<CruiseCtrl>` puisque celui est justement la prémisse de la règle d'inférence.
Comme le composant `<Car>` peut lui même être composé avec le composant `<User>` il y a lieu d'établir un contrat qui garantisse la validité de cet assemblage (figure 4.23). Il serait possible d'appliquer à nouveau le théorème pour en déduire une spécification contractuelle de cette composition etc...

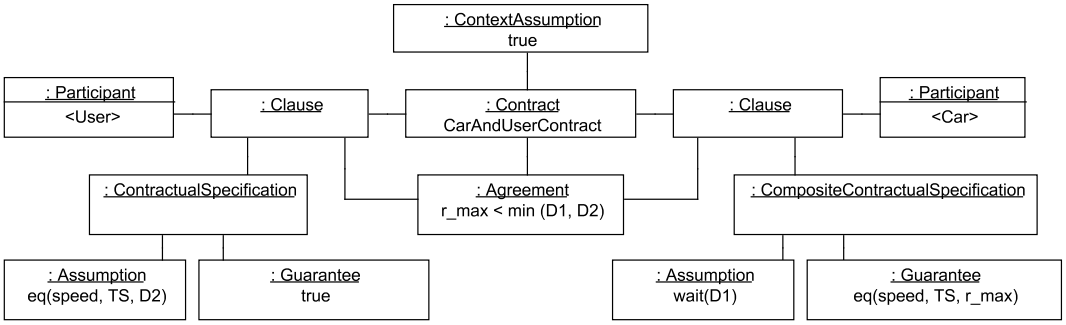


FIGURE 4.23 – Contrat entre `<Car>` et `<User>`

4.4 Conclusion

Nous avons montré dans cette partie comment il était possible de modéliser de manière générique un certain type d'architectures et de formalismes. Nous avons exposé ainsi comment il était alors possible de décrire les systèmes à base de composants Fractal sans utiliser les spécificités de ce modèle de composant. Il en va de même pour la plupart des systèmes à base d'objets de composants ou de services, dont l'architecture relève du modèle objet - relation. Nous avons aussi présenté comment exprimer des spécifications de sorte que seuls leurs traits nécessaires au modèle de contrat soient réifiés. Une illustration de cette approche sur les formalismes CCLJ et Behavior Protocol a été par la suite donnée. Puis la mécanisation de l'interprétation des spécifications en couple d'hypothèse et garantie a été exposée sur la base de la description générique de l'architecture et des observations. Nous avons enfin montré comment les contrats pouvaient alors être batis et évalués sur la base de ces modélisations génériques. Pour illustrer ceci nous avons fourni trois exemples de contrat respectivement sur des spécifications CCLJ, Behavior Protocol, et un formalisme générique correspondant au paradigme hypothèse-garantie.

CHAPITRE 5

Validation

5.1 Introduction

Dans les deux précédents chapitres nous avons présenté les principes de fonctionnement et de mise en oeuvre de notre modèle de contrat. Il nous faut maintenant envisager sa réalisation concrète. Dans ce chapitre nous allonc donc envisager l'application de notre modèle à une application réelle (5.2.1 et 5.2.3). A cette occasion nous pourrons décrire comment les différents intervenants dans la mise en oeuvre et l'exploitation de l'application, tirerons partie de son utilisation (5.2.2). Nous présenterons par la suite notre conception d'implémentation de son prototype (5.3). Finalement nous tirerons un bilan qui discute de notre solution par rapport à l'état de l'art, tout en présentant aussi ses avantages et limites intrinsèques (5.4).

5.2 Cas d'étude

5.2.1 Contexte de l'exemple

Pour ce cas d'étude nous allons considérer le contexte fonctionnel offert par un système d'information "éphémère". Ce dernier terme désigne un système d'information mis en oeuvre temporairement à l'intention d'un rassemblement de personnes limité dans le temps. Par exemple si nous considérions le salon de l'automobile qui est très vaste, les exposants pourraient souhaiter offrir différents services d'information aux visiteurs en utilisant le téléphone portable de chacun d'entre eux comme support de communication. Ainsi les exposants pourraient souhaiter :

- envoyer des publicités aux visiteurs par sms ou mms, des rappels d'événements prévus sur les stands, etc..
- faire participer les visiteurs à des jeux concours où la question est envoyée par sms ou mms et la réponse retournée par le même mode (éventuellement gratuit), avec des lots à retirer sur les stands etc...,
- faire en sorte que un visiteur puisse obtenir une liste des stands, proches d'où il se trouve, susceptibles de l'intéresser sur la base de l'envoi d'un sms contenant l'identifiant du stand sur lequel il se trouve,

Immédiatement il apparaît que tous les exposants ne vont pas souhaiter ni pouvoir s'adresser à tous les visiteurs, sous peine de surcharger les téléphones et de lasser l'attention des visiteurs. Il faut un système capable de mettre en relation exposants et visiteurs sur la base de l'identification et du rapprochement des offres des exposants (produits) et des attentes des visiteurs (centres d'intérêts).

5.2.2 Motivation de l'utilisation des contrats

Pour l'administrateur. Du point de vue de l'administrateur de l'application, l'intérêt de mettre en oeuvre des contrats est multiple. En effet nous pouvons concevoir que ce service de communication offert aux exposants est payant. Il faut donc s'assurer qu'il fonctionne en permanence correctement, car en cas d'interruption de ce dernier les exposants pourraient par exemple réclamer des pénalités. Dans certains cas il faut aussi pouvoir prouver aux clients que le service fonctionne bien. Enfin il faut limiter dans le temps toute interruption du service. Or il se trouve que cette application est destinée à avoir de nombreux utilisateurs qui n'ont quasiment pas la possibilité de détecter ni faire remonter un possible dysfonctionnement :

- un utilisateur ne recevant pas de message ne sait pas si il aurait du en recevoir un,
- un utilisateur dont l'inscription sur une borne ne se déroule pas correctement ne pourra peut être pas trouver une personne à son écoute pour faire remonter l'incident, sans même parler de la description de ce dernier etc...
- un utilisateur demandant les stands l'intéressant autour de celui où il se trouve ne saura pas si certains ont été oubliés dans la réponse, si il ne reçoit pas de réponse il ne saura sans doute pas non plus à qui s'adresser etc...
- les exposants n'ont pas de moyen de vérifier que leurs messages sont bien émis vers tous les utilisateurs susceptibles d'être intéressés etc...

Il est donc nécessaire d'utiliser un système assurant du bon fonctionnement de l'application depuis son intérieur. Or les interactions entre les composants constituent le fonctionnement même de l'application. Comme les contrats surveillent ces interactions (via la conformité des composants à leurs spécifications et la compatibilité de celles-ci), ils constituent donc un moyen pragmatique de garantir que l'application fonctionne normalement. De plus les contrats sont explicites, c'est-à-dire, que les propriétés qu'ils garantissent sont humainement lisibles, ce qui permet d'exprimer et d'évaluer la garantie qu'ils fournissent.

Il importe aussi que l'éventuelle interruption du service soit aussi brève que possible et pour ce faire il est nécessaire d'avoir un diagnostic précis de la panne. A nouveau, les contrats permettent d'identifier non seulement l'élément à l'origine de l'erreur mais aussi la nature de celle-ci : problème de conformité ou de compatibilité. Par ailleurs, le diagnostic présente une description complète du contexte de l'erreur facilitant d'autant la réparation (remplacement du composant fautif par une version plus ancienne mais plus sure, nouveau composant, changement du paramétrage etc...).

Pour le fournisseur de formalisme. Du point de vue du concepteur de formalisme, notre approche permet l'utilisation d'un formalisme en dehors d'une intégration adhoc et exclusive de ce dernier à une unique architecture. Il ne s'agit plus en effet d'adapter l'architecture à la mise en oeuvre d'un formalisme mais d'autoriser l'intégration de celui-ci à une modélisation donnée de l'architecture. Or comme cette modélisation est destinée à être commune à différentes architectures concrètes, notre approche des contrats facilite ainsi l'application d'un formalisme dans différents contextes. Cela permet de valider un formalisme sur des architectures aux propriétés variées. En l'occurrence,

un formalisme développé pour notre architecture pourra être réutilisé sur d'autres.

Pour le fournisseur des spécifications. Notre approche des contrats autorise la mise en oeuvre de différents formalismes sur un même système. Cela permet de vérifier les assemblages sur la base des différentes propriétés qui les caractérisent et qui doivent être concommitemment satisfaites pour que l'assemblage soit valide. De plus notre approche fournit un cadre d'interprétation commun aux différents formalismes sur la base de la sémantique hypothèse-garantie. Le fournisseur des spécifications peut ainsi être assuré que même si l'administrateur n'est pas un expert des différents formalismes il pourra avoir une première lecture du diagnostic d'erreur en termes de responsabilité et d'échanges. L'administrateur saura à qui s'adresser pour régler le problème car son origine sera clairement déterminée. Ainsi dans notre exemple nous allons appliquer deux formalismes de spécification à notre système. Les Behavior Protocol vont garantir les ordres d'appels des méthodes, tandis que les assertions CCLJ vont s'assurer que les grandeurs échangées sont satisfaisantes, mais pour autant, la validité de l'architecture sera basée et interprétée sur celle de ses échanges.

Pour l'assembleur d'application. L'assembleur de l'application peut mettre en oeuvre des applications complexes sans se préoccuper de la possibilité de les spécifier de manière globale et extérieure pour s'assurer de leur bon fonctionnement. En effet comme nous avons vu sur notre simple exemple, les spécifications externes d'une application peuvent se révéler aussi complexes que l'application elle même. Il suffit d'observer dans notre cas la difficulté qu'auraient les utilisateurs à distinguer un dysfonctionnement. Ainsi quand les entrées de l'application interagissent de manière complexes pour fournir ses sorties, la spécification de ces dernières se révèle problématique comme c'est le cas dans notre exemple. Or le fonctionnement de ces applications étant basé sur les interactions de leurs composantes, garantir ces interactions via les contrats revient à garantir le bon fonctionnement de l'application. C'est ce que nous allons effectuer pour notre application exemple.

Pour l'intégrateur d'architecture. L'intégrateur d'un modèle concret d'architecture (framework de composants, de services etc...) est assuré que les systèmes bâtis sur son modèle seront évalués sur la base des principes de séparation des responsabilité, modularité, collaboration compatibles avec son modèle. Il aussi assuré que même sans une connaissance précise de l'implémentation du modèle, l'administrateur disposera en première lecture d'un diagnostic lui permettant de réagir par rapport à une architecture donnée. Par ailleurs cette intégration autorise l'application de formalismes variés sans que leurs concepteurs aient une grande connaissance de l'implémentation de l'architecture. Par exemple dans notre application, la mise en oeuvre du formalisme des Behavior Protocol ne fera pas appel à d'autres connaissances que celles utilisées pour mettre en oeuvre CCLJ.

5.2.3 Le système Amui

5.2.3.1 Présentation

Amui est un framework dont la principale fonctionnalité consiste à former automatiquement des groupes d'utilisateurs et à leur faire partager diverses applications en fonction de leurs centres d'intérêts. Il se présente sous la forme d'un service web implé-

menté à l'aide de la librairie Axis et hébergé sous un serveur apache. Ce service fournit deux ensembles d'opérations, l'un pour l'enregistrement et la gestion des utilisateurs et des groupes, un autre pour l'administration des applications partagées par les utilisateurs. Le serveur héberge en plus les pages internet donnant accès à ces opérations. Ainsi un utilisateur peut se connecter au système à l'aide d'un simple navigateur et fournir les informations le concernant (dans notre cas son nom, numéro de portable, ...) ainsi que la liste des mots clés décrivant ses centres d'intérêts. Le système trouve alors automatiquement les groupes dont les thèmes correspondent aux centres d'intérêts de l'utilisateur et y ajoute ce dernier. De cette manière à l'intérieur d'un même groupe, les utilisateurs peuvent alors partager diverses applications qui correspondent aux thématiques du groupe et qui sont adaptées à leurs centres d'intérêts. Dans cette optique les opérations d'administration des applications permettent symétriquement de spécifier les thèmes auxquels ces dernières sont associées. Dans notre cas de figure, un exposant peut via un navigateur définir les thèmes des messages, concours etc... qu'il souhaite diffuser, ainsi que les thèmes auxquels il souhaite voir son stand associé.

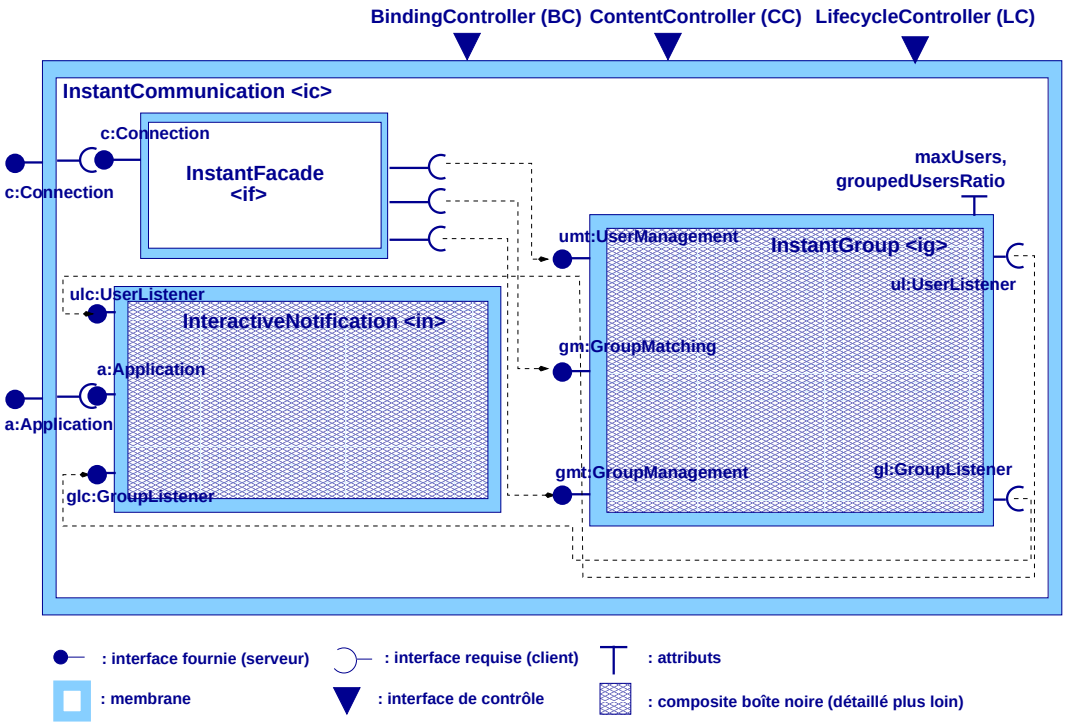


FIGURE 5.1 – Architecture de la plate-forme Amui

5.2.3.2 Mise en oeuvre

Architecture générale

L'architecture générale de l'application est décrite à la figure 5.1. Le composant principal `<InstantCommunication>` représente l'application. D'un point de vue externe, il offre l'interface Fractal `c : Connection` de nom `c` et de signature `Connection`[1]. La méthode `connectUser()` de cette interface représente le point d'entrée de l'application en permettant de connecter un utilisateur au système en lui fournissant certaines

[1]Dans le cas de julia, cette signature peut être assimilée à une interface java.

```
interface Connection{
int connectUser(String username, String password, Collection<String> keywords);
}

interface GroupManagement{
Collection<String> getGroupsId();
Collection<String> getGroupTopics(String groupid);
int addUserToGroups(User u, Collection<String> groupsid);
...
}

interface GroupMatching{
Collection<String> searchGroupForKeywords(Collection<String> keywords);
...
}

interface UserManagement{
User createUser(String username,  String password);
int registerUser(User u);
User getUser(String username);
...
}

interface Application{
void registerAdvertisement(String message, Collection<String> topics,
        Schedule schedule);
void registerContest(String question,  Collection<String> topics,
        Schedule schedule);
void registerStand(int standNumber, Collection<String> topics);
}
```

informations (login, mot de passe, centres d'intérêts). L'architecture interne de l'application (cf. figure 5.1) est composée des trois sous-composants suivants :

- le composant `<InstantFacade>` a en charge le pilotage de l'ensemble des fonctionnalités de gestion des utilisateurs et des groupes. Il offre le service de connexion au système au travers de la méthode `connectUser()` de l'interface `c : Connection`, et requiert les interfaces de`InstantGroup` de la façon suivante : *(i)* `umt : UserManagement` pour créer et enregistrer les utilisateurs, `(ii) gm : GroupMatching` pour obtenir la liste des identifiants des groupes dans lesquels ajouter les utilisateurs, suite à l'appariement entre les centres d'intérêts des utilisateurs et les thèmes des groupes, et *(iii)* `gmt : GroupManagement` pour demander l'ajout des utilisateurs dans des groupes donnés.
- le composite `<InstantGroup>` permet de gérer tout ce qui concerne les utilisateurs et les groupes. D'un point de vue externe, il offre trois interfaces `umt : UserManagement`, `gm : GroupMatching` et `gmt : GroupManagement` pour, respectivement, gérer les utilisateurs, effectuer l'appariement entre les centres d'intérêts des utilisateurs et les thèmes des groupes, et gérer les groupes. Les interfaces requises `un : UserListener` et `gn : GroupListener` servent à notifier le composant `<InteractiveNotification>` des événements pertinents sur les utilisateurs et les groupes, selon un patron de conception *Observer*[2]. L'architecture interne de ce composant est décrite ci-après.
- le composant `<InteractiveNotification>` gère les interactions entre les services des exposants et les utilisateurs, sur la base de l'appartenance des seconds à des groupes d'intérêts. Il permet aux exposants de configurer leurs services et de les associer à des groupes sur la base de leurs centres d'intérêts. Ce composant est plus précisément décrit dans la suite.

`<InteractiveNotification>` : le système de communication

Dans notre cas de figure le système de communication avec les utilisateurs est représenté par le composant `<InteractiveNotification>`. Il propose aux exposants d'associer leurs communications à des mots clés. Ces communications concerneront alors les groupes dont les centres d'intérêts recouvrent leurs mots clés. Ainsi un exposant peut auprès de ce composant à l'aide de l'interface `Application` :

1. enregistrer un message à diffuser (`registerAdvertisement`), suivant un timing donné (`Schedule`), associé à des thèmes donnés,
2. enregistrer une question à diffuser, à un horaire donné (`registerContest`), associée à des thèmes donnés et enregistrer la période durant laquelle les réponses à la question lui seront remontées (`Schedule`),
3. associer l'emplacement de son stand (numéro de stand) à des thèmes (`registerStand`),

Ce composant est notifié par le composant `<InstantGroup>` des créations et suppressions de groupes et d'utilisateurs via les interfaces `UserListener` et `GroupListener`. Il maintient sa propre liste des groupes et des utilisateurs car d'une part diverses informations d'identification personnelle des utilisateurs ne lui sont pas transmises, et ainsi il garde le contrôle (évolution dynamique etc...) de sa source de données.

[2] L'inscription des écouteurs se fait lorsque les composants se connectent à travers le *BindingController*.

- pour les services 1 et 2 : pour chaque message le `<InteractiveNotification>` sélectionne les groupes dont les thèmes comprennent ceux du message, puis suivant le timing prévu, il envoie ce messages aux utilisateurs appartenant à ces groupes,

- pour le service 3 : `<InteractiveNotification>` sélectionne les groupes de l'utilisateur demandeur et les compare aux thèmes des stands qui environnent celui auquel se trouve l'utilisateur,

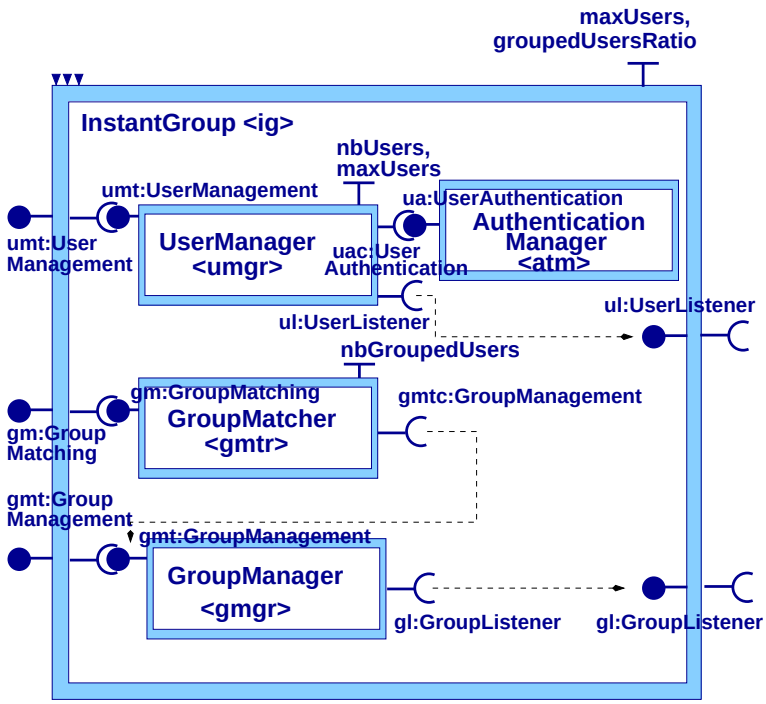


FIGURE 5.2 – Architecture interne du composant `<InstantGroup>`

`<InstantGroup>` : la gestion des groupes

L'intérieur du composite`<InstantGroup>` est formé de quatre sous-composants avec les principales fonctionnalités fournies et requises suivantes (cf. figure 5.2) :

- `<UserManager>` prend en charge la gestion des utilisateurs (création, enregistrement, etc.). Au travers de son interface fournie `umt : UserManagement`, il permet notamment de créer de nouveaux utilisateurs (`createUser()`), d'enregistrer des utilisateurs existants dans le système (`registerUser()`), de récupérer des utilisateurs crées (`getUser()`). Ce composant, au travers de son interface requise `uac : UserAuthentication`, utilise des services d'authentification des utilisateurs. Il émet aussi des événements liés aux utilisateurs par son interface requise `un:UserListener`.

- `<AuthenticationManager>` apporte le service technique qui permet d'authentifier des utilisateurs. La méthode `checkAuthentication()` de l'interface `ua : UserAuthentication` permet de vérifier la validité d'un utilisateur à partir d'un couple (login,mot de passe).

```
interface UserManagement{
User createUser(String username, String password);
int registerUser(User u);
User getUser(String username);
...
}

interface UserAuthentification{
boolean checkAuthentication(String username, String password);
...
}

interface GroupManagement{
 Collection<String> getGroupsId();
 Collection<String> getGroupTopics(String groupid);
 int addUserToGroups(User u, Collection<String> groupsid);
...
}

interface GroupMatching{
Collection<String> searchGroupForKeywords(Collection<String> keywords);
...
}
```

- `<GroupManager>` prend en charge la gestion des groupes (création, abonnement d'utilisateurs, etc.). Son interface fournie `gmt : GroupManagement` permet de récupérer les identifiants de tous les groupes (`getGroupsId()`), de récupérer les thèmes d'un groupe donné (`getGroupTopics()`), et d'ajouter un utilisateur dans un groupe (`addUserToGroups()`). De plus, il émet des événements liés aux groupes par son interface requise`gn:GroupListener`.

- `<GroupMatcher>` permet de faire le lien entre les mots-clés fournis par les utilisateurs et les groupes qui y correspondent. L'interface fournie `gm : GroupMatching` comporte notamment la méthode`searchGroupsForKeywords()` qui effectue l'appariement entre les centres d'intérêts fournis par les utilisateurs (*keywords*) et les thèmes des groupes (*topics*). À partir des mots-clés fournis par les utilisateurs, il détermine et rend les identifiants des groupes auxquels il faudrait les ajouter. La dépendance aux services liés aux groupes est exprimée par l'interface requise `gmtc : GroupManagement`, connectée à l'interface fournie `gmt : GroupManagement` de `<GroupManager>`.

5.2.4 La mise en oeuvre des contrats

5.2.4.1 Spécifications

Nous allons dans cette partie considérer deux ensembles de spécifications exprimées dans deux formalismes différents :

Behavior Protocol

Les Behavior Protocol vont nous permettre de décrire le comportement des échanges de messages entre les composants `<Facade>`, `<InstantGroup>`,

`<InteractiveNotification>`, afin de nous assurer qu'ils sont bien compatibles et respectent bien leurs protocoles. Une intégration des Behavior Protocol à la plate-forme Fractal, déjà réalisée de manière adhoc, et nous assure qu'ils sont bien applicables à celle-ci. Nous considérons donc les spécifications suivantes :

- `<Facade>`:
  (1)

```
(?c.connectUser;(!umt.startUserCreation; !umt.createUser;
  !gm.searchGroup;!gmt.addUserToGroup*;
  !umt.closeUserCreation))*
+
(?c.disconnectUser;(!umt.startUserSuppression;
  !gmt.getUserGroups;!gmt.removeUserFromGroup*;!umt.deleteUser;
  !umt.CloseUserSuppression))*
```

  Cette spécification décrit les interactions entre `<Facade>` et `<InstantGroup>` dans le cas où un utilisateur est ajouté (se connecte) ou est supprimé (se déconnecte) du système. Les procédures de création et de suppression se font dans des sessions encadrées de commandes `start` et `close` afin de permettre une approche transactionnelle de ces opérations, même si le rollback n'est pas mis en oeuvre ici.

- `<InstantGroup>`:
  (2)

```
(?umt.startUserCreation;?umt.createUser;
  !ul.notifyUserCreation;?gm.searchGroup;
  (?addUserToGroup;!gl.notifyUserAddedToGroup)*;
   ?umt.closeUserCreation)*
+
(?umt.startUserSuppression;?gmt.getUserGroups;
  (?gmt.removeUserFromGroup;!gl.notifyUserRemovedFromGroup)*;
    (?umt.deleteUser;?umt.CloseUserSuppression;
      !ul.notifyUserSuppression))*
```

  Cette spécification présente les sessions de création et suppression du point de vue du composant `<InstantGroup>`. Nous ne présentons pas, pour simplifier les spécifications, les notifications que devrait émettre ce composant vers `<InteractiveNotification>` en cas d'interruption d'une de ces manoeuvres pour le ramener dans un état cohérent.

- `<InteractiveNotification>`:
  (3)

```
(?ul.notifyUserCreation + ?ul.notifyUserSuppression)*
|
(?gl.notifyUserAddedToGroup+?gl.notifyUserRemoveFromGroup)*
|
```

```
(?a.registerAdvertisement + ?a.registerContest +
 ?a.registerStand)*
```

Le composant <InteractiveNotification> autorise l'ajout de messages, etc... pendant la présence d'utilisateurs, ce qui lui permet de s'adapter dynamiquement aux événements imprévus du salon.

Le respect de ces spécifications assure que tout utilisateur qui est créé est bien affecté aux groupes qui le concernent et que inversement à sa suppression il en complètement retiré. Leur compatibilité assure que l'ajout ou la suppression réussi(e) de tout utilisateur est notifié(e) au composant <InteractiveNotification>.

CCLJ

L'application InstantCommunication peut accepter simultanément plusieurs connexions d'utilisateurs souhaitant s'enregistrer auprès du système. La méthode getMaxConnections de l'interface Connection de ce composant retourne le nombre maximum de connexion pouvant être ouverte simultanément. A l'intérieur du système, comme le composant <InstantGroup> a nécessairement des ressources limitées, il utilise une file d'attente pour stocker les demandes d'enregistrement qu'il doit traiter. La méthode registrationQueueSize retourne le nombre d'enregistrements en attente. Nous décrivons cela à l'aide des assertions suivantes :

```
on <InstantFacade>
 context void umt.addUserTab(List> users)
  pre:
   users.size()<c.getMaxConnections();

on <InstantGroup>
 context void umt.addUserTab(List users)
  pre:
   users.size()<20-umt.registrationQueueSize();
```

La première précondition exprime que la taille de la liste de demandes d'enregistrement que <InstantFacade> passe au <InstantGroup> en appelant addUserTab, est inférieure que le nombre de connexions d'utilisateur que l'<InstantFacade> peut accepter simultanément. La seconde précondition contraint le nombre de demande d'enregistrement que le <InstantGroup> reçoit de sorte que la taille finale de la file d'attente soit inférieure à 20. Il est de plus intéressant de constater que la première précondition met en oeuvre une caractéristique de CCLJ qui est de considérer les composants dans leur ensemble. En effet elle fait intervenir l'interface c dans le cadre de la spécification de l'interface umt.

5.2.4.2 Les contrats

Contrat basé sur les Behavior Protocol

Nous considérons miantenant le contrat généré par notre système pour le motif d'architecture de la figure 3.12 de la partie 3.4.3, ayant pour participants <Facade>, <InstantGroup> et <InteractiveNotification>. Nous pouvons tout d'abord remarquer que, le contrat portant sur les échanges entre composants, les observations

qui sont l'objet d'une hypothèse dans une clause sont l'objet d'une garantie dans une autre (sauf pour l'hypothèse de la première clause qui repose sur l'environnement du système). Ceci est illustré dans l'extrait de contrat qui suit, dans lequel les interceptions de méthode de la garantie d'une clause, sont utilisées dans l'hypothèse de celle qui suit (cf les blocs `Observe`). Cette remarque montre l'importance de l'accord chargé d'-exprimer la compatibilité des clauses, c'est à dire celles des propriétés fournies par des participants et celles requises par les autres, car elles portent sur les mêmes grandeurs.

```
Contract
{
  ...
  Clause :
  {
   responsible : <InstantGroup>;

   assumption :
  ...

   guarantee :
   On <InstantGroup>
     Observe{
       val : evt.notifyUserCreation
               at : entry ul.notifyUserCreation(...);
       val : evt.notifyUserAddedToGroup at : entry gl.
           notifyUserAddedToGroup(...);
       ...
     Verify : runtimeCheck ( (2),
     {"notifyUserCreation", "notifyUserAddedToGroup",
       "notifyUserSuppression", "notifyUserRemoveFromGroup"});
   }

  Clause :
  {
     responsible : <InteractiveNotification>;

     assumption :
     On <InteractiveNotification>
     Observe{
       val : evt.notifyUserCreation
               at : entry ul.notifyUserCreation(...);
       val : evt.notifyUserAddedToGroup at : entry gl.
           notifyUserAddedToGroup(...);
        ...
       }
     Verify : runtimeCheck ( (3),
     {"notifyUserCreation", "notifyUserAddedToGroup",
       "notifyUserSuppression", "notifyUserRemoveFromGroup",
       "registerAdvertisement","registerContest", "registerStand"});
   }
   ...
 }
```

En rentrant plus dans le détail nous pouvons constater que l'accord est vérifié statiquement (`parallelCheck`) au démarrage du composant `<InstantFacade>`.

```
Contract
{
  ...
```

```
  Agreement :
  {
    On <InstantFacade>
      Observe {
        val: evt.start at : entry lc.start ()
    }
    Verify : parallelCheck({ (1), (2), (3) });
  }
}
```

L'accord et les clauses peuvent être évalués à des instants différents comme le montrent les clauses qui, elles, vérifient que les composants respectent leurs spécifications tout au long de l'exécution (`runtimeCheck`). L'important est que les clauses et l'accord soient évalués sur les mêmes grandeurs, pour que la combinaison de leur évaluation dans le cadre du contrat fasse sens. Or par définition l'objet même de l'accord est de vérifier la compatibilité des clauses vis à vis des grandeurs, échangées par leur porteur, qu'elles contraignent. L'accord contraint donc le système sur les mêmes grandeurs que les clauses, même si ils ne sont pas évalués en même temps. Dans notre exemple l'accord est vérifié statiquement par modelchecking, pour l'ensemble exhaustif des suites d'appels de méthode possibles entre composants. Les clauses par contre sont évaluées pour les interceptions des appels de méthodes qui sont réellement effectués à l'exécution mais qui sont donc inclus dans l'ensemble des suites d'appels contraints par l'accord (si les composants respectent leurs clauses). Ainsi l'évaluation du contrat n'est donc pertinente que sur les appels contraints par les clauses, car c'est seulement sur ceux-ci que le contrat peut combiner évaluation des clauses et de l'accord. Mais nous pouvons alors considérer que l'évaluation courante du contrat est recevable tant qu'aucune nouvelle observation ne vient la remettre en cause.

Enfin nous pouvons noter que la clause du composant `<InteractiveNotification>` (décrite plus haut) ne comporte qu'une hypothèse. Elle pourrait donc sembler inutile car elle ne permet pas de vérifier que ce composant se comporte conformément à sa spécification. Toutefois elle est importante pour la validation de l'assemblage car elle exprime ce que ce composant attend des autres pour fonctionner correctement. Cette donnée est ainsi intégrée dans l'expression de l'accord, qui se trouve remis en cause si les garanties des autres composants ne satisfont pas à cette hypothèse.

Contrat basé sur les spécifications CCLJ

Nous considérons ici le contrat généré pour le motif client-serveur dont les composants `<InstantFacade>` et `<InstantGroup>` ainsi que leur connexion forment une instance. Dans ce contrat comme dans le précédent nous pouvons constater que la grandeur (`users`) échangée par les deux composants (`<InstantFacade>` et `<InstantGroup>`) est l'objet d'un couple de clauses. En observant chacun des blocs `Observe`, nous voyons que l'une apporte une garantie sur la grandeur, l'autre pose dessus une hypothèse.

```
Clause :
    responsible : <InstantFacade>
    guarantee : rule {
      On <InstantFacade>
        Observe : val : users at entry umt.addUserTab(List users);
        Verify : users.size()<c.getMaxConnections();
    }
```

```
Clause :
  responsible : <InstantGroup>
  assumption : rule {
    On <InstantGroup>
      Observe : val : users at entry umt.addUserTab(List users);
      Verify : users.size()<20-umt.registrationQueueSize();
  }
```

Par ailleurs pour cet échange l'accord comporte une expression exprimant la compatibilité de son couple de clauses :

```
Agreement :
  On <InstantGroup>
  Observe :
    val : users at entry umt.addUserTab(List users);
  Verify :
    users.size()<c.getMaxConnections() =>
      users.size()<20-umt.registrationQueueSize();
```

Ainsi pour la grandeur échangée, le contrat évalue les deux clauses, reflétant la conformité des parties (`<InstantFacade>` et `<InstantGroup>`), et l'expression de l'accord, reflétant la compatibilité des parties pour cette valeur. Comme précédemment ces évaluations portent sur une même grandeur, elles peuvent donc être combinées pour fournir l'état du contrat et donc celui de l'assemblage au moment où l'échange a lieu.

5.2.5 Conclusion

Nous avons montré comment à l'aide d'un même outil il est possible d'envisager la validité d'un assemblage de composants pour différentes natures de propriétés et sur la base des interactions qu'il abrite. L'unicité de l'outil est fondamentale car elle permet d'interpréter de la même manière des erreurs issues de propriétés différentes. Une gestion uniforme du système, qui permet de réduire le niveau requis d'expertise des propriétés pour administrer le système, devient alors possible. Par ailleurs comme il est ainsi possible de considérer différentes propriétés, il devient possible de prendre en compte simultanément différentes dimensions du fonctionnement d'un système. Dans notre exemple nous nous sommes focalisés sur l'expression du comportement du système à l'aide de deux formalismes différents contraignant pour l'un, l'ordre des échanges, pour l'autre, les valeurs échangées. Nous aurions aussi pu considérer des paramètre de QoS que nous aurions pu contraindre au sein d'assertions (par exemple). Il nous faut toutefois noter que parmi ces propriétés nous aurions considéré celles qui caractérisent les interactions (par exemple fréquence d'appels d'une méthode etc...). En effet notre approche ne s'applique qu'aux propriétés qui s'articulent autour des interactions entre les parties du système. Mais loin d'être restrictif, le fait de se concentrer sur ces interactions internes ne fait que suivre la tendance du génie logiciel à considérer les programmes comme des assemblages, d'objets, de composants ou de services. Cette dernière propriété peut finalement se révéler incontournable comme nous avons vu dans notre exemple, car il n'est pas toujours possible de surveiller les interactions d'un système complexe avec son environnement, et surtout de distinguer, parmi celles-ci, celles qui sont valides de celles qui ne le sont pas.

5.3 Implémentation

Dans cette partie nous allons aborder les différents éléments structurant l'implémentation de la gestion des contrats et celle du système dans son ensemble. Puis nous exposerons les différents mécanismes à spécialiser pour permettre au système de contrat de prendre en compte différentes architectures et formalismes de spécification, ainsi que leurs implémentations dans des cas concrets.

5.3.1 Gestion des contrats

5.3.1.1 Principe du `ContractManager`

Le rôle du `ContractManager` est de déclencher la création ou la suppression d'un contrat qu'il possède alors. Chacun de ces gestionnaires est associé à un ensemble donné d'éléments d'architecture et prend en charge les contrats dans lesquels ils interviennent. Inversement chaque élément d'architecture n'est associé qu'à un seul gestionnaire de contrat (bien qu'il puisse participer à plusieurs contrats). Classiquement chacun de ces ensembles d'éléments d'architecture est défini par l'ensemble des éléments enceints dans une frontière de visibilité, par exemple la limite d'un composant composite, ou celle d'une orchestration, car aucune relation de motif d'architecture ne traverse cette limite. Dans les systèmes non hiérarchiques, sans limite de visibilité, un seul gestionnaire a en charge l'ensemble des éléments d'architecture. De manière générique chaque `Reference` à un élément d'architecture donne accès au `ContractManager` qui gère les contrats auxquels il peut appartenir. Dans le cas des systèmes hiérarchiques, l'objet `Reference` désignant un élément d'architecture composite donne aussi accès au gestionnaire de contrats des éléments qu'il contient. Les portées des différents `ContractManager` sont illustrées sur la figure 5.3.

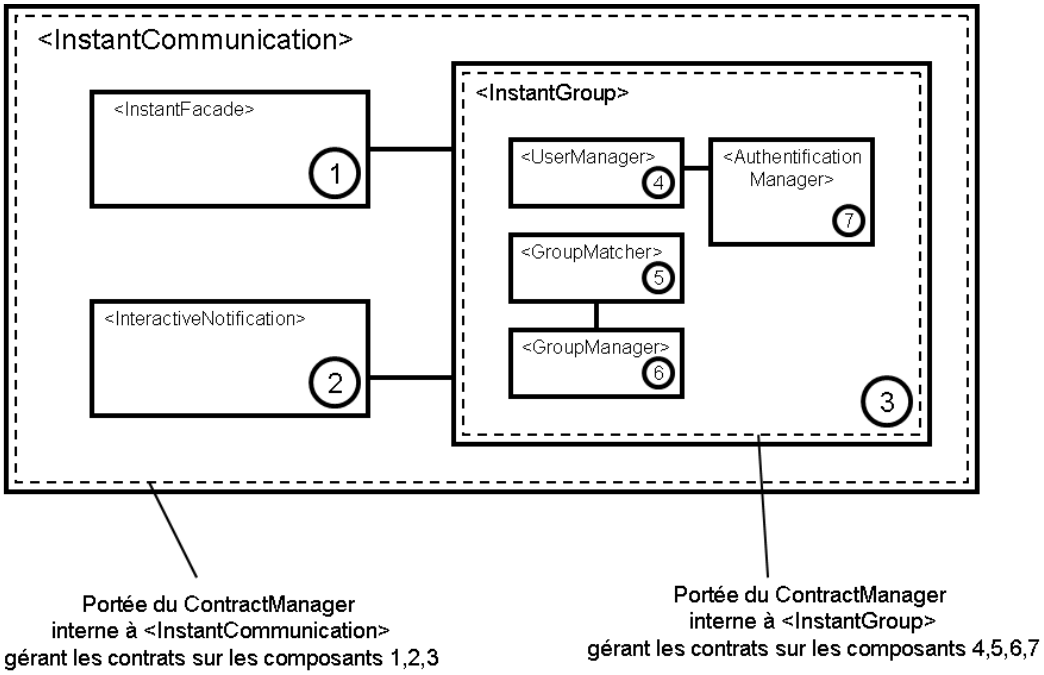


FIGURE 5.3 – Portées des `ContractManager`

5.3.1.2 Fonctionnement du `ContractManager`

Le `ContractManager` peut fonctionner de deux manières : une "statique" et une "dynamique". Dans les deux cas chaque type de contrat référencé par le `ContractManager` est associé à un motif architectural. Pour chacun, le `ContractManager` construit et s'abonne à un `PatternInstanceManager` chargé d'observer la présence, l'apparition, la disparition de ce motif dans l'architecture. Dans le cas du mode "statique", le `PatternInstanceManager` parcourt l'architecture et chaque fois qu'une instance de motif est détectée (au sein de l'ensemble d'éléments surveillés par le `ContractManager`), le type de contrat qui lui est associé est utilisé pour instancier un exemplaire du contrat. Les participants sont les noeuds du motif d'architecture, sachant que celui-ci peut être incomplet au moment de la création du contrat et augmenté au fur et à mesure de la configuration de l'architecture. Le mode"dynamique" consiste pour le `ContractManager` à suivre l'évolution de l'architecture. Dans ce cas, le `PatternInstanceManager` observe l'apparition et la disparition d'instance de motif d'architecture et le `ContractManager` crée ou supprime des contrats en conséquence. A nouveau les instances de motif d'architecture peuvent être incomplètes et les contrats correspondant augmentés avec l'avancement de la configuration de l'architecture. Le `ContractManager` référence pour son fonctionnement différents objets comme illustré sur la figure 5.4 :

- `ContractType` : il détient le motif d'architecture associé à ce type de contrat, par ailleurs non seulement il fournit la factory du contrat, il définit aussi les spécifications sur lesquelles les clauses et l'accord du contrat sont bâtis,
- `ArchitectureDescription` : il fournit les méthodes d'exploration de l'architecture et d'observations de son évolution,
- `PatternInstanceManager` : initialisé pour un motif d'architecture donné, cet objet observe l'architecture à l'aide du `ArchitectureDescription` pour y détecter la présence, l'apparition ou la disparition d'instance de ce motif d'architecture,

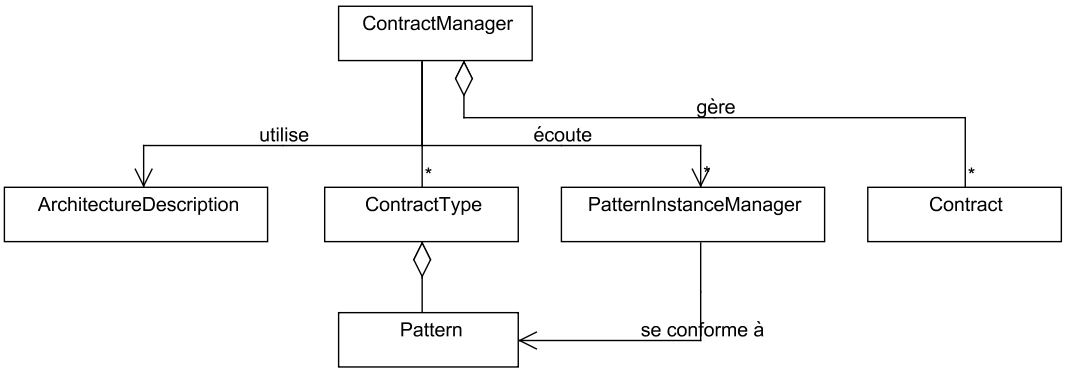


FIGURE 5.4 – ContractManager

5.3.1.3 Evolution du contrat

A sa création un contrat est associé avec l'instance de motif d'architecture `PatternInstance` qu'il contraint. Chaque contrat a la particularité d'être un écouteur de son

PatternInstance qui émet des événements notifiant son évolution. Ainsi chaque fois qu'une relation (Relation) est ajoutée au PatternInstance, le contrat est averti. Il peut ajouter l'une ou l'autre des extrêmités, voir les deux, de la relation à ses participants. Suivant la nature de la relation et la description de ce qui est échangé via cette relation, le contrat obtient les spécifications qui contraignent ces nouveaux participants auprès du référentiel de spécification ContractualSpecificationReferential. Le contrat crée alors les clauses qui en découlent et met à jour son accord. Le PatternInstance est notifié des évolutions de l'architecture par le PatternInstanceManager qui l'a initialement créé et qui lui est abonné au RelationObserver que lui fournit l'ArchitectureDescription.

5.3.2 Système de contrat

5.3.2.1 Gestion des ContractManager

Le système de contrat global est matérialisé par un singleton ContractSystemManagement. Ce dernier possède un ContractManagerSet qui référence l'ensemble des ContractManager comme montré sur la partie centrale de la figure 5.5. Parmi ces derniers, le ContractManagerSet distingue le ContractManager "toplevel" qui gère les contrats entre les entités de plus haut niveau, des ContractManager, associés chacun à une Reference, qui gèrent les contrats entre les éléments à l'intérieur de cette dernière. Le ContractManager gérant les contrats entre les entités de plus haut niveau est systématiquement créé. Le ContractManager associé à un élément de plus haut niveau, c'est à dire gérant les contrats entre sous éléments de ce dernier, est créé si ce dernier est composite. La création d'un ContractManager, associé à une Reference vers un élément qui n'est pas de plus haut niveau, a lieu sous deux conditions :

- la Reference désigne un élément composite (c'est à dire abritant potentiellement des sous éléments),
- une relation d'inclusion est crée (ou existe), incluant le Reference associé au nouveau ContractManager, dans une Reference déjà associé à un ContractManager,

La disparition d'un ContractManager a lieu quand la relation d'inclusion, plaçant sa Reference associée en sous élément d'une autre Reference associée à un autre ContractManager disparait. L'observation de l'existence, création et suppression des relations est effectuée à l'aide de l'interface ArchitectureDescription. Comme précédemment la mise en oeuvre des ContractManager est à la fois "statique" et "dynamique". A l'initialisation du système de contrat, l'architecture est explorée et les ContractManager nécessaires créés, puis d'autres sont ajoutés ou supprimés au gré de la création ou suppression des relations architecturales d'inclusion.

5.3.2.2 Architecture et plugins

Comme nous avons vu précédemment les ContractManager et le système de contrat font reposer leur fonctionnement sur deux ressources qui dépendent de la nature concrète de l'architecture et du formalisme de spécification : la description de l'architecture et les types de contrats. Ces ressources exposent des interfaces (ArchitectureDescription, ConttractType) définies dans le noyau du modèle de contrat mais dont l'implémentation est adhoc et donc externe au noyau car découplée et interchangeable. Chaque

`ContractManager`, ainsi que le `ContractManagerSet` doivent pourtant accéder à cette dernière. Pour ce faire le système de contrat, matérialisé par un singleton nommé `ContractSystemManagement`, charge les plugins d'architectures et de formalismes qui contiennent les implémentations adhoc des interfaces génériques nécessaires aux `ContractManager`. Le système global de contrat a donc l'architecture générale illustrée par la figure 5.5.

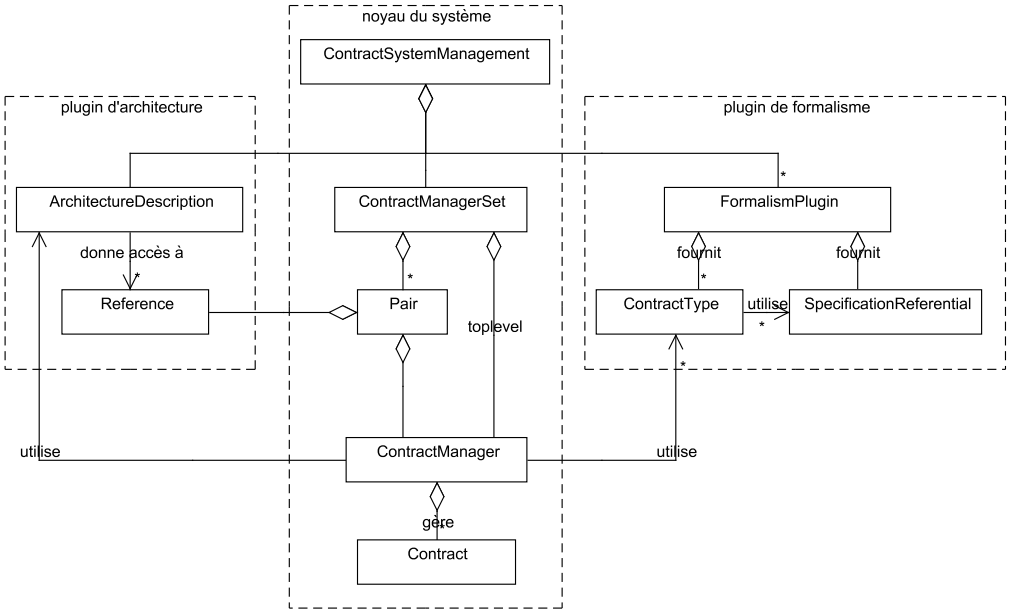


FIGURE 5.5 – Architecture du système de contrat

5.3.2.3 Accès au système de contrat

C'est depuis les `ContractManager` que l'administrateur du système gère les contrats. Pour ce faire il doit accéder à ces derniers et le fait via les `EntityReference` qui désignent les éléments de l'architecture. Ceux-ci lui sont accessibles via l'interface `ArchitectureDescription` que lui fournit le singleton `ContractSystemManagement`. Cette interface permet à l 'utilisateur d'accéder à des `Reference` en partant d'une qu'il possède, ou bien de créer une `Reference` à partir d'un objet (composant, service...) de l'architecture. Ainsi l'administrateur peut naviguer dans l'architecture et manipuler le `ContractManager` qui a en charge l'ensemble d'éléments d'architecture sur lequel il souhaite agir.

5.3.2.4 Fonctionnement global du système de contrat

Le fonctionnement global du système de contrat est illustré par la figure 5.6. Concrètement l'utilisateur fait lire au singleton `ContractSystemManagement` un fichier de propriétés qui spécifie les plugins à charger et le répertoire où se trouve les fichiers de spécification. Le `ContractSystemManagement` instancie alors les plugins. Il demande

alors à chacun des plugins de formalisme de lire les spécifications et de construire un référentiel qui sera interrogé par le système pour créer les contrats. L'utilisateur initialise ensuite le plugin d'architecture en lui passant une référence à l'élément le plus élevé dans la hiérarchie de l'architecture du système qu'il souhaite contraindre, ou vers tous les éléments si le système est plat. Le `ContractManagerSet` explore alors l'architecture pour produire les `ContractManager` adaptés et aussi se mettre à l'écoute de l'évolution de cette dernière. Une fois cela fait l'utilisateur peut utiliser le plugin d'architecture pour naviguer dans la configuration du système architectural. Pour chaque élément il peut obtenir le `ContractManager` qui gère les contrats auxquels il est susceptible de participer. Il peut observer les instances de contrats créées pour les différents types de contrat proposés par chaque plugin de formalisme. Les contrats ainsi créés peuvent s'évaluer sur le champs, si ils ne requièrent pas d'observations du système (preuve, model-checking, etc), ou bien ils peuvent attendre des observations produites par le plugin d'architecture sur la configuration ou l'exécution du système. Pour résumer nous pouvons dire que l'utilisateur doit fournir au système :

- un fichier de propriétés pour l'initialisation du système de contrat (pour désigner les plugins à charger),
- des fichiers de spécifications (lus par le `FormalismPlugin`),
- des références vers les éléments d'architecture du système qu'il souhaite contraindre (pour initialiser le `ArchitectureDescription`),

Le système en contrepartie lui fournit des diagnostics de l'état du système vis à vis des spécifications.

5.3.3 Plugins et spécialisation du système de contrat

Dans cette partie nous allons aborder le système de contrat du point de vue de l'intégrateur d'architecture et de celui de l'intégrateur de formalisme. Nous allons décrire comment occupant chacun de ces rôles nous avons intégré la plate-forme Fractal et le formalisme CCLJ au modèle de contrat.

5.3.3.1 Intégration de Fractal

Nous avons souhaité intégrer le modèle de composant Fractal et plus particulièrement son implémentation de référence Julia au système de contrat. Pour ce faire il nous a fallu extraire de ce modèle les entités et mécanismes réclamés par la représentation de l'architecture qu'a le modèle de contrat. Nous nous sommes à cette fin appuyés sur trois caractéristiques essentielles de Fractal : sa capacité d'introspection, d'intercession et d'extensibilité.

Désignation et description de l'architecture

Fractal réifie ses composants et leurs interfaces fonctionnelles. Julia, l'implémentation de référence de Fractal étant en java, les interfaces des composants ont pour implémentation des interfaces java et leurs méthodes sont donc aussi réifiées. Ainsi les éléments composant un système Fractal/Julia sont tous référençables. De ce fait il suffit alors, pour le type de chacun d'entre eux, de spécialiser la classe `Reference` en une sous classe possédant une référence vers cet élément. Nous avons donc créé les classes

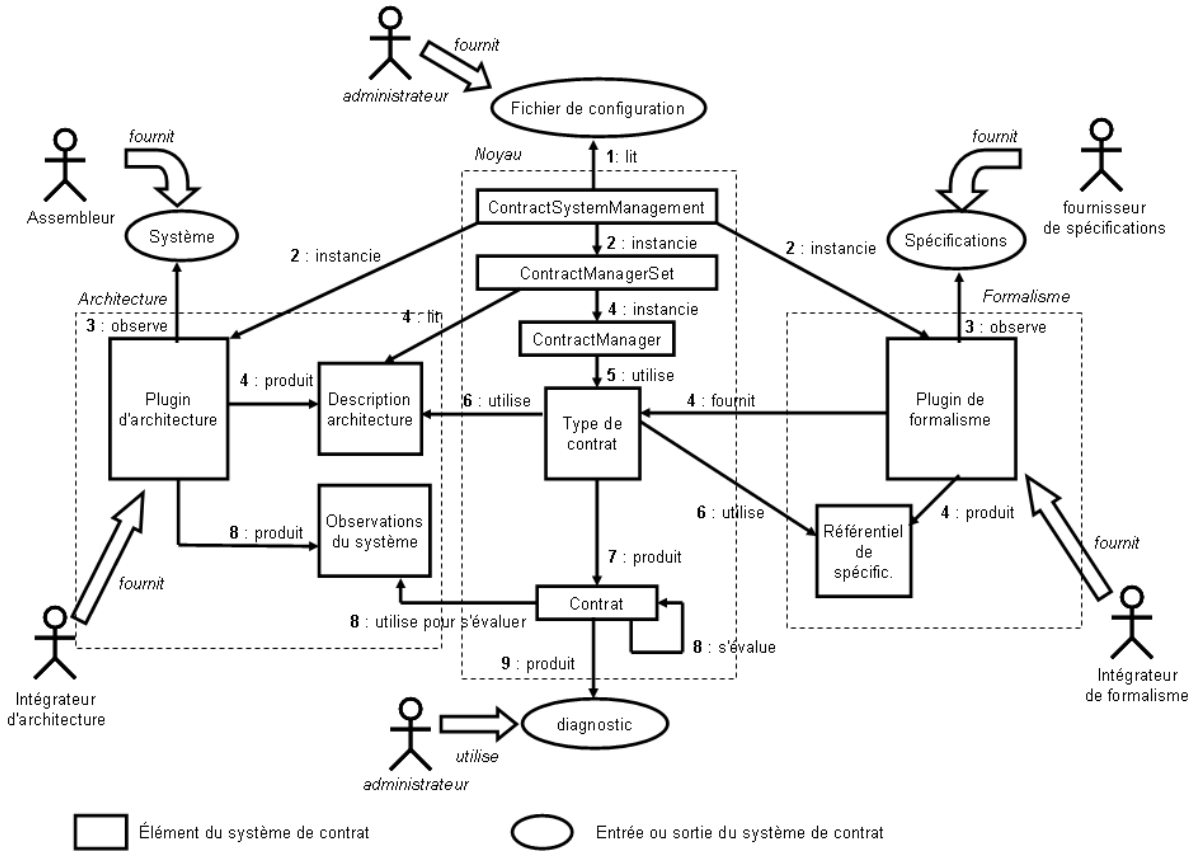


FIGURE 5.6 – Fonctionnement du système de contrat

`FractalComponentReference`, `FractalInterfaceReference` et `FractalMethodReference`. Par ailleurs le modèle Fractal est de plus "navigable" c'est à dire qu'il permet de connaître dans le cadre d'une configuration de composants :

- pour un composant les interfaces qu'il expose,
- pour une interface le composant qui l'expose,
- pour une interface, celle qui lui est fonctionnellement connectée,
- pour un composant composite, ceux qu'il contient,
- pour un composant celui qui le contient,

Pour chacune de ces relations, il est possible de spécialiser la classe `Link` avec la fonction de navigation correspondante fournie par Fractal. Chacune de ces sous classes permettra alors de passer d'une référence à celle qui lui est liée via la fonction de navigation associée à la relation qu'elle représente. Ainsi en mettant bout à bout ces sous classes de `Link` il est possible en partant d'une référence à un élément du système d'obtenir celle(s) qui lui est liée via cette suite de relations. Comme de plus Fractal décrit chaque élément de son architecture à l'aide d'attributs comme le nom (pour les composants, interfaces et méthodes), la signature (pour les méthodes et les interfaces), des variables définies par le configurateur de l'élément etc... il est possible de restreindre via ceux-ci les ensembles d'éléments retournés ou pris en compte par les `Link`. La classe `Node` vient pour se faire s'intercaler entre chaque paire de `Link`. Par elle transitent les références résultats d'une navigation qui servent de point de départ à une autre. Elle peut être spécialisée pour filtrer les éléments sur la valeur de leurs attributs et pour ne transmettre au `Link` sur lequel elle pointe que ceux correspondant à ses critères. Nous avons donc une sous classe de `Node` pour chaque type d'élément "navigué", `FractalComponentNode`, `FractalInterfaceNode` et `FractalMethodNode`.

Observation du système

L'implémentation Julia de Fractal propose deux mécanismes d'interception de l'exécution. Les "intercepteurs" pour les interfaces fonctionnelles et les "mixins" pour les interfaces non fonctionnelles (essentiellement de configuration) sont des fragments de code intercalés entre une interface et l'environnement de son composant. Ils sont traversés par tout message entrant ou sortant de l'interface que celle-ci soit cliente ou serveur. Sur la base de ces mécanismes nous avons construit une librairie d'observation de l'activité (aussi bien fonctionnelle que non fonctionnelle) des systèmes de composants Fractal. Les différents événements produits par cette librairie sont autant d'instants auxquels peuvent être définies des observations. Ils sont donc autant d'objets de spécialisation de la classe `TriggerDescription`: `FractalInterceptionTriggerDescription`, `ComponentStartTriggerDescription`, qui déclencheront l'observation sur leur détection. De plus notre librairie d'observation de l'activité associe à l'interception d'un message la description de celui-ci. Valeurs des paramètres d'un appel de méthode, valeur de retour, référence de l'appelant, de l'appelé font ainsi partie des événements émis par la librairie. Il y a donc autant de spécialisations de `DataDescription`, l'information que peut recueillir une observation.

Intégration des contrats dans Fractal

L'implémentation Julia de Fractal offre la possibilité d'ajouter de définir des types de composants aux caractéristiques non fonctionnelles étendues. Ces fonctionnalités sont implémentées dans la membrane du composant sous la forme d'objets nommés "Controller". Cycle de vie, connexion d'interface, inclusion de sous composants etc... sont à la charge des Controller. Le type d'un composant est défini par la constitution de sa membrane en termes de Controller, et il est possible de créer de nouveaux Controller et de les inclure dans la membrane de nouveaux types de composant. Nous avons vu que chaque `ContractManager` administrait les contrats entre les sous composants d'un composite. Ainsi chaque `ContractManager` est associé à un composant composite dont il garantit la validité via celle de l'assemblage qu'il abrite. Il est donc naturel de faire du `ContractManager`un Controller, inclus dans la membrane du composant composite dont il gère les sous composants. Cela permet de tirer parti de l'organisation naturellement hiérarchique des composants Fractal tout en conservant leur modularité. Pour ce faire nous avons défini `FractalContractManager` qui hérite de l'interface `Controller` et qui dispose du code permettant son intégration dans Julia. Par contre cela suppose de définir un nouveau type de composant, dit "contractualisé" qui doit explicitement être mis en oeuvre dans l'ADL du système Fractal, qu'on souhaite contraindre.

5.3.3.2 Intégration de CCLJ

Afin d'intégrer le formalisme de spécification CCLJ dans le système de contrat nous avons du implémenter un mécanisme qui soit capable, sur la base des spécifications CCLJ, de retourner les spécifications contractuelles correspondantes à un élément donné du système. Nous avons du aussi définir les types de contrat associés à ce formalisme.

Obtention des spécifications contractuelles

Pour bâtir le mécanisme de traduction en spécifications contractuelles des spécifications CCLJ nous avons du nous appuyer sur une modélisation de ces dernières. Nous avons pour ce faire réutilisé une modélisation que nous avions développé pour Confract. Une illustration simplifiée du diagramme de classe d'une spécification est donnée dans la figure 5.7. Chaque `FractalConstraint` regroupe un ensemble d'assertions (`JavaConstraint`) portant sur un composant Fractal donné. Chaque `JavaConstraint` fait porter une contrainte sur une méthode ou toutes les méthodes d'une interface Fractal (`ContextDeclaration`). Nous n'avions pas au moment de cette modélisation le principe de représentation générique des formalismes. Toutefois les différents éléments de la modélisation correspondent assez directement à des éléments de l'approche générique que nous avons défini par la suite. Nous distinguons la contrainte (`Formula`) et le moment (`composant Fractal + ContextDeclaration + TemporalStereoType`) auquel doivent être effectuées les observations. L'objet de ces dernières n'apparait pas sur le diagramme mais est bien présent dans la modélisation. Pour obtenir la modélisation de spécifications données, deux grammaires imbriquées sont mises en oeuvre à l'aide de parseur ANTLR.
Une première grammaire parse les expressions du type :

```
on '<nom_de_composant>'
 context 'interface_fractale.signature_de_methode'
   pre : 'formule_en_java_etendu'
```

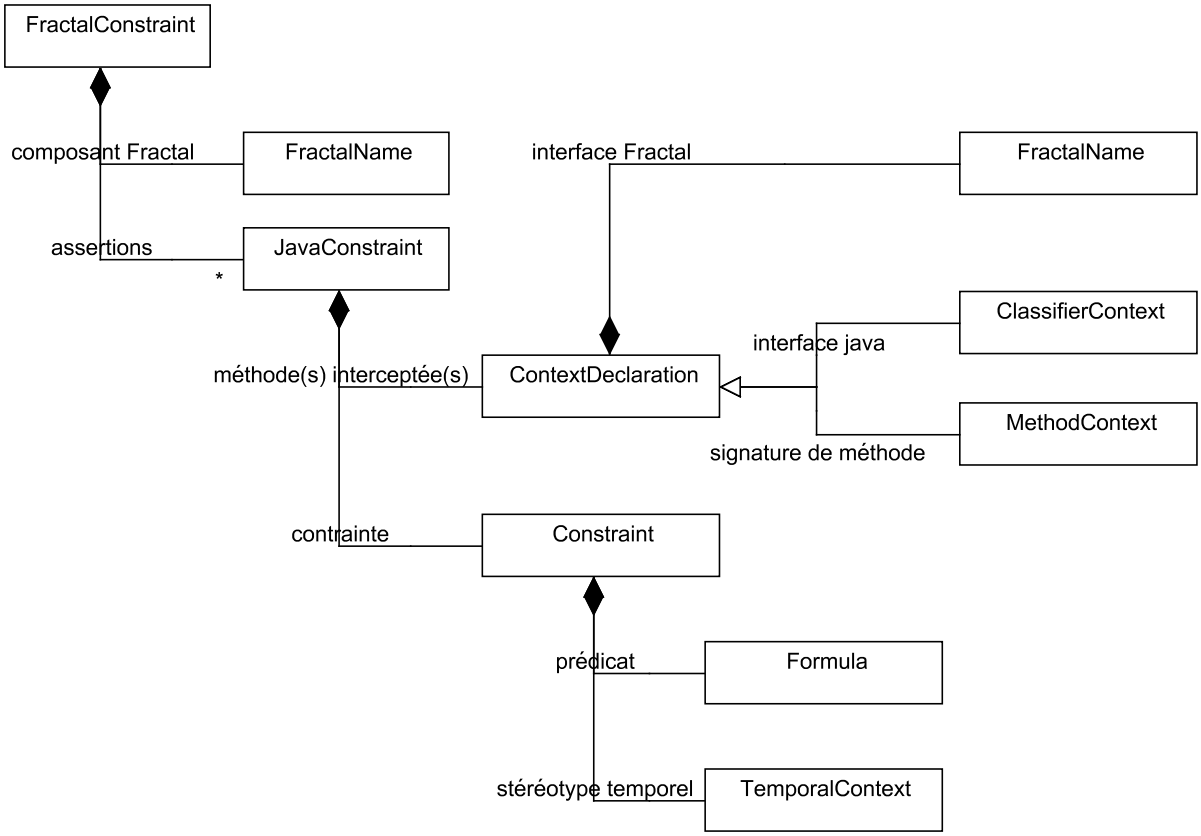


FIGURE 5.7 – Modélisation d'une contrainte CCLJ

```
post : 'formule_en_java_etendu'
inv : 'formule_en_java_etendu'
```

Il faut noter que le nom du composant et la signature de la méthode (interface Fractal, nom de la méthode, types des arguments) peuvent contenir des caractères génériques. Une seconde grammaire est utilisée pour parser les expressions de java étendu dont sont constitués les prédicats des assertions. En effet ces prédicats peuvent contenir des noms composés faisant intervenir des noms de composants et d'interface Fractal, par exemple `<ig>.gmt.getGroupsId()`. Cette dernière expression décrit l'appel de la méthode `getGroupsId()` de l'interface `gmt` du composant `<ig>`. Le parseur définit alors une variable contenant une référence vers l'interface `<ig>.gmt`. Cette variable sera initialisée dans l'environnement d'évaluation de la formule. Le parseur identifie par ailleurs les autres variables (arguments, valeur de retour, autres références architecturales) utilisées dans la formule et les lui associe. Par ailleurs des structures de quantification ont été ajoutées à java, les objets de la classe `Collection` peuvent être utilisé de la manière suivante :

```
List collec=new ArrayList();
collec.add(new Integer(2));
collec.add(new Integer(4));
collec.add(new Integer(7));

Collection res = collec.select(Integer y: y.intValue()>4);
// retourne la collection de valeurs > 4

Collection res = collec.reject(Integer y: y.intValue()>4);
 // retourne la collection de valeurs <= 4

Collection res = collec.collect(Integer y:
                                      new Integer(y.intValue()+10));
// retourne la collection avec les valeurs augmentées de 10

boolean res = collec.include(new Integer(5)
// retourne vrai si la collection contient 5

boolean res = collec.exclude(new Integer(5)
// retourne vrai si la collection ne contient pas 5

boolean res = forEach(int x in 0 to 2 : collec.get(x)<10))
// retourne vrai si les trois premiers éléments sont < 10

boolean res = collec.exists(Integer x:x>4);
// vrai si au moins une valeur > 4

boolean res = collec.forAll(Integer x:x>5);
// toutes les valeurs sont elles > 5
```

Une fois la spécification modélisée elle est rangée dans un référentiel. Ce dernier est un dictionnaire qui associe une contrainte (`Constraint`) à une contexte d'interception (`composant Fractal` + `ContextDeclaration` + `TemporalStereoType`). A ce stade la spécification n'est pas encore traduite en spécification contractuelle de la forme

hypothèse-garantie. En effet pour savoir si une précondition, une postcondition est une hypothèse ou une garantie il faut savoir si l'interface à laquelle elle est attachée est cliente ou serveur. Or à ce stade, même si nous avons le nom du composant et de l'interface Fractal associés à l'assertion, nous ne disposons pas de description de l'architecture. La traduction a lieu quand le système de contrat produit les clauses. En effet pour chaque participant le système demande alors à la librairie les spécifications qui concernent ce dernier. Il est alors possible de connaître le statut de chaque interface en observant le participant et donc d'interpréter pré et post conditions qui lui sont attachées. Les couples d'hypothèses et garanties sont formés à partir des assertions associées dans un même bloc déclaré par une expression : `on <composant> context 'method'`. Les spécifications sont donc traduites à la volée en spécifications contractuelles suivant le composant auquel elle sont appliquées.

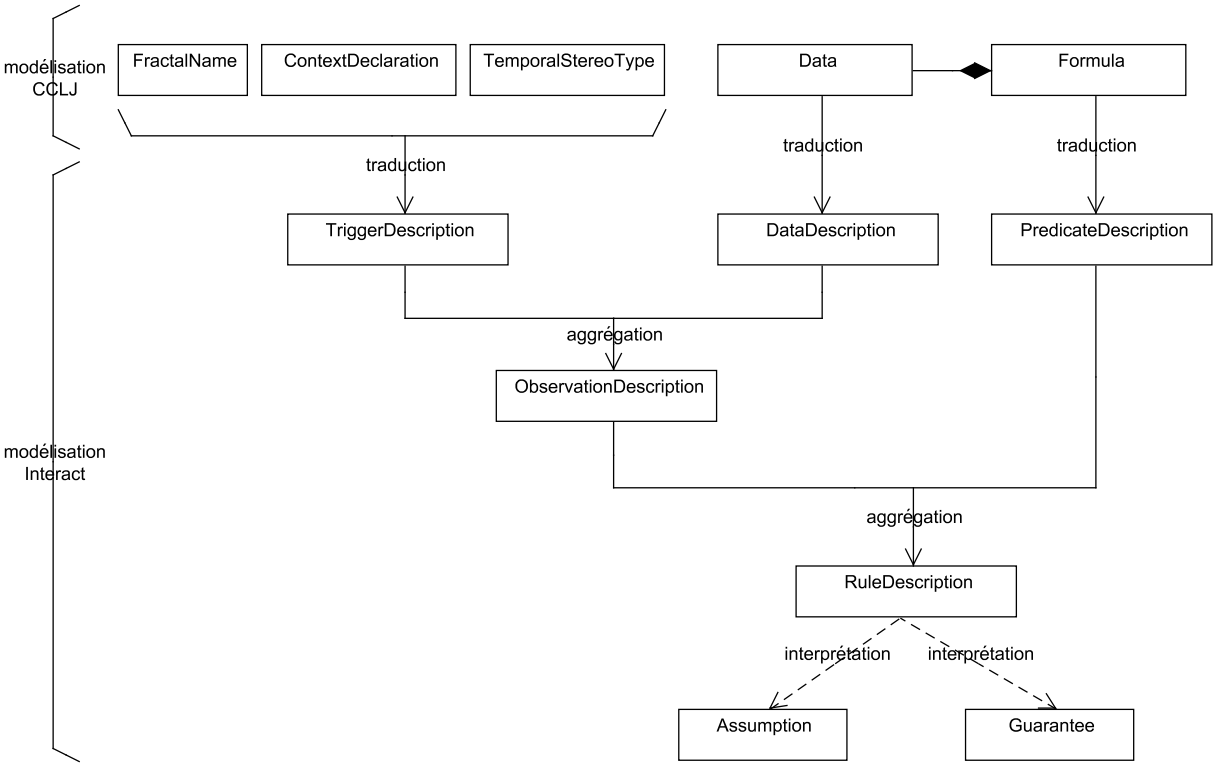


FIGURE 5.8 – Traduction de la modélisation CCLJ

Le processus de traduction et d'interprétation de spécifications CCLJ est illustré sur la figure 5.8. Les classes décrivant le contexte de l'interception de l'assertion (`composant Fractal + ContextDeclaration + TemporalStereoType`) sont utilisées pour produire la description du déclencheur d'observation (`TriggerDescription`). La description des données (`Data`) utilisées par le prédicat (`Formula`) pour s'évaluer sont transformées en description des valeurs à observer (`DataDescription`). Données à observation et déclencheur de l'observation sont rassemblés dans la description de l'observation (`ObservationDescription`). Enfin le prédicat de la spécification (`Formula`) est utilisé pour produire la description du prédicat de la règle (`PredicateDescription`). Cette dernière associée aux descriptions d'observations formeront la description de la règle (`RuleDescription`) qui sera par la suite interprétée comme étant une garantie

ou une hypothèse.

Définition des types de contrat

Comme nous avons vu dans la description du modèle de contrat, le type de contrat est défini sur la base :

- d'un motif architectural,
- des types de spécifications considérées pour les différents participants du contrat définis par le motif architectural,

Ces deux éléments sont fortement liés. En effet les spécifications définissent les interactions que le contrat prend en compte et donc les relations à observer. Sous cet angle le motif architectural découle des spécifications considérées. Il est possible inversement de partir des interactions portées par les relations du motif architectural pour ne retenir que les spécifications qui les contraignent.

Dans un premier temps nous avons choisi de distinguer deux types de spécification afin de décomposer l'architecture Fractal en motifs récurrents et indépendants du point de vue de la compatibilité des spécifications. Ces deux types de spécifications diffèrent par leur portée (cf. figure 5.9) :

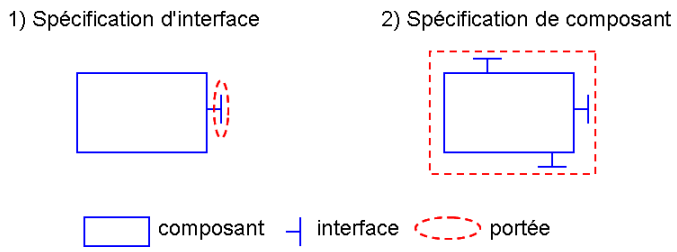


FIGURE 5.9 – Portées des spécifications CCLJ

1. l'assertion dont la portée est l'interface de la méthode qu'elle contraint : nommée "spécification d'interface", par exemple :

```
on <c>
 context void umt.addUserTab(List users)
  pre:
   users.size()<20-umt.registrationQueueSize();
```

   Cette spécification ne repose que sur valeurs issues de l'interface `umt` : `users.size()` et `umt.registrationQueueSize()`.

2. l'assertion dont la portée est le composant portant l'interface de la méthode qu'elle contraint : nommée "spécification de composant", par exemple :

```
on <af>
 context void umt.addUserTab(List> users)
  pre:
   users.size()<c.getMaxConnections();
```

Cette spécification fait intervenir une grandeur associée à l'interface `umt` : `users.size`, et une autre associée à l'interface `c` : `c.getMaxConnections()`.

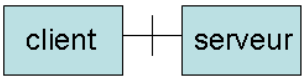
Pour classer les spécifications dans ces deux catégories, nous nous basons sur la modélisation que nous avons utilisé dans la partie précédente. En particulier il s'agit d'étudier les `ObservationDescription` d'une règle pour voir si elles se restreignent à une seule interface ou portent sur plusieurs interfaces d'un composant.

Distinguer les spécifications d'après leur portée permet d'étudier différents types de compatibilité et de conformité. Du point de vue de la conformité, on peut ainsi étudier séparément la conformité d'une interface à sa spécification de celle d'un composant à sa spécification. Du point de vue de la compatibilité, cela permet de distinguer finement :

- la compatibilité de deux interfaces,
- la compatibilité de deux composants,
- la compatibilité d'une interface et d'un composant.

Nous envisageons ainsi quatre types de contrat, chacun associé à un motif architectural donné pour des types de spécification donnés. Pour chaque interface, il est possible d'étudier la conformité de son porteur à sa spécification (propriété n°1), et pour chaque connexion, la compatibilité des spécifications des interfaces connectées (propriété n°2). Les motifs correspondants aux quatres types de contrats sont les suivants :

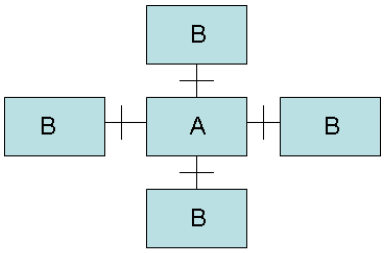
contrat Client-Serveur (motif architectural n°1), correspondant, pour les composants Client et Serveur, aux spécifications d'interfaces sur l'interface connectée.



n° 1

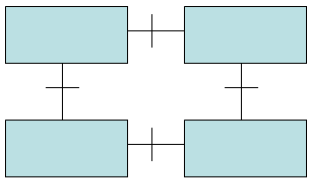
Contrat d'utilisation (motif architectural n°2), correspondant pour :

- composant A : spécifications de composant sur ses interfaces connectées ;
- composants B : spécifications d'interface sur leurs interfaces connectées au composant A.



n° 2

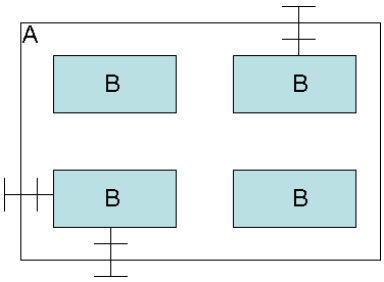
Contrat d'assemblage horizontal (motif architectural n°3), correspondant pour tous les composants d'un même niveau aux "spécifications de composants" sur leurs interfaces interconnectées.



n° 3

Contrat d'assemblage vertical (motif architectural n°4), correspondant :

- composant A : spécifications de composant, pour le composite ;
- composants B : spécifications de composant, pour ses sous-composants.



n° 4

La prise en compte de ces différents motifs architecturaux passe par la spécialisation des classes `Pattern` et `PatternInstance`. La classe `Pattern` indique si une relation du système contraint peut faire partie d'une de ses instances (`PatternInstance`). Tandis que les instances de `PatternInstance` rassemblent les relations qui dans l'architecture forment des instances de motifs d'architecture (i.e. sont acceptées par les `Pattern`). Par la suite c'est l'usine de contrat incluse dans le type de contrat qui filtre les spécifications associées à un élément en fonction de sa position dans son `PatternInstance`, comme décrit pour les différents types de contrat.

5.4 Discussion

5.4.1 Par rapport à l'état de l'art

Validité de l'assemblage. La nécessité pour un assemblage de voir ses participants conformes à leurs spécifications et celles-ci compatibles est communément admise. Toutefois ces propriétés ne sont pas réifiées par les contrats rencontrés dans l'état de l'art. Notre modèle de contrat au contraire réifie la conformité et la compatibilité de ses participants. Ces objets sont évaluables et leur état reflète dans quelle mesure les participants du contrat satisfont à ces propriétés. Ainsi l'état du contrat, "mécaniquement" déduit par conjonction de ces propriétés, reflète la validité de l'assemblage. Un autre intérêt de cette réification est que ces objets "propriétés" alors explicitement liés aux participants et à leurs spécifications, permettent de raisonner "mécaniquement"

sur leur valeur par rapport à ces derniers. Un exemple simple en est la responsabilité des participants explicitement établie dans le cadre de chaque clause. Dans ce cas la non satisfaction de la clause est imputée au composant désigné responsable. Mais dans le cas par exemple du contrat composite il est possible de naviguer dans des contrats hiérarchiques pour découvrir un responsable situé à un autre niveau de composition architecturale. Cette approche ne semble accessible que dans le cas des contrats de service pour des relations uniquement client serveur et non hiérarchiques.

Découplage du formalisme. Par ailleurs il est intéressant de constater dans l'état de l'art que les modèles de contrat qui considèrent la compatibilité entre les spécifications imposent le plus souvent le formalisme de ces dernières. Nous pensons avoir montré que la contrainte pour un formalisme de pouvoir s'interpréter en terme d'hypothèse-garantie et de disposer d'outils d'évaluation adhoc, était suffisante pour permettre son utilisation dans le cadre de l'étude de la compatibilité, dans le cadre de leurs échanges, des participants du contrat.

Découplage de l'architecture. Nous croyons avoir aussi montré qu'il était suffisant de modéliser un système, qui fonctionne sur la base de la collaboration de ses éléments, par les relations conduisant ses échanges. Celles-ci peuvent être explicitement fournies comme dans le cas des composants, ou être déduites comme dans le cas des objets ou des services. Toutefois dans les deux cas elles définissent les interactions qui vont avoir lieu et par là guident la recherche et le choix des spécifications qui vont les décrire et dont il va falloir considérer la conformité et la compatibilité.

5.4.2 Intérêts de l'approche

Architecture : systèmes complexes. En dehors de la comparaison avec les travaux existants nous pensons que notre solution possède des intérêts intrinsèques. Tout d'abord, plus que la conformité d'un système à sa spécification nous nous intéressons à la validité des interactions qu'il abrite. Ce trait est intéressant car certains systèmes sont difficilement spécifiables, le système présenté dans le premier chapitre en est un exemple : la spécification prévoyant quand des messages doivent être envoyés serait sans doute aussi longue que la description du système lui même. Pour s'assurer du bon fonctionnement de ces systèmes l'étude de la validité des interactions qu'ils abritent est alors une solution pragmatique car elles composent leur fonctionnement même.

Spécifications : cohérence des observations. Par ailleurs il est reconnu que pour s'assurer du bon fonctionnement d'un composant ou d'un service différentes natures de spécifications doivent être prises en compte et nous avons vu que notre système autorisait cela. Toutefois il convient que ces différentes propriétés soient satisfaites simultanément. Il faut donc pouvoir comparer les observations sur lesquelles leur évaluation repose. Or notre conception impose que les observations soient toutes fournies par le plugin d'architecture, sous une forme unique, même si il faut augmenter le plugin pour les besoins particuliers d'un formalisme. Ce point est incontournable pour rendre possible le partage des observations par les différents formalismes, ou quand ce n'est pas le cas la comparaison des différentes observations utilisées par chacun. Il est de ce fait alors possible de discuter de la concomitance de la satisfaction des spéci-

fications.

Administration : support d'architectures en couches. Dans le cadre d'une architecture logicielle en couches, l'étanchéité des couches peut handicaper fortement la maintenance de l'application, car une erreur à un niveau donné n'est alors pas associée à une autre à un niveau ou il est possible d'agir. Or nous avons montré que notre système, découplé de la nature concrète de l'architecture, pouvait s'appliquer aussi bien à des composants qu'à des services. Nous avons aussi vu que notre modèle autorisait la création de dépendances entre niveaux de compositions architecturales. C'est à dire que dans un contrat à un niveau donné de composition, une clause peut dépendre du respect d'un contrat entre les éléments qui composent son responsable. Or il n'est pas rare que dans un système constitué de couches, une couche de services soit implémentée par une sous couche de composants. Les services sont alors composés de composants. La combinaison des deux propriétés de notre modèle que nous venons de donner permettrait de l'appliquer à ce système de sorte que les contrats entre services dépendent des contrats entre composants. Ainsi il pourrait être possible de mécaniquement lier la défaillance d'un service à celle d'un composant et par là de faire communiquer l'administration des différents niveaux à l'aide de concepts communs.

5.4.3 Limites de la solution

Formalismes de spécification. Du point de vue de l'intégrateur de formalismes, dans le cadre de notre conception d'implémentation nous avons laissé de côté les formalismes qui ne sont pas explicites quant aux éléments d'architecture qu'ils contraignent, comme par exemple TLA [22]. Les spécifications exprimées dans ces formalismes nécessiteraient pour être appliquées que leur soit ajouté un fichier qui désigne les éléments de l'architecture correspondant aux expressions qu'elles emploient. Par ailleurs nous avons vu que pour être traduite, de manière mécanique, sous forme "hypothèse-garantie" les spécifications devaient être modélisables. Il convient de pondérer cette limite. Il faut pouvoir distinguer au sein de la spécification les observations et les contraintes à évaluer. Pour les observations il faut distinguer leur objet et leur instant, mais il n'est pas nécessaire de modéliser complètement le corps des contraintes dont l'interprétation est laissée aux outils d'évaluation. Enfin la limitation la plus forte concernant les formalismes de spécification est que dans certains cas leurs sémantiques mettent en oeuvre des durées qu'il faut modéliser à l'aide d'observations discrètes. Un exemple classique de ce problème est de celui de l'expression des invariants de la conception par contrat, qui expriment qu'une propriété doit rester vraie entre les exécutions de méthodes, alors qu'il n'est possible de la vérifier qu'en des instants discrets.

Architectures. Du point de vue de l'intégration de différents modèles d'architecture dans le système de contrat nous avons vu qu'il fallait que celles-ci devaient pouvoir être interprétées en termes de relations. Dans le cadre de notre modèle nous accordons la même valeur à toutes les relations. Si nous considérons une orchestration de services dans laquelle le résultat d'un service A sert d'entrée à un service B, alors nous pouvons considérer que ces derniers sont liés. Mais il est possible que l'orchestration prenne une valeur issue d'un service C et sur la base d'un test sur cette valeur, l'envoie soit au service A, soit au service B. Alors les relations entre les services C et A ,et les services C et B peuvent être de courtes durées, voire ne servir qu'une seule fois. Nous

considérons dans notre approche que le contrat qui lie C et A et B doit être satisfait. Toutefois si nous considérions séparément le contrat entre A et C et celui entre C et B, et si A n'était appelé qu'exceptionnellement, son contrat avec C risquerait d'avoir moins de valeur que son contrat avec B, du point de vue du fonctionnement du système. Nous n'abordons pas ce point.

Par ailleurs nous ne considérons que les incompatibilités que nous pourrions nommer "directes", c'est à dire entre des éléments directement connectés et interagissant. En effet si nous observons un fournisseur envoyant une même valeur simultanément à plusieurs consommateurs, il peut y avoir une incompatibilité entre ces derniers si leurs hypothèses sur la valeur qu'ils reçoivent sont contradictoires. Pour autant les consommateurs ne sont pas liés entre eux et n'échangent rien, mais le contrat entre le fournisseur et les consommateurs sera toujours invalide car il y aura toujours un consommateur d'insatisfait. Cette remarque nous ramène, dans le cadre de l'interprétation de la compatibilité, à la notion du partage plus fondamentale que celle de l'échange. En effet car avant même d'être échangée une chose doit être partagée.

Enfin nous n'avons pas voulu dans notre approche rentrer dans le détail de la modélisation temporelle des instants d'observation. La cohérence temporelle des observations est laissée à la charge de celui qui les implémente car elle peut être mise en oeuvre de nombreuses manières suivant les caractéristiques des architectures sur lesquelles portent les observations. Les observations fournies par l'implémentation du plugin d'architecture doivent fournir suffisamment d'information pour que les gestionnaires d'observations implémentés dans le plugin de formalisme puissent remplir leur tâche de garantie de leur cohérence temporelle.

5.5 Conclusion

L'application du contrat dans un contexte réel nous a montré que l'observation et la contrainte des interactions internes pouvait se révéler nécessaire à l'évaluation du bon fonctionnement de certains systèmes. Par ailleurs nous avons montré comment notre conception d'implémentation permettait de rassembler autour d'une même logique (celle des contrats) les différents intervenants à la mise en oeuvre du système contraint. Le modèle de contrat leur offre en effet une plate-forme commune d'échanges. De plus par rapport à l'état de l'art, notre modèle permet un raisonnement mécanique par rapport à l'évaluation de la validité de l'assemblage. Dans cette optique, il a l'avantage d'utiliser des notions indépendantes des formalismes de spécification et de l'implémentation de l'architecture. Enfin notre modèle présente des limitations par rapport à l'interprétation de certains formalismes et à la mise en oeuvre des observations. Mais il permet de traiter des systèmes au comportement complexe, éventuellement organisés en couches.

CHAPITRE 6

Conclusion

Notre approche se place dans le cadre des systèmes collaboratifs à base d'objets, de composants ou de services. Le fonctionnement de ces systèmes reposant sur leurs assemblages, nous nous sommes intéressés à la validité et au diagnostic de ces derniers. Mais comme les propriétés à prendre en compte pour discuter cette validité sont diverses, ainsi que les éléments qui peuvent composer ces assemblages, notre objectif s'est décomposé en trois points :

- évaluer la validité d'un assemblage et fournir un diagnostic en cas d'erreur,
- autoriser l'application conjointe de formalismes variés aux architectures d'un modèle de composants et services,
- autoriser l'application, de manière uniforme, d'un formalisme à des architectures de différents modèles de composants et services,

Pour les atteindre, nous avons basé notre démarche sur une définition simple de la validité d'un assemblage. Nous l'avons posée comme équivalente à la correction des interactions qu'il supporte, c'est à dire que chacun de ses élément doit recevoir ce qu'il requiert et fournir ce que les autres en attendent. Cette appréhension nous a ainsi directement amené à considérer l'approche par contrat, cette dernière organisant les échanges au sein d'une collaboration.

D'un point de vue applicatif, l'outil que nous proposons est utile aussi bien pour limiter le temps d'arrêt d'applications à haute disponibilité via leur diagnostic précis, que pour autoriser des compositions dynamiques non anticipées via leur validation à la volée.

Modèle proposé. Pour organiser et intégrer les propriétés caractérisant un assemblage, nous proposons un contrat qui est une entité de premier ordre dont l'état est évaluable et observable. Cet état qui reflète la validité de l'assemblage auquel est appliqué le contrat et permet d'appréhender ce dernier comme une fonction du système et de ses spécifications.

Par ailleurs, dans le cas où l'assemblage se révèle invalide, le contrat fournit un diagnostic du problème. Mais il faut noter que le contrat explicite aussi les raisons pour lesquelles l'assemblage est valide. En effet il réifie, organise, et raisonne sur les trois propriétés que nous avons utilisé pour définir la validité de l'assemblage sur la base de la correction de ses interactions : la conformité des participants à leurs spécifications, leurs responsabilités vis à vis de celles-ci, et la compatibilité des spécifications.

De plus notre modèle de contrat satisfait aux deux dernières propriétés que nous avions requises : il est indépendant du formalisme des spécifications pourvu que celles-ci

puissent s'interpréter en terme d'hypothèses et de garanties, et il est indépendant de la nature de l'architecture pourvu que celle-ci soit constituée d'éléments dont le fonctionnement repose sur la collaboration.

Framework développé. Afin de fournir une mise en oeuvre de notre modèle de contrat, nous avons conçu un framework qui génère dynamiquement les contrats, sur la base des spécifications et de l'architecture, tout en prenant en compte les rôles des différents intervenants d'un système contractualisé. Ainsi, notre framework fournit à l'administrateur du système l'évaluation de la validité de l'assemblage et un diagnostic en cas de problème, les deux dans les termes du contrat de la vie courante ("responsabilité", "accord"...). L'administrateur peut donc appréhender le fonctionnement de son système sans être un spécialiste de son implémentation et des formalismes qui le spécifient. D'autre part le système est décomposé en un ensemble de motifs architecturaux contraints séparément ce qui permet de simplifier la tâche d'administration.

La facette "bus de validation" de notre framework organise l'intégration indépendante d'un modèle architectural et de celle de ses formalismes de spécification. La première autorise l'application de notre contrat à des systèmes relevant de ce modèle. La seconde permet l'interprétation des spécifications dans ces formalismes pour composer les contrats.

Enfin, l'auteur des spécifications peut se concentrer sur la production de ces dernières sans connaître les travaux de l'intégrateur du formalisme qu'il utilise. Tout comme l'architecte du système n'a pas, pour bâtir un système, à connaître les travaux de l'intégrateur du modèle d'architecture qu'il utilise. Les rôles des intervenants vis-à-vis de la contractualisation des systèmes sont ainsi clairement séparés et relativement indépendants, ce qui facilite le découpage des tâches de développement.

Discussion vis-à-vis de l'existant. Vis-à-vis de l'état de l'art, nous proposons une définition pragmatique de la validité d'un assemblage d'objets, de composants ou de services. En effet, d'une part, elle met en oeuvre un affaiblissement des propriétés de conformité et compatibilité des spécifications afin de les traiter mécaniquement. D'autre part, elle est indépendante du formalisme des spécifications ainsi que du détail de l'implémentation du système. Elle repose sur les échanges observables réalisés sur les relations entre les participants du système. Elle fournit de cette manière un guide de mise en oeuvre pour les spécialistes de l'architecture et des formalismes tout en restant appréhendable pour leurs non spécialistes.

Par rapport à l'approche par contrat classique (Design By Contract) notre contrat réifie la propriété de conformité des participants à leurs spécifications. Il lui ajoute l'expression de leur responsabilité. Il complète cette conformité par la réification de la propriété de compatibilité des spécifications, établie sur la base des échanges définis par le motif d'architecture. Ces réifications lui permettent de raisonner sur la validité de l'assemblage et en particulier de fournir un diagnostic explicite et précis en cas de problème de collaboration des éléments de l'assemblage.

En étudiant les différents modèles d'architecture que sont les objets, composants et services, nous avons remarqué que la collaboration sous tendait le fonctionnement de ces différents systèmes, il est donc pertinent de faire reposer sur celle-ci notre modèle. Parallèlement l'observation de différents formalismes utilisés pour décrire les propriétés de ces systèmes nous a conduit à juger recevables les hypothèses que nous faisions sur les formalismes que pouvaient intégrer notre modèle. La plupart sont en effet accompagnés d'outils de mise en oeuvre, une certaine part d'entre eux autorisent l'expression de la conformité et de la compatibilité, ainsi qu'un traitement mécanique.

Par rapport aux modèles de contrat que nous avons rencontrés dans l'état de l'art, nous avons réifié l'ensemble des propriétés que nous avons définies comme nécessaires à la validation d'un assemblage. Nous avons par ailleurs explicitement lié celles-ci à la configuration architecturale à laquelle le contrat s'applique. Finalement notre framework tire partie des découplages autorisés par notre modèle pour cloisonner les tâches des différents intervenants autour du système contractualisé au travers des rôles que nous avons définis.

Limites. Les contributions de cette thèse comportent bien évidemment un certain nombre de limites. Parmi celles-ci, celles qui concernent sa portée doivent être clairement perçues. Ainsi, du point de vue des formalismes acceptés par nos contrats, il faut que ceux-ci, modulaires, se prêtent à une interprétation dans le paradigme hypothèse-garantie. Par ailleurs, pour exploiter complètement notre outil, ils doivent disposer de solutions d'évaluation de leur conformité et compatibilité. Enfin notre approche mécanique de leur traitement restreint leur niveau d'abstraction par rapport au système sur lequel ils portent, ils doivent explicitement se référer aux entités de ce dernier.

Du point de vue des architectures auxquelles notre modèle s'applique, celles-ci doivent être composées d'entités distinguables les unes des autres. Leur fonctionnement doit reposer sur la collaboration de ces dernières via des relations. La durée de vie de ces relations peut être variable mais le contrat que nous définissons est associé à une configuration donnée de relations. Chaque fois que cette configuration change, un nouveau contrat doit être établi. Il est intéressant de remarquer que si nous considérons que les systèmes informatiques fonctionnent en général soit sur la base de flots de messages soit sur celle de flots de contrôle, ou sur les deux à la fois, notre approche se limite à la garantie des échanges de messages. Toutefois, de cette manière, elle se trouve complémentaire des approches issues de la logique de Hoare qui, aussi sur la base du paradigme hypothèse-garantie, valident le flot de contrôle.

Nous devons aussi considérer notre modèle du point de vue du cycle de vie global d'un système auquel il s'appliquerait. Bien que nous définissions les rôles des différents intervenants autour du système augmenté de notre outil, nous ne fournissons pas de méthodologie globale de conception. De par son principe, notre outil ne peut s'appliquer avant que les relations entre éléments du système ne soient instanciées. Par contre les contrats créés au fur et à mesure de l'instanciation du système peuvent être évalués avant, pendant ou après l'exécution de celui-ci suivant que les outils de vérification des spécifications nécessitent ou non des observations de l'exécution.

Perspectives. Nous pourrions intégrer à notre outil un mécanisme de négociation, sachant que des travaux concernant celle-ci, dans le cadre du formalisme CCLJ, sont en cours([21], [20]). En effet, actuellement, la création de l'accord du contrat se fait sur la base de la configuration des éléments de l'architecture et de leurs spécifications sans adaptation des uns ou des autres. Pourtant les choix d'adaptation des éléments ou des spécifications pourraient être associés aux différents types de contrat, qui définissent chacun un schéma d'interaction vis à vis d'un type de propriété donné comme le comportement, la qualité de service... Ainsi le processus de négociation pourrait être adapté à chacun des schémas d'interactions défini par le type de contrat.

De plus, afin d'assurer une meilleure validation de notre modèle de contrat, il nous faudrait envisager son application à d'autres modèles d'architecture, pour d'autres formalismes et types de propriétés. Par exemple la prise en compte de propriétés de QoS et de concurrence permettrait d'assurer la pertinence de notre modèle sur l'ensemble des types de propriétés communément étudiées pour la validation des architectures. Dans ce cadre, nous verrions augmenter le nombre de contrats appliqués à une même partie de l'architecture, ainsi donc que le nombre total de contrats appliqués à un système. Il serait alors utile de pouvoir appliquer des priorités ou des "poids" à ces contrats. Ainsi propriétés et relations pourraient se voire appliquer des coefficients d'importance permettant le calcul du poids du contrat correspondant. De cette manière il serait par exemple possible d'autoriser certaines dégradations avant de déclencher des mécanismes de récupération. Considérer d'autres architectures nous amènerait d'autre part à étudier dans quelle mesure la présence de connecteurs ou de médiateurs dans l'architecture modifierait la mise en oeuvre de notre modèle. Ils joueraient sur l'expression de la compatibilité et sur les possibles processus de négociation.

Il pourrait par ailleurs être utile d'améliorer la finesse de la sémantique des contrats, relativement simple dans notre modèle. Pour ce faire il serait intéressant de proposer au sein du contrat des clauses de second ordre, c'est à dire dont les hypothèses et garanties pourraient faire intervenir d'autres clauses du même contrat, et même l'évaluation d'autres contrats. Dans cette optique il faudrait très certainement envisager une étude du contrat ainsi produit à l'aide d'outils formels comme les logiques temporelles et déontiques. Mais cette approche permettrait un calcul plus évolué des dépendances et des causalités entre propriétés des éléments de l'architecture sous une forme abstraite par rapport à leurs caractéristiques techniques. Ces "méta"clauses pourraient ainsi être associées à un motif architectural donné dans le cadre d'un type de contrat.

Enfin nous n'avons envisagé la collaboration des éléments de notre système que sous la forme d'échanges. Pourtant l'échange n'est qu'une des formes du partage, concernant dans ce cas la grandeur échangée, qui sous-tend l'idée même de collaboration. Ainsi, les collaborations à base de partage, sans échange direct, doivent aussi être prises en compte. Des éléments d'un système peuvent interagir via une ressource qu'ils partagent, en influant sur l'état de celle-ci. Ce type d'approche se conçoit facilement pour des ressources non fonctionnelles limitées, réifiées et utilisées par différents éléments du système (comme la bande passante, les ressources de stockage ou de calcul etc). De cette manière il peut, comme nous avons vu dans nos limitations, exister des incompatibilités sans relation directe entre les éléments, et la prise en compte de ces interactions

indirectes est nécessaire à la garantie globale d'un système.

Annexe A — Compléments à l'état de l'art des formalismes

A.1 Formalismes fontionnels pour les objets

A.1.1 Extension temporelle d'OCL :

Plusieurs travaux ont proposé l'extension de OCL avec une logique temporelle, mais Ziemann et Gogolla rapportent [109] que ceux ci, soit ne donnaient pas de base formelle à leur extension, soit ne prenait pas en compte l'héritage et le sous-typage, soit encore requéraient du développeur des compétences mathématiques avancées. Ces auteurs présentent donc le TOCL [109] qui propose des opérateurs temporels capables de poser des conditions temporelles au sein d'expression OCL. Les opérateurs décrits sont ceux portant sur des temps futurs : next, existNext, always, sometime, always, always e1 until e2, sometime e3 before e4, @next. Les auteurs exposent leur sémantique formelle, qui repose sur le triplet : séquence d'états, index de référence le long de la séquence, assignation de variables pour l'état courant. Quelques exemples d'utilisation sont les suivant :

```
context Program inv :
  (mode = 'initialization' and wlmdFailure)
     implies next mode = 'emergencystop'

context SteamBoiler::openValve ():
   post : always valve = #open until wlmd.q <= n2
```

Les opérateurs équivalents, tournés vers le passé, sont obtenus en inversant le parcours de la séquence d'état dans la sémantique formelle et les opérations sur l'index de référence. Les auteurs font remarquer par exemple que l'existence d'instances d'objets ou de relations peut être spécifiée dans le temps à l'aide de ces opérateurs. Enfin au moment de la rédaction de l'article, les auteurs ne disposaient pas encore d'outils de vérification de ces contraintes qu'ils auraient souhaité basés sur l'étude de traces d'exécution.
OCL dispose d'une bonne expressivité de part ses opérateurs et quantificateurs sur les collections. Néanmoins il ne dispose pas d'outils de réflexivité nécessaires à la manipulation des spécifications qu'il décrit. Avec l'exemple de [BDL04] une première mise en oeuvre opérationnelle efficace d'OCL a été produite. Et si elle n'est pas réflexive, elle offre une technique d'intégration non intrusive en utilisant les aspects.

A.1.2 JML

Description du langage

Dans le cadre des assertions classiques JML propose des quantificateurs de collection et des bibliothèques adhoc. Ces assertions portent [63] pour les invariants sur les

attributs de l'instance et pour les pre/post conditions sur les paramètres de la méthode, éventuellement mémorisés de l'entrée dans le cas des post conditions. L'héritage des spécifications est pris en compte par la vérification des affaiblissements et renforcements nécessaires, par exemple les préconditions sont affaiblies (disjonction) au contraire des invariants qui sont conjugués. Le modèle de comportement d'une classe repose sur la définition de variables dites de modèles. Celles-ci sont déclarées dans le code JML et visibles uniquement de celui-ci. Grâce à ces variables la spécification peut s'appuyer sur la description de modèles plus abstraits et génériques que les contrats traditionnels. Par ailleurs il est possible d'établir un lien entre les variables de modèles et les variables concrètes à l'aide de depends et represents. Ces deux clauses représentant des fonctions d'abstraction permettent le raffinement de la spécification JML en code exécutable.
Exemple :

```
  //@ public model non_null JMLObjectSequence absVal;
  private /*@ non_null @*/ Collection stackCollection;
  //@ private represents absVal <- stackCollection;
```

Dans cet exemple le contenu de la variable de modèle absVal, déclarée par public model, est associé à celui de la variable d'implémentation stackCollection par la clause represents. Ceci permet de mettre en oeuvre des spécifications du type :

```
public interface Stack
{
//@ public model instance JMLObjectSequence absVal;
// @ public instance invariant absVal != null;
/*@ public normal_behavior
   @         requires true;
   @         assignable absVal;
   @         ensures absVal.equals(\old(absVal.insertFront(x))); @*/
void push(Object x);
...
}
```

En effet il suffit alors que dans la classe implémentant l'interface Stack, la variable contenant la liste d'éléments implémentant la pile soit associée à absVal.
JML permet aussi de spécifier des conditions de terminaison brutale de la méthode avec une levée d'exception.
Exemple :

```
/*@ normal_behavior
  requires < précondition> ;
  ensures < postcondition> ;
  signals ( Exception1 ) <condition1> ;
  ...
  signals ( Exeptionn ) <conditionn> ;
@*/
```

Dans le cas où la levée d'exception fait partie d'un comportement particulier il est possible de le décrire à l'aide de la construction exceptionnal_behavior.
Exemple :

```
/*@ exceptional_behavior
  requires < précondition> ;
  signals ( Exception1 ) <condition1> ;
  ...
  signals ( Exeptionn ) <conditionn> ;
@*/
```

Les outils de JML

JML dispose d'un ensemble d'outil de vérification statique :

- ESC/Java : analyse le source java, et effectue des tests au delà de la vérification de type, de null pointer possibles, d'out of bound, et cast error. Il supporte une partie de JML et vérifie la synchronisation entre celle-ci et java. Il se veut accessible au développeurs java,
- ESC/Java2 : commence à prendre en compte les programmes modèles et prend en compte les spécifications assignable (autorisation d'effets de bords spécifiés),
- Chase : complète ESC/Java en testant les clauses assignable,
- LOOP : transforme les annotations JML en entrées pour prouveur de théorème,
- Krakatoa : produit des obligations de preuve pour le prouveur de théorème Coq pour un sous ensemble de java,
- JACK : vérification statique de code annoté par JML. produit des obligations de preuve pour "prouveurs B", se veut accessible aux développeurs (comme ESC/-Java),

En plus de ces outils, JML dispose de :
Outils de test :

- jmlUnit : combine JUnit avec JML. Il automatise la production de code de test JUnit en se basant sur les spécifications JML. Le code de test, dont le comportement est dérivé de la spécification de la classe testée, considère la violation d'une assertion comme un cas d'échec de test. Les auteurs font remarquer que la moitié des cas d'erreurs détectés provenaient d'erreurs de spécification, mais que de ce point de vue le temps passé à écrire des programmes de test est reporté sur la rédaction de spécification aussi complètes que possible.

D'autre part des outils d'assistance à la production de spécification, toujours décrits dans [18] :

- Daikon : se base sur le fonctionnement du programme pour détecter des invariants, se base aussi sur des suites de tests, il fait encore intervenir des analyses statiques et statistiques, et donne de bons résultats,
- Houdini : il fait des hypothèses d'annotations JML complétant celles existantes (analyse le code source), et appelle ESC/Java pour les vérifier et cycle ainsi. Il peut s'aider d'annotations JML existantes dans le programme,

Extensions et travaux autour de JML

Le succès de JML a rendu possible des études de retours d'utilisation en vraie grandeur. En particulier dans le domaine des applications JavaCard destinées aux SmartCard, JML a été utilisé, entre autre, pour écrire une spécification de quasiment toute l'API Java Card [18], dans un cadre ou la sûreté de fonctionnement est critique. Ainsi en termes de correction, Patrice Chalin propose une mise à jour de la spécification de JML [19], pour prendre en compte des problèmes de mise en oeuvre de grandeurs numériques dans la spécification. L'auteur a en effet pu mettre en évidence des différences entre la sémantique JML et la sémantique java, du moins celle à laquelle les utilisateurs s'attendent, autour de ces points. Il propose de fournir à terme une extension de la sémantique formelle de cet aspect de JML, et d'étudier la formalisation d'autres points, afin d'éviter les imprévus rencontrés dans la mise en oeuvre des grandeurs numériques et de rendre accessible ces formules aux prouveurs. Un outil de spécification temporelle de programme java est proposé par [56]. Celui-ci définit un formalisme temporel dont une partie peut s'exprimer en JML. Différents opérateurs temporels définissent la partie de la trace d'exécution sur laquelle certaines propriétés doivent être vraies : after, before, until, unless, never, always etc... Ces opérateurs s'appuient sur :

- des événements (pour méthode m : m "appelée", m "retournée", etc...),
- des propriétés sous forme d'expression booléennes JML,
- des propriétés de trace (propriétés booléennes JML quantifiées à l'aide de always, eventually, never),

les méthodes supportent une propriété supplémentaire sur leur terminaison, elles peuvent être enabled ou not enabled :

- enabled : quand la méthode est appelée alors elle termine normalement
- not enabled : quand la méthode est appelée et se termine alors elle lance une exception,

ce qui dans l'exemple suivant de spécification, donne dans le cas d'une transaction :

```
after beginTransaction called
   (always beginTransaction not enabled
    unless abortTransaction called, commitTransaction called)
```

Cet exemple exprime le fait qu'une fois beginTransaction appelée celle-ci se termine toujours par une exception sauf si abortTransaction ou commitTransaction ont été appelées. Soit qu'une transaction ne peut se terminer sans que l'une des deux fins aient été appelée.

Le formalisme proposé par les auteurs permet la spécification en JML de propriétés de sûreté. Ils prévoient à plus long terme l'intégration de leur formalisme dans la grammaire JML et la mise en oeuvre d'outils de vérification associés tels que Jass, ou JavaPathExplorer.

JML présente du point de vue du formalisme des traits remarquables par son abord aisé pour le développeur. Si cet aspect peut être tempéré de remarques relatives aux risques de ce côté intuitif, il n'en reste pas moins essentiel. Il est d'autre part générique

et raffinable. Enfin sa vérification est intéressante car elle peut intervenir à différents moments du cycle de l'application. Une part peut en être faite en amont de l'exécution, réduisant d'autant les vérifications à effectuer lors de celle-ci.

A.2 Formalismes fonctionnels pour les composants

A.2.1 Plate-forme .Net

le ContractWizard ([53]) présente l'avantage de ne pas nécessiter le code source des classes cibles. Ainsi à partir d'un assembly (unité de déploiement .Net) il produit les wrappers Eiffel associés et les range dans un nouvel assembly qui prend alors la place du précédent. L'article ne précise pas si les options d'armement d'Eiffel et autres traits sont supportés. A priori le code du corps des assertions pourrait ne pas être en Eiffel, facilitant encore la prise en main par le développeur. Tran et Mingins ([103]) proposent de leur côté une implémentation de machine virtuelle qui présente les fonctionnalités suivantes :

- pré/post/invariant,
- prise en compte de la réentrance afin de ne pas vérifier les invariants sur les auto-appels de méthodes internes,
- prise en compte du sous typage en combinant les spécifications des types parents,
- prise en compte de valeur passée dans les clauses old,

A l'exécution le code décoré remplace le code standard juste avant l'intervention du JIT, le tissage des assertions et du code standard peut d'ailleurs attendre jusque là. Comme les assertions sont prises en compte au niveau de l'IL (bytecode), divers outils peuvent les générer aussi bien au niveau de l'IL que du code source (metadonnées etc...). Ce travail propose les fonctionnalités de bas niveau nécessaire à la mise en oeuvre de contrats par assertions exécutables directement intégrées à la machine virtuelle. Cette approche garantit une uniformité de la prise en compte des assertions. Néanmoins, elle supposerait pour l'utilisateur d'adopter une machine virtuelle donnée non microsoft, au risque de ne suivre les évolutions de celle de la plateforme .net qu'avec un certain décalage.

A.2.2 Plate-forme J2EE

Dans [104], àA la différence de l'approche par contrat traditionnelle, l'évaluation des préconditions n'est pas à la charge du client mais effectuée par le conteneur d'EJB. Bien qu'ils n'exposent pas le pas le formalisme qu'ils ont utilisé pour décrire ces conditions, cela permet aux auteurs d'appliquer la conception par contrat aux composants. Ils montrent comment cette démarche est suffisante pour mettre en oeuvre divers exemples de politiques de HCS dont ils ont besoin, telle la vérification de contraintes en amont de l'exécution d'une action critique (exemple de fonctionnement d'un système d'armes), la vérification à intervalles fixes de la consistance de l'état d'un jeu de composants, ou la mise en oeuvre d'un firewall logiciel en assurant dans tous les cas un retour d'erreur prédéfini au client.

A.2.3 L'approche par modèle de Barnett

Un exemple de programme modèle est le suivant :

```
class Idictionary_Contract implements Idictionary
    var map as Map of object to object
    var enums as Set of Ienumerator

    invariant null notin domain(map) and null notin enums
    set (key as object, value as object)
       step if key = null
    throw new ArgumentNullException()
    map(key):=value
      step forall e in enums
    e.Invalidate ()
      step return map(key)
....
```

Le mot clé step introduit une étape du programme modèle. En effet bien que l'exécution de ce dernier soit par défaut parallèle, des étapes peuvent être exécutées séquentiellement dans l'ordre dans lequel elles sont ainsi déclarées. Cela permet ainsi, à la manière des spécifications par boite grise, d'ordonner les comportements de l'opération. Ainsi dans l'exemple, la deuxième clause step est utilisée pour spécifier que tous les enumérateurs devront avoir été invalidés avant de passer à l'étape suivante.

```
MoveNext() as Boolean
    if not valid
         throw new InvalidOperationException()
    else
         choose e in unvisited
         unvisited := unvisited - { e }
         current := e
         return true
    ifnone
        current := null
        return false
```

L'opérateur choose permet d'exprimer le non déterminisme d'une spécification. Il permet ainsi de laisser libre l'algorithme de parcours la collection mais de fixer les règles de celui-ci.
Pour la vérification dynamique du code, le programme modèle, décrit en Asml, est transcrit en pre/post conditions et invariants placés sur les méthodes du programme, et en un automate exécutable de vérification du séquençage des messages alimenté par les assertions. Le bytecode du programme est alors modifié juste avant sa compilation en code exécutable. Bien que le langage Asml utilise le modèle de type de la plateforme .Net, il s'exécute dans un espace séparé de celui du programme contraint. D'autre part il se passe de fonction d'abstraction en autorisant les instructions abstraites à agir sur les variables de l'interface spécifiée pour rendre les effets d'une implémentation. Par là cette approche diffère de JML dans lequel il faut déclarer explicitement les variables de modèle.

Cette approche présente l'intérêt d'être générique. Par ailleurs sa mise en oeuvre est effectuée dynamiquement de manière non intrusive par le tissage du code juste avant la compilation de l'ensemble. Par contre elle impose un réel travail de reprogrammation pour produire le modèle.

A.2.4 Wright et Darwin

Wright

L'ADL Wright utilise les CSP pour spécifier le comportement de ses d'interfaces [38]. Par exemple le comportement d'un " Server " peut être décrit de la manière suivante :

```
role Server = (invoke?x -> return!y -> Server) [] f
```

Le rôle correspond à une interface du système. Server est un processus qui attend (" ? ") un message x , et retourne (" ! ") ensuite un message y. Alors, soit (choix noté " [] "), il boucle en attente d'un message x, soit il termine (" f "). Il existe la même description pour l'interface Client, et les deux comportements sont mis en relation par ce qui est nommé glue dans Wright qui correspond à la description du processus mettant en oeuvre les rôles Client et Server :

```
glue = (Client.request?x -> Server.request!x ->
Server.result?y -> Client.result!y -> glue) [] f
```

Le processus correspondant à la connexion du Client et du Serveur est obtenu par la mise en parallèle des trois processus : `client || glue || server`. Du point de vue des vérifications statiques, Wright autorise la vérification d'absence d'inter blocage ainsi d'autres tests de consistance. En particulier il permet l'évaluation de la compatibilité des parties connectées sur la base de la compatibilité de leurs protocoles d'échange de messages [38]. Ces tests sont réalisés à l'aide de FDR un model-checker commercial, pour lequel les outils de Wright produisent les entrées.

Darwin

Darwin est un ADL qui base la sémantique de ses descriptions structurelles sur le pi-calcul [47]. Dans cet ADL chaque composant est considéré comme une entité atomique d'exécution. La composition hiérarchique de composants est présentée comme la mise en parallèle des processus qu'ils représentent. Plus généralement une architecture Darwin se traduit en un programme de pi-calcul dont la validité est synonyme de celle de l'architecture. Ainsi Darwin base sa description architecturale sur l'échange de messages entre processus. Dans Darwin le comportement des composants est spécifié à l'aide d'une autre algèbre de processus nommé FSP (Finite State Processes) et de diagrammes de transitions d'état (Labelled Transition System Diagram) associées aux messages des FSP. L'exemple que les auteurs en donnent dans [V] est le suivant :

Composant Darwin :

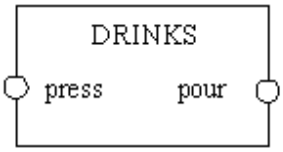


FIGURE A.1 – Composant Darwin

```
interface BUTTON {red; blue;}
interface BEVERAGE{coffee; tea;}
component DRINKS
{
   portal press:BUTTON;
   portal pour :BEVERAGE;
}
```

Spécification FSP :

```
DRINKS
   = (press.red ->pour.coffee->DRINKS
   | press.blue->pour.tea ->DRINKS) @ {press,pour}.
```

Spécification LTS :

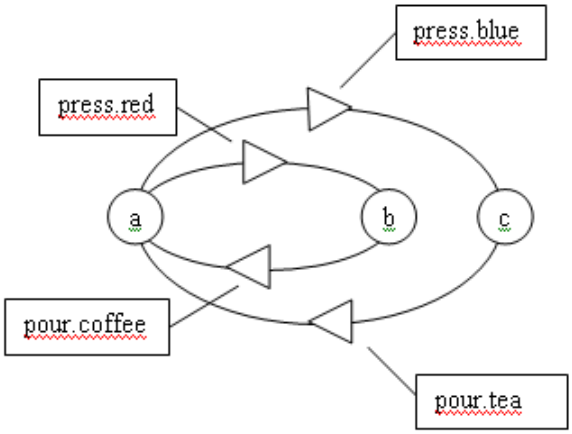


FIGURE A.2 – Darwin LTS

Dans cet exemple le composant DRINKS est vu comme un processus dont les comportements possibles sont complètement spécifiés au travers des actions (processus primitifs) qu'ils acceptent. L'action press.red entraine l'action pour.coffee et un retour à l'état normal, l'autre possibilité étant de commencer par l'action press.blue. Le diagramme intervient comme un outil de visualisation des comportements globaux qui d'après l'expérience des auteurs occupe un rôle important dans la vérification et le débuggage de la conception. Les auteurs de Darwin ont développé des techniques de vérification de propriétés de sûreté et de vivacité. Des outils logiciels permettent la mis en oeuvre de ces techniques pour la composition, l'analyse, le rendu des modèles.

A.2.5 Description Logic et Kind Theory

Dans l'approche de Description Logic, proposée par Devanbu et Borgida ([5]), les " concepts " associés à l'IDL, ont des constructeurs (opérateurs divers de composition), des attributs, ils supportent la satisfiabilité (sous-typage) par rapport à un autre " concept ". De plus attributs et concepts peuvent être composites. Les outils de raisonnement des DL sont alors capables d'évaluer la compatibilité de deux concepts ou la cohérence d'un à plusieurs concepts (un concept est dit incohérent si il ne peut être instancié). Par exemple, si on définit un objet CAR de la manière suivante :

```
CAR = (and (the couleurr RGB) (the typeRoue (maximum 4)))
```

Les outils associés détecteront l'incohérence de la spécification de sous-typage :

```
SUPER-CAR = (CAR and (the nbroues (maximum 6)))
```

Les auteurs expliquent qu'en mettant en oeuvre un " DL " pour décrire des pre/post conditions et invariants, il est possible de détecter des incompatibiltés ainsi que des incohérences dans les spécifications de manière statique.

```
interface Car
{
...
   void deliver (in CAR object, in MANUFACT src, in DEALER dest,
    in DATE time);
...
}
```

la méthode deliver peut être modélisée par :

```
DELIVER = (the object CAR) (the src MANUFACT) (the dest DEALER)
                     (the time DATE)
```

un concept à quatre attributs (object, src, dest, time) de types (CAR, MANUFACT, DEALER, DATE),qui peut associer au concept CAR associé à l'interface Car, une précondition décrite par :

```
CAR : (same-as
             deliver.object.location
             deliver.src.location)
```

Il devient alors possible de comparer les pre/post conditions et invariants d'interfaces requises ou fournies. Les DL apparaissent comme des langages de typage à mi chemin entre la compatibilité de spécifications syntaxiques et sémantiques. Leur expressivité est certes limitée mais ils offrent des outils de raisonnement et n'imposent pas une spécification complète.
Dans l'idée, le principe de la kind theory [57] est assez proche des Description Logic. Des données externes, les kinds, sont associées à des objets de génie logiciel, composants, interfaces, méthodes etc... La théorie de l'auteur comprend des opérateurs de

composition et sous-typage des kinds correspondant à la mise en oeuvre de leurs entités informatiques associées. Ainsi le kind d'une interface se déduit de celui de ses divers composants et il est possible de vérifier qu'il satisfait à celui d'une autre interface. Les auteurs montrent que dans la mesure où les kinds choisis représentent des propriétés sémantiques des entités logicielles, la satisfaction d'un kind par un autre entraîne la compatibilité sémantique des entités associées. Ils se servent de cette propriété pour mettre en oeuvre des adapters de connexion, car deux interfaces et méthodes de même kind sont alors compatible même si de noms différents. Les auteurs ont mis en oeuvre leur système dans le framework logique Maude et sont parvenus à automatiser des adaptations simples. Ils ont aussi appliqué leur théorie à la prise en compte de propriétés non fonctionnelles.

A.2.6 Approche assume-guarantee d'Abadi et Lamport

Nous allons brièvement décrire l'approche du paradigme hypothèse-garantie proposée par Abadi et Lamport. Elle présente l'intérêt de ne aps dépendre d'un formalisme particulier. Elle se caractérise par sa sémantique et repose sur les états du modèle auquel elle s'applique.

Sémantique. Un état décrit les parties observables de l'univers concerné, c'est à dire les interfaces des agents (par exemples des composants) considérés. L'état global se décompose en états attachés à chacun des composants, de sorte que l'état d'un composant ne soit accessible aux autres compsoants que via des interactions explicites. Nous faisons l'hypothèse à ce niveau abstrait que les composants interagissent en échangeant des signaux aux travers de leurs interfaces et que les interactions sont controlées soit par un composant soit par son environnement. Sémantiquement, un état est une assignation de valeurs à des variables, un comportement est une suite infinie d'états et une transition d'un comportement dénote une action dans laquelle un agent est responsable d'un changement d'état. Une propriété d'un comportement est vraie ou fausse, et est dite de sureté si elle est réfutable en un temps fini (par exemple le débit ne peut dépasser un certain seuil), est dite de vivacité si elle n'est jamais réfutable en un temps fini (par exemple une réponse sera finalement donnée).
Nous faisons l'hypothèse que les propriétés de sûreté sont suffisantes en première approche pour modéliser une large classe de contraintes de QoS ou de comportement. Une propriété de sureté contraint un composant si elle ne peut être violée que par ce composant, c'est à dire par une transition sous son contrôle, comme une action interne ou l'émission d'un signal, dans lequel il est responsable du changement d'état. Une spécification hypothèse-garantie attachée à un composant peut s'exprimer sous la forme $E \rightarrow M$, où M est une propriété de sûreté contraignant ce composant, E une propriété de sûreté contraignant son environnement, et $\rightarrow$ signifie que M doit rester vraie aussi longtemps que E. Cette relation temporelle est très utile pour les spécifications modulaires de composants, et pour distinguer les responsabilités des composants de celles de leur environnement.

Composition des spécifications. Il devrait être possible de prouver les spécifications (QoS ou de comportement) d'un système important sur la base des spécifications de ses composants. Le résultat suivant répond à ce besoin et constitue le théorème de composition établi par Abadi et Lamport, en considérant dans notre contexte orienté com-

posant seulement les propriétés de sûreté. Il se généralise aisément à n composants.

Théorème n°1. *Soit C_1 et C_2 deux composants dont les spécifications hypothèse-garantie sont les suivantes : $E_1 \rightarrowtriangle M_1$ et $E_2 \rightarrowtriangle M_2$, et E une hypothèse supplémentaire sur l'environnement des deux composants. Faisons l'hypothèse que M_1, M_2, E_1, E_2, E sont des propriétés de sûreté telles que :*

- *M_1 et M_2 contraignent respectivement C_1 et C_2,*
- *E_1, E_2 et E contraignent respectivement $\neg C_1$, $\neg C_2$ et $\neg(C_1 \& C_2)$,*

Alors la règle d'inférence suivante est valide :

$$\frac{E \cap M_1 \cap M_2 \subseteq \ E_1 \cap E_2}{(E_1 \rightarrowtriangle M_1) \cap (E_2 \rightarrowtriangle M_2) \subseteq (E \rightarrowtriangle M_1 \cap M_2)}. \tag{A.1}$$

Ce théorème fournit un critère de composabilité de C_1 et C_2, et une puissante technique pour prouver comment la spécification résultante peut être garantie. Plus précisément si la prémisse de la règle d'inférence est satisfaite, la composition de C_1 et C_2 satisfait à la spécification $E \rightarrowtriangle M_1 \cap M_2$. Cette spécification garantit que les propriétés de sûreté M_1 et M_2 doit rester satisfaite aussi longtemps que E. A l'exécution toute violation dénote une action observable (via une interface) par laquelle C_1, C_2 ou leur environnement est responsable du changement d'état.

A.3 Formalismes non fonctionnels pour les objets

A.3.1 QuO

Exemple de contrat :

```
contract repl_contract (
  syscond ValueSC ValueSCImpl ClientExpectedReplicas,
  callback AvailCB ClientCallback,
  syscond ValueSC ValueSCImpl MeasuredNumberReplicas,
  syscond ReplSC ReplSCImpl ReplMgr
) is
  negotiated regions are
  region Low_Cost : when ClientExpectedReplicas == 1 =>
     reality regions are
       region Low : when MeasuredNumberReplicas < 1 =>
       region Normal : when MeasuredNumberReplicas == 1 =>
       region High : when MeasuredNumberReplicas > 1 =>
     transitions are
       transition any->Low : ClientCallback.availability_degraded();
       transition any->Normal :
           ClientCallback.availability_back_to_normal();
       transition any->High :
          ClientCallback.resources_being_wasted();
     end transitions;
   end reality regions;
```

```
 region Available : when ClientExpectedReplicas >= 2 =>
    reality regions are
       region Low : when
           MeasuredNumberReplicas < ClientExpectedReplicas =>
       region Normal : when
           MeasuredNumberReplicas >= ClientExpectedReplicas =>
   transitions are
      transition any->Low : ClientCallback.availability_degraded();
      transition any->Normal :
           ClientCallback.availability_back_to_normal();
    end transitions;
 end reality regions;
 transitions are
   transition Low_Cost->Available :
      ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
   transition Available->Low_Cost :
      ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
 end transitions;
end negotiated regions;
end repl_contract;
```

Dans l'exemple donné les régions sont introduites par le mot clé region suivant de leur identifiant et de leur prédicat d'appartenance. On peut aussi constater que les régions sont composites. Parmi ces domaines le modèle distingue les attentes du client, les offres du fournisseur et des domaines de réalités qui sont relatifs aux données effectivement mesurées. La clause de transition entre régions est donnée au même niveau que celles-ci, sur la base de leurs identifiants. Les références aux entités nécessaires à la mise en oeuvre du contrat sont déclarées dans sa signature. Le lien avec l'implémentation n'est pas explicite au niveau du contrat. C'est à la charge du programmeur de fournir au contrat concret ces valeurs et pointeurs de méthodes lors de son évaluation.

Pour décrire les comportements d'adaptation de l'application, un autre langage, le Structure Description Language est utilisé. Il sert à décrire le comportement des délégués suivant les régions de qos. Ceux-ci sont en effet générés sur la base de behavior dont chacun décrit le comportement d'un ensemble d'interfaces, vis-à-vis d'un ensemble de régions de fonctionnement. Ainsi à chaque appel ou retour de méthode est associé pour chaque région de fonctionnement (définie dans un contrat) un comportement, qui peut être de quatre types :

- choix d'une méthode ou d'un objet d'implémentation autre que celle/celui de l'appel,
- levée d'une exception,
- connexion/déconnexion du délégué à autre objet distant,
- exécution d'une portion de code java ou c++,

Un exemple de spécification de SDL :

```
delegate behavior for Targeting and Replication is
    obj : bind Targeting with name SingleTargetingObject
```

```
    group : bind Targeting with characteristics { Replicated = True }

call calculate_distance_to_target :
  region Available.Normal :
     pass to group;
  region Low_Cost.Normal :
    pass to obj;
  region Available.Low :
    throw AvailabilityDegraded;
return calculate_distance_to_target :
    pass through;
default :
  pass to obj;

end delegate behavior;
```

Cet exemple décrit comment suivant les régions de qos, l'appel est transféré à l'objet group, sinon à l'objet obj et dans le cas de la région Available.Low une exception est levée. Dans le cas du retour d'appel, la valeur est retournée simplement. La clause default exprime que tous les autres appels sont transmis à obj. A la place des redirections de méthode, auraient pu se trouver des clauses de connexion/déconnexion du délégué à un objet distant (typiquement obj ou group), ou encore une portion de code java ou c++.
Des générateurs de code sont fournis pour traduire les QDL en contrats, délégués etc... Les contrats ne sont donc pas interprétés mais compilés ce qui ne va pas dans le sens de la dynamicité de leur mise en oeuvre. La modification du comportement d'un délégué nécessite alors production de code et recompilation. D'autre part l'utilisation de prédicats arbitraires pour décrire les régions de qos empêche leur comparaison, même si cela augmente l'expressivité du contrat et permet l'utilisation de toutes les grandeurs que peut fournir le développeur à son implémentation. Enfin l'association d'un contrat avec des entités logicielles n'est pas explicite et réalisé dans le code associé au contrat produit par le développeur.

A.3.2 QML

Chaque contrat QML contient une liste de contraintes dont chacune est associée à une dimension. Les opérateurs utilisables au sein de celles ci sont <, <=, ==, =>, >. Il est aussi possible de contraindre de la même manière une statistique de la dimension : quantile, moyenne, variance, ou fréquence tel que dans l'exemple de contrat vu précédemment. Les contrats peuvent se déduire les uns des autres par raffinement, en ajoutant ou redéfinissant des contraintes. La comparaison de la force de deux contraintes est utilisée pour définir une relation de conformité permettant de simplifier statiquement le contrat raffiné ou d'établir la compatibilité de deux contrats. Ainsi une spécification est dite conforme à une autre si elle est plus forte. La résolution de la comparaison de deux contraintes, repose sur l'association d'une "direction" de force à l'ordre des valeurs d'un domaine. Les mots clés increasing ou decreasing désignent une force croissante ou décroissante par rapport à la relation d'ordre définie. Ainsi un temps de réponse plus court est une contrainte plus forte qu'un long etc... Les contraintes, ajoutées à un contrat lors du raffinement, plus fortes que les originales

les remplacent. La relation peut aussi être utilisée pour évaluer le renforcement ou le relâchement de contraintes d'un contrat par rapport à un autre.

A.4 Formalismes non fonctionnels pour les composants

A.4.1 Le modèle des KComponent

Le métamodèle des KComponent fournit :

- le graphe de configuration,
- les protocoles et actions de reconfiguration de base agissant sur le graphe,
- la gestion de la reconfiguration dynamique intégrée au sein des programmes nommés contrats, situés au métaniveau.

De manière pratique, le graphe de configuration est déduit automatiquement d'un système construit de manière programmatique. La reconfiguration est transactionnelle, et gèle l'exécution et l'état des composants. Les contrats sont décrits dans un langage d'adaptation spécifique, diffèrent du langage de programmation des composants. Ils sont chargés/déchargés dynamiquement (en dehors des phases de configuration du système) par un "configuration manager" situé avec eux dans le métaniveau (Figure A.3).

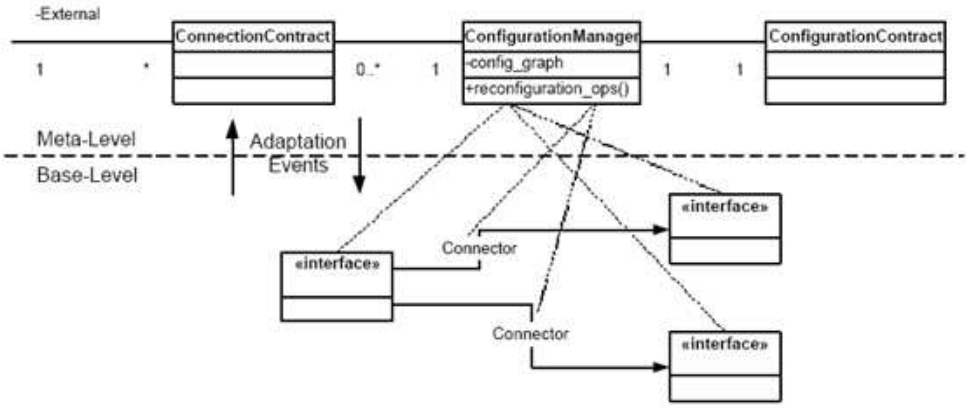


FIGURE A.3 – KComponent

Ces contraintes sont évaluées à la perception d'événements de reconfiguration du système, produisant ainsi un découplage entre niveau de base et méta et une possibilité de mise en oeuvre dynamique des contrats. Du point de vue de l'état de l'art, les règles dans les contrats d'adaptation sont interprétées comme une convention entre les participants (composants) au contrat.
Deux types de contrats d'adaptation sont distingués :

- configuration : spécifie des contraintes architecturales entre composants d'un même système (techniquement limité à un espace d'adressage)
- connexion : spécifie des contraintes sur une connexion individuelle entre le système et un composant dont il dépend et qui lui est extérieur. Techniquement il peut s'agir d'un composant dans un autre espace d'adressage.

Les auteurs décrivent les contrats de la manière suivante :

```
/* Automatically generated from XML configuration descriptor */
  configuration configuration* ::= configuration -descriptor.xml
  interface interface* ::= component-descriptor.xml

/* Automatically generated from component XML descriptors */
  component component-name* ::= component-descriptor.xml
  consumes user_defined_handler* ::= component-descriptor.xml
  emits adaptation_event* ::= component-descriptor.xml
  handler handler* ::= reconfiguration_op | user_defined_handler
  reconfiguration_op ::= change_component | change_configuration |
  install_interceptor | remove_interceptor | rollback |
  callback_client
/* User-defined below here*/
contract configuration configuration
{
  /* Conditional statements and reconfiguration operations */
}

contract connection connection-contract* ::=
  [component client_name] [interface port_name]
  [component provider_name]
{
  /* Conditional statements and reconfiguration operations */
}
```

A.4.2 CQML

La caractéristique de qos Les caractéristiques acceptent des paramètres et peuvent inclure dans leur définition des invariants, exprimés à l'aide de prédicats OCL. De plus une clause value : suivie d'une formule OCL, fonction de paramètres de la caractéristique, permet de fixer les valeurs possibles de la grandeur, par exemple :

```
quality_characteristic startUpTime (flow : Flow)
{
   domain : decreasing numeric milliseconds;
   values : if flow.SE->isEmpty() then invalid
     else flow.SE->first.time() - flow.initiate.time()
     endif;
   invariant : flow.initiate = invalid implies flow.SE->isEmpty
}
```

Enfin la composition de composants peut être prise en compte du point de vue de la qos. CMQL distingue la relation entre un composant utilisé par un autre et la relation entre plusieurs composants eux mêmes utilisés par un autre. Dans le premier cas la composition est modélisée par l'opérateur "sequential" indiquant la dépendance d'un composant par rapport à un autre. Dans le second cas les composants sont indépendants, néanmoins, vis à vis de leur utilisateur, ils sont susceptibles de devoir être utilisés conjointement ou non. Par exemple un client utilisant deux connexions

internet peut passer soit par l'une soit par l'autre, l'utilisation commune n'est pas obligatoire. Dans le cas d'un service de lecture de film, le composant de rendu d'image est à utiliser conjointement avec celui de son. Ces deux cas sont respectivement traités par les opérateurs parallel-or et parallel-and. sequential, parallel-or, parallel-and sont des opérateurs binaires.

```
quality_characteristic
{
...
  composition : parallel-and
                  left_component_charac.max(right_component_charac)
  parallel-or left_component_charac.min(right_component_charac)
  sequential left_component_charac + right_component_charac
}
```

Par ailleurs CQML permet de spécifier plus finement les caractéristiques que QML par l'utilisation de clauses values et invariant qui suivies d'une formule OCL décrivent les valeurs des caractéristiques ou les invariants qu'elles doivent respecter.

Le contrat Le corps de chaque contrainte est une expression OCL et sa validité peut être modulée dans le cas de contraintes statistiques qui requièrent de manière globale plusieurs mesures avant de pouvoir être décidées. Par exemple :

- best-effort : la contrainte n'est pas garantie,
- compulsory, threshold (appliqués à best-effort) with limit : respectivement, le service s'arrête si il ne peut maintenir une valeur donnée, et les clients d'un service (définis par l'intermédiaire du profile) sont prévenus quand une grandeur limite est atteinte par l'implémentation sous jacente du framework.

```
quality try_high
{
  compulsory best-effort frameOutput > 25 with limit 20;
  threshold best-effort delay < 5 with limit 8;
}
```

Dans cet exemple, le service s'arrêtera si il ne peut se maintenir au dessus de 20, par contre les clients seront simplement prévenus quand la limite de 8 est dépassée.
ou encore :

```
quality quickAndAlert (s : Stream) : fast
{
  s->forAll (f : Flow | startUpTime(f) < 100);
}

startUpTime étant une opération OCL.
```

Les profils La spécification des profils est moins fine que dans QML, CQML ne descend pas jusqu'au niveau de la méthode. Mais comme pour QML il est possible d'attacher plusieurs contrats à une même entité d'implémentation, permettant ainsi la séparation des préoccupations.
A la manière de QuO avec ses régions de fonctionnement, CQML fait reposer les négociations de QoS en phase de configuration ainsi que la gestion dynamique de celle-ci, sur la fourniture pour chaque profil d'un ensemble de sous profils imbriqués, requis ou fournis et nommés, par exemple :

```
profile adaptiveBinding for videoBinding
{
   profile good
   {
      uses high (incoming.videoFlow);
      provides high(outgoing.videoFlow) and
        fast(outgoing.videoFlow);
   }
   profile average
   {
      uses medium(incoming.videoFlow);
      provides medium(outgoing.videoFlow) and
        fast(outgoing.videoFlow);
   }
   transition any->average : incoming.some_callback("average");
   transition average->good :
   {
      incoming.some_callback("good");
      outgoing.another_callback("good");
   }
   precedence : good, average;
}
```

En phase d'exécution, la transition d'un profil à un autre peut être associée à des réactions du système (à travers des callbacks). Du point de vue de la composition des composants, les auteurs expliquent que les caractéristiques qos d'un assemblage peuvent se déduire de celles de ses composants mais placent ces travaux en dehors du champs d'étude de CQML.

A.4.3 CQML+

Cette extension de CQML offre encore dans le langage une prise en charge de la qos des ressources distincte de celle des composants. En effet le système de composants ne gère pas concrètement les ressources systèmes, à la différence des ressources des composants. Le langage CQML est aussi étendu par des caractéristiques de qos composites ex. :

```
resource cpu
{
   quality_characteristic cpu_demand (r: Resource)
   {
```

```
      domain: tupel
      {
         period (r),
         execution_time (r)
      };
      invariant: execution_time < period;
   }

   quality_characteristic execution_time (r: Resource)
   {
      domain: numeric real [0..) milliseconds;
   }

   quality_characteristic period (r: Resource)
   {
      domain: numeric real [0..) milliseconds;
   }
}
```

Dans cet exemple la caractéristique cpu_demand est composée des caractéristiques execution_time et period, et fait porter sur celles-ci un invariant.
Enfin l'extension permet de définir des dépendances explicites entre les clauses uses et provides des profils. En effet dans certains cas un profil peut accepter plusieurs régions de fonctionnement. Il est possible de les décrire à l'aide d'un ensemble de sous-profils présentant les caractéristiques uses et provides correspondantes. Mais il peut être plus efficace de relier les caractéristiques uses et provides du profil par une ou plusieurs équations. Finalement Les concepteurs de CQML travaillent sur son intégration au EJB, dans le cadre de laquelle la spécification CQML serait transcrite en XML.

A.5 Formalismes non fonctionnels pour les services

A.5.1 WSLA

Description de l'opération. Un exemple d'opération sera :

```
<Operation xsi:type="wsla:WSDLSOAPOperationDescriptionType"
   name="WSDLSOAPGetQuote">
   <SLAParameter name="AverageResponseTime"
      type="float"
      unit="seconds">
   ....
   </SLAParameter>
   ....
   <Metric name="SumResponseTimeTimeSeries" type="TS"
      unit="milliseconds">
     <Source>ACMEProvider</Source>
     <Function xsi:type="TSConstructor" resultType="TS">
        <Schedule>hourlyschedule</Schedule>
        <Metric>SumResponseTime</Metric>
        <Window>2</Window>
```

```
      </Function>
   </Metric>
   ....
   <!-- This part is the wsdl-specific definition
           of the operation. -->
   <WSDLFile>StockQuoteService</WSDLFile>
   <SOAPBindingName>GetQuoteSoapBinding</SOAPBindingName>
   <SOAPOperationName>getQuote</SOAPOperationName>
</Operation>
```

L'opération getQuote est ainsi associée au paramètre AverageResponseTime. Celui-ci doit être fourni par le ACMEProvider selon la métrique SumResponseTimeTimeSeries. En effet en plus des clients et fournisseurs du service, il est possible de faire intervenir des entités chargées de mesurer les valeurs.

ServiceLevelObjective Par exemple :

```
<ServiceLevelObjective name="g1">
   <Obliged>provider</Obliged>
   <Validity>
      <Start>2001-11-30T14:00:00.000-05:00</Start>
      <End>2001-12-31T14:00:00.000-05:00</End>
   </Validity>
   <Expression>
      <Predicate xsi:type="wsla:Less">
      <SLAParameter>AverageResponseTime</SLAParameter>
      <Value>5</Value>
      </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
</ServiceLevelObjective>
```

Dans cet exemple le parameter AverageResponseTime doit être inférieur à 5 sur la durée définie par le champs validity. L'expression est évaluée quand est détecté l'événement NewValue.

ActionGuarantee Par exemple :

```
<ActionGuarantee name="ga3">
   <Obliged>ZAuditing</Obliged>
   <Expression>
      <Predicate xsi:type="Violation">
         <ServiceLevelObjective>ga1</ServiceLevelObjective>
      </Predicate>
   </Expression>
   <EvaluationEvent>NewValue</EvaluationEvent>
   <QualifiedAction>
      <Party>XInc</Party>
      <Action actionName="notification" xsi:type="Notification">
         <NotificationType>Violation</NotificationType>
```

```
        <CausingGuarantee>ga1</CausingGuarantee>
        <SLAParameter>Availability_UpTimeRatio</SLAParameter>
      </Action>
   </QualifiedAction>
   <ExecutionModality>Always</ExecutionModality>
</ActionGuarantee>
```

Dans cet exemple la partie "ZAuditing" s'assure que le ServiceObjective "ga1" est vérifié à chaque événement "NewValue". Dans le cas ou "ga1" n'est pas vérifié alors "ZAuditing" notifie la partie "Xinc".

Annexe B — Plugins et spécialisation du système de contrat

Dans cette partie nous allons présenter les classes et mécanismes spécialisés par chacun des plugins, ainsi que les implémentations de ces derniers propres à Fractal et CCLJ.

B.1 Plugin d'architecture

Le plugin d'architecture a pour objet de fournir l'implémentation de classes génériques du noyau liées à l'architecture, adaptée à une architecture concrète. Les classes du noyau à spécialiser sont les suivantes :

pour la description de l'architecture :

- `ArchitectureDescription` : cette interface offre des fonctionnalités d'introspection de l'architecture du système contraint en des termes génériques, c'est à dire que les objets retournés sont des instances de `Reference` et `Relation`. Elle donne aussi accès au `RelationObserver`, moniteur de l'apparition et disparition de relations dans le système contraint. Elle permet aussi à l'utilisateur d'obtenir pour une référence issue de l'implémentation du système architectural, la `Reference` équivalente générique.

pour la désignation d'éléments d'architecture :

- `Reference` : chaque spécialisation de cette classe est destinée à désigner un élément d'architecture dans le système contraint (composant, méthode, interface, service etc...),
- `Link` et `Node` : ces classes servent à construire un chemin (alternance de `Link` et de `Node` chainés) décrivant une navigation, par rapport à un élément de départ dans le système contraint pour désigner un ensemble d'éléments d'architecture. Elles doivent être spécialisés pour tenir compte des différents types de relations qui existent dans le système contraint (enfants de `Link`, connexion, contenance etc...), et des différentes propriétés qui peuvent servir à filtrer les éléments du système (enfants de `Node`, nom de l'élément, etc...) à chaque étape de la navigation.

pour l'observation du système :

- `ObservationDescription` : la description d'une observation peut être spécialisée pour tenir compte de spécificités des classes `Trigger`, `Data`, `ObservationNotifier` propres à un type d'architecture donné,

- `Trigger` : la nature du déclencheur des observations est lié à celle des événements qui se produisent dans le système contraint, il doit donc être spécialisé pour chaque type d'événement différent et mettre en oeuvre les ressources appropriées d'observation du système concret (interception d'appel de méthode, de message non fonctionnel, etc...),
- `Data` : les données qui peuvent être lues sur un système dépendent de sa nature concrète, les ressources à mettre en oeuvre pour les obtenir dépendent naturellement aussi de l'implémentation du système observé. Il peut s'agir de paramètres échangés, de références à des éléments d'architectures, d'attributs de ces derniers etc...
- `ObservationNotifier` : permet d'enregistrer le lien entre un ou des observateurs (`Watcher`) et une requête d'observation (`ObservationDescription`) : suivant la nature des ressources mises en oeuvre dans le cadre des observations cet objet peut proposer des politiques d'économies de ces dernières en mutualisant les requêtes d'observation identiques faites par plusieurs observateurs, en relachant immédiatement une ressource d'observation dès lors qu'elle n'a plus d'observateurs etc...

pour la description du motif d'architecture :

- `Relation` : cette classe est utilisée pour décrire une configuration architecturale donnée, faite de `Reference` liées par des `Relation`. En particulier chaque instance de motif d'architecture est composée d'un ensemble de `Reference` et des relations `Relation`qui les relient.. Ainsi suivant la nature de l'architecture cette classe doit être spécialisée pour prendre en compte les différents types de relations (connexion, inclusion ...) apparaissant dans le système.

Fractal. Dans le cas du modèle de composants Fractal, ces classes sont spécialisées de la manière suivante :

description et désignation de l'architecture :

Nous avons montré, dans la partie "Mise en oeuvre du modèle de contrat", comment les classes `Reference` `Link` `Node` devaient être spécialisées :

- `Reference` : `FractalComponentReference` et `FractalInterfaceReference` possèdent une référence vers un composant et une interface Fractal, tandis que `FractalMethodReference` désigne une interface Fractal et une méthode (au sens java du terme),
- `Link` : les objets `Link` sont des objets fonctions qui partant d'une `Reference` retournent celles qui sont liées à celle-ci via une relation donnée. Ils utilisent les fonctionnalités d'introspection de la plate-forme Fractal pour obtenir les éléments connectés à un autre, inclus dans celui-ci ou l'incluant,
- `Node` : les spécialisations de `Node` pour Fractal sont aussi des objets fonctions capables de trier des `Reference` sur la base de certaines propriétés. Pour ce faire elles utilisent les attributs exposés par les éléments Fractal (composant et interface), ainsi que java (pour les méthodes). Les noms utilisés comme paramètres de filtrage par un `Node` peuvent être remplacés par le caractère générique "*",

- `ArchitectureDescription` : comme cette classe permet l'introspection du système contraint, elle utilise dans le cas de Fractal les fonctionnalités d'introspection de cette plate-forme,

observation du système :
Les observations du système sont spécialisées de deux points de vue que nous avons abordés dans la partie "Mise en oeuvre du modèle de contrat" et utilisent différentes ressources de la plate-forme Fractal :

- le déclencheur : les événements pris en compte consistent en l'interception d'appel de méthode, qu'il s'agisse de méthode d'une interface fonctionnelle ou d'une interface non fonctionnelle (par exemple la requête de démarrage d'un composant). Pour ce faire nous avons utilisé un framework d'observation de système de composants Fractal qui met en oeuvre des intercepteurs et des mixins, outils de tissage de code sur les composants. Ils permettent d'intercepter les appels sur, respectivement, les interfaces fonctionnelles et les interfaces de Controller. Intercepteurs et mixins sont des ressources de l'implémentation de référence Julia de Fractal,
- la valeur observée : elle est soit relative à l'exécution interceptée, soit relative à l'architecture contrainte. Ainsi il est possible d'obtenir les valeurs des paramètres ou de la valeur de retour de la méthode, ou bien d'obtenir une référence sur l'appelant, ou de résoudre un chemin donné sur l'architecture. Les valeurs relatives à l'exécution sont obtenues dans le cadre de l'interception, celles relatives à l'architecture sont issues de l'introspection de l'architecture Fractal.

`ContractManager` :
Dans le cas de l'application du système de contrat à la plate-forme Fractal, les `ContractManager` ne sont pas gérés par le `ContractSystemManagement`. Pour préserver la modularité des composants Fractal, chaque composant composite possède intègre sous forme de Controller le `ContractManager` qui va gérer les contrats entre ses sous composants. Il n'y a pas non plus de `ContractManager` spécifique pour gérer les contrats entre les composants de plus haut niveau car il n'y a qu'un seul composant de plus haut niveau qui contient le tout le système.

B.2 Plugin de formalisme

Comme le plugin d'architecture, ce plugin a pour objet de fournir une implémentation des classes génériques du noyau dédiée à un formalisme donné. Pour ce faire un certain nombre de classes doivent être spécialisées :

pour les spécifications :

- `ContractualSpecificationReferential` : cette classe doit être capable de lire un fichier de spécifications et de stocker ces dernières, pour les ressortir sous la formes des spécifications contractuelles associées à l'élément d'architecture qui lui est passé en argument. Elle doit donc traduire les spécifications en `ContractualSpecification`, c'est à dire en extraire les règles

(`RuleDescription`) qui les composent. Elle doit aussi stocker les spécifications de manière à en distinguer et extraire celles associées à un élément d'architecture donné.

pour les règles :

- `PredicateDescription` : chaque `RuleDescription` est composée d'une `PredicateDescription` et d'un certain nombre d'`ObservationDescription`. Cette classe est donc la description d'un prédicat, matérialisant une hypothèse ou une garantie, issue du formalisme pris en charge par le plugin. Elle doit contenir les informations suffisantes à la production d'une expression évaluable de la contrainte qu'elle décrit,
- `PredicateFactory` : il s'agit de l'usine à prédicats qui sont les formes évaluables des `PredicateDescription` qui servent de base à leur production. Suivant le formalisme différentes données ou mécanismes peuvent être nécessaire à la traduction du `PredicateDescription` en `Predicate` évaluable, par exemple les noms de variables utilisées dans le `PredicateDescription` peuvent devoir être transformés pour être évaluables etc... La `PredicateFactory` s'appuie pour ce faire sur le `PredicateEvaluator`,
- `PredicateEvaluator` : cette classe est l'évaluateur de prédicat et elle doit donc être spécialisée à l'aide des outils liés au formalisme du plugin. Elle peut présenter un certain nombre de services pour permettre l'adaptation d'une formule issue de la spécification en une forme évaluable (comme la traduction des noms de variables etc...),
- `PredicateEvaluatorFactory` : cette classe est l'usine qui produit les évaluateurs de prédicats, elle peut être spécialisée pour faire partager un même évaluateur à plusieurs prédicats, que ce soit pour des raisons d'économie de ressources ou parce que différents prédicats ont besoin d'être évalués dans des environnements d'évaluation identiques pour partager des variables etc...

pour les contrats :

- `ContractType` : les différents types de contrat sont obtenus en spécialisant cette classe, ils doivent notamment fournir le service de construction du contrat dont ils représentent le type,
- `Pattern` : de cette classe héritent les différents motifs d'architecture associés aux différents types de contrat, en particulier
- `ContractualSpecificationFilter` :

pour l'accès aux ressources du plugin :

- `FormalismPlugin`
- `FormalismToolkit`

Il faut noter de manière générale que le plugin de formalisme est dédié à un plugin d'architecture donné. En effet le plugin de formalisme fournit la fonctionnalité de traduction des spécifications d'un formalisme donné en règles telles que nous les avons décrites, et pour ce faire il doit associer à chacune des observations dont les prototypes lui sont fournis par le plugin d'architecture.

ANNEXE C Exemples de contrat de validation

C.1 Contrat basé sur les Behavior Protocol

Le contrat basé sur les spécifications décrites en Behavior Protocol est le suivant :

```
Contract
{
  Participants : <Facade>, <InstantGroup>, <InteractiveNotification> ;

  Clause:
  {
   responsible : <Facade>;

   assumption :
     On <Facade>
     Observe{
       val : evt.connectUser at : entry c.connectUser (...);
       val : evt.disconnectUser at : entry sns.disconnectUser ();
       }
     Verify : runtimeCheck ( (1),
     {"connectUser", "disconnectUser"});

   guarantee :
     On <Facade>
     Observe{
       val : evt.startUserCreation at : entry umt.startUserCreation();
       val : evt.closeUserCreation at : entry umt.closeUserCreation();
       val : evt.createUser at : entry umt.createUser(...);
       val : evt.searchGroup at : entry gm.searchGroup(...);
       val : evt.addUserToGroup at : entry gmt.addUserToGroup(...);

       val : evt.startUserSuppression
            at : entry umt.startUserSuppression(...);
       val : evt.closeUserSuppression
            at : entry umt.closeUserSuppression(...);
       val : evt.deleteUser at : entry umt.deleteUser(...);
       val : evt.getUserGroup at : entry gmt.getUserGroups(...);
       val : evt.removeUserFromGroup at : entry gmt.removeUserFromGroup
           (...);
       }
     Verify : runtimeCheck ( (1),
     {"startUserCreation", "closeUserCreation", "createUser",
       "searchGroup", "addUserToGroup",
       "startUserSuppression", "closeUserSuppression",
       "deleteUser","getUserGroups", "removeUserFromGroup"
     });
  }


  Clause :
```

```
{
 responsible : <InstantGroup>;

 assumption :
 On <InstantGroup>
   Observe{
     val : evt.startUserCreation at : entry umt.startUserCreation();
     val : evt.closeUserCreation at : entry umt.closeUserCreation();
     val : evt.createUser at : entry umt.createUser(...);
     val : evt.searchGroup at : entry gm.searchGroup(...);
     val : evt.addUserToGroup at : entry gmt.addUserToGroup(...);

     val : evt.startUserSuppression
          at : entry umt.startUserSuppression(...);
     val : evt.closeUserSuppression
          at : entry umt.closeUserSuppression(...);
     val : evt.deleteUser at : entry umt.deleteUser(...);
     val : evt.getUserGroup at : entry gmt.getUserGroups(...);
     val : evt.removeUserFromGroup at : entry gmt.removeUserFromGroup
         (...);
     }
   Verify : runtimeCheck ( (2),
   {"startUserCreation", "closeUserCreation", "createUser",
   "searchGroup", "addUserToGroup",
   "startUserSuppression", "closeUserSuppression",
   "deleteUser","getUserGroups", "removeUserFromGroup"
   });

 guarantee :
 On <InstantGroup>
   Observe{
     val : evt.notifyUserCreation
          at : entry ul.notifyUserCreation(...);
     val : evt.notifyUserAddedToGroup at : entry gl.
         notifyUserAddedToGroup(...);

     val : evt.notifyUserSuppression at : entry ul.
         notifyUserSuppression(...);
     val : evt.notifyUserRemovedFromGroup at : entry gl.
         notifyUserRemovedFromGroup(...);
     }
   Verify : runtimeCheck ( (2),
   {"notifyUserCreation", "notifyUserAddedToGroup",
     "notifyUserSuppression", "notifyUserRemoveFromGroup"});
 }

Clause :
{
   responsible : <InteractiveNotification>;

   assumption :
   On <InteractiveNotification>
   Observe{
     val : evt.notifyUserCreation
          at : entry ul.notifyUserCreation(...);
     val : evt.notifyUserAddedToGroup at : entry gl.
         notifyUserAddedToGroup(...);
```

```
        val : evt.notifyUserSuppression at : entry ul.
            notifyUserSuppression(...);
        val : evt.notifyUserRemovedFromGroup at : entry gl.
            notifyUserRemovedFromGroup(...);

        val : evt.registerAdvertisement
             at : entry a.registerAdvertisement(...);
        val : evt.registerContest at : entry a.registerContest(...);
        val : evt.registerStand at : entry a.registerStand(...);
        }
      Verify : runtimeCheck ( (3),
      {"notifyUserCreation", "notifyUserAddedToGroup",
        "notifyUserSuppression", "notifyUserRemoveFromGroup",
        "registerAdvertisement","registerContest", "registerStand"});
  }

  Agreement :
  {
    On <InstantFacade>
      Observe {
        val: evt.start at : entry lc.start ()
    }
    Verify : parallelCheck({ (1), (2), (3) });
  }
}
```

C.2 Contrat basé sur des spécifications CCLJ

A l'aide des spécifications en CCLJ nous allons pouvoir contraindre les grandeurs qui sont échangées par les composants. Nous allons en particulier nous intéresser aux échanges entre `<InstantGroup>` et `InteractiveNotification`. Nous considérons les spécifications suivantes :

- `<InstantGroup>`:

```
On <InstantGroup>
context void UserListener.notifyUserCreation (User user)
 pre : user.name != "" && user.getPhoneNumber().isValid() &&
          user.getKeywords.size() != 0;

context void GroupListener.notifyUserAddedToGroup (User user,
                          String groupId)
 pre : user.isRegistered() && gmt.isUserInGroup(user, groupId) ==
     true;

context void GroupListener.notifyUserRemovedFromGroup(User user,
                          String groupId)
 pre : user.isRegistered() && gmt.isUserInGroup(user, groupId) ==
     false;
```

  Les deux dernières spécifications mettent en avant un point important de notre approche des contrats, qui rejoint une caractéristique de CCLJ. Nous ne considérons pas un composant comme un ensemble d'interfaces séparées mais comme un tout au sein duquel les interfaces peuvent collaborer. Ainsi la spécification de

l'interface `GroupListener` peut s'appuyer sur l'interface `GroupManagement`.

- `<InteractiveNotification>`

```
On <InteractiveNotification>
 context void UserListener.notifyUserCreation (User user)
  pre : user.name != "" && user.phoneNumber.isValid();

 context void GroupListener.notifyUserAddedToGroup (User user,
                                String groupId)
  pre : user.isRegistered() && groupId != "";

context void GroupListener.notifyUserRemovedFromGroup(User user,
                                String groupId)
  pre : user.isRegistered() && groupId != "";
```

Le contrat basé sur les spécifications CCLJ est le suivant :

```
Contract
{
  Participants : <Facade>, <InstantGroup>, <InteractiveNotification> ;

  Clause:
  {
   responsible : <InstantGroup>;

   guarantee :
     On <InstantGroup>
     Observe
     {val : user at : entry ul.notifyUserCreation(User user);}
     Verify : user.name()!=""&&user.getPhoneNumber().isValid()
                  &&user.getKeywords.size()!=0;
  }

  Clause:
  {
   responsible : <InstantGroup>;

   guarantee :
     On <InstantGroup>
     Observe
     {val : user, groupId
            at : entry ul.notifyUserAddedToGroup(User user, String
                groupId);}
     Verify : user.isRegistered&&gmt.isUserInGroup(user,groupId)==true;
  }

  Clause:
  {
   responsible : <InstantGroup>;

   guarantee :
     On <InstantGroup>
     Observe
     {val : user, groupId
             at : entry ul.notifyUserRemovedFromGroup(User user, String
                groupId);}
```

```
    Verify : user.isRegistered&&gmt.isUserInGroup(user,groupId)==false;
}

Clause:
{
 responsible : <InteractiveNotification>;

 assumption :
   On <InteractiveNotification>
   Observe
   {val : user at : entry ul.notifyUserCreation(User user);}
   Verify : user.name()!=""&&user.getPhoneNumber().isValid();
}

Clause:
{
 responsible : <InteractiveNotification>;

 assumption :
   On <InstantGroup>
   Observe
   {val : user, groupId
          at : entry ul.notifyUserAddedToGroup(User user, String
              groupId);}
   Verify : user.isRegistered&&groupId !== "";
}

Clause:
{
 responsible : <InteractiveNotification>;

 assumption :
   On <InteractiveNotification>
   Observe
   {val : user, groupId
         at : entry ul.notifyUserRemovedFromGroup(User user, String
             groupId);}
   Verify : user.isRegistered&&groupId !== "";
}

Agreement :
{
  On <InstantGroup>
    Observe
    {val : user at : entry ul.notifyUserCreation(User user); }
    Verify : (user.name()!=""&&user.getPhoneNumber().isValid()
                &&user.getKeywords.size()!=0)
               => (user.name()!=""&&user.getPhoneNumber().isValid());

  On <InstantGroup>
    Observe
   {val : user, groupId
          at : entry ul.notifyUserAddedToGroup(User user, String
              groupId);}
    Verify :
         (user.isRegistered&&gmt.isUserInGroup(user,groupId)==true)
         => (user.isRegistered&&groupId !== "");
```

```
    On <InstantGroup>
      Observe
     {val : user, groupId
            at : entry ul.notifyUserRemovedFromGroup(User user, String
                groupId);}
      Verify :
           (user.isRegistered&&gmt.isUserInGroup(user,groupId)==false)
            => (user.isRegistered&&groupId !== "");
  }
}
```

ANNEXE D

Publications

Au cours de cette thèse j'ai co-écrit plusieurs articles dont voici la liste :

[201] Philippe Collet and Alain Ozanne. Un système de contractualisation pour Fractal : intégration et retours sur expérience, In *Journées Composants 2005* Le Croisic (France).

[202] Philippe Collet, Alain Ozanne and Nicolas Rivierre. Enforcing Different Contracts in Hierarchical Component-Based Systems, In *5th International Symposium on Software Composition, SC'06 (ETAPS satellite event)* (25-26 March 2006), Vienna (Austria), Lecture Notes in Computer Science, Springer Verlag.

[203] Philippe Collet, Alain Ozanne and Nicolas Rivierre. On contracting different behavioral properties in component-based systems, In *SAC* (2006), Hisham Haddad Ed., ACM, pp 1798–1799.

[204] Hervé Chang, Philippe Collet, Alain Ozanne and Nicolas Rivierre. From Components to Autonomic Elements using Negotiable Contracts, In *3rd International Conference on Autonomic and Trusted Computing (ATC'06)* (3-7 September 2006), Lecture Notes in Computer Science, Springer Verlag.

[205] Hervé Chang, Philippe Collet, Alain Ozanne and Nicolas Rivierre. Some Autonomic Features of Hierarchical Components with Negotiable Contracts In *3rd IEEE International Conference on Autonomic Computing (ICAC'06)* (12-16 June 2006), Dublin (Ireland) IEEE Computer Society Press.

[206] Philippe Collet, Alain Ozanne and Nicolas Rivierre. Towards a Versatile Contract Model to Organize Behavioral Specifications, In *33rd International Conference on Current Trends in Theory and Practice of Computer Science SOFSEM* 2007, Harrachov (Czech Republic).

[207] Philippe Collet, Jacques Malenfant, Alain Ozanne and Nicolas Rivierre. Composite Contract Enforcement in Hierarchical Component Systems In *ETAPS 2007, 6th International Symposium on Software Composition (SC 2007)* Lecture Notes in Computer Science, Springer Verlag.

J'ai aussi participé à plusieurs livrables du projet RNTL Faros ("Composition de contrats pour la Fiabilité des ARchitectures Orientées Services") :

[301] Philippe Collet, Fabien Balligand, Hervé Chang, Pierre Combes, Laurence Duchien, Alain Ozanne, Anne-Marie Pinna-Déry, Nicolas Rivierre, Roger Rousseau, Bruno Traverson. Chapitre Contrats dans Livrable Faros : État de l'art sur la contractualisation et composition Research Report 23-74 pages, oct 2006.

[302] Anne-Françoise Le Meur, Philippe Lahire, Guillaume Waignier, Philippe Collet, Noël Plouzeau, Michel Dao, Laurence Duchien, Alain Ozanne, Nicolas Rivierre. Spécification d'une architecture pour la contractualisation de services : expression du besoin Research Report RNTL Faros, number F-1.2., 1-30 pages, jun 2007.

[303] Fabien Balligand, Mireille Blay-Fornarino, Hervé Chang, Daniel Cheung-Foo-Wo, Philippe Collet, Guillaume Dufrêne, Vincent Hourdin, Stéphane Lavirotte, Sébastien Mosser, Alain Ozanne, Anne-Marie Pinna-Déry, Nicolas Rivierre, Lionel Seinturier, Jean-Yves Tigli. Identification des modalités de prise en charge des contrats pour chaque plate-forme cible Research Report RNTL Faros, number F.3.1, 2007.

J'ai enfin participé à la rédaction d'un chapitre d'un livre :

[401] Thierry Coupaye, Pierre-Charles David, Bruno Dillenseger, François Horn, N. Jayaprakash, Alain Ozanne, Nicolas Rivierre. Les caractéristiques architecturales de l'autonomique. In *L'autonomie dans les réseaux* Francine Krief and Mikael Salaun. Hermes Science Publications, 2006.

Bibliographie

[1] W. Aalst. Don't go with the flow : Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1) :72–76, 2003.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1) :73–132, jan 1993.

[3] J. Aegedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University Of Oslo, 2001.

[4] M. Aiello, G. Frankova, and D. Malfatti. What's in an agreement ? an analysis and an extension of WS-agreement. In *ICSOC*, pages 424–436, 2005.

[5] P. D. Alex Borgida. Adding more "dl" to idl : towards more knowledgeable component interoperability. In *International Conference on Software Engineering*, 1999.

[6] L. F. Andrade and J. L. Fiadeiro. Interconnecting objects via contracts. In *UML*, pages 566–583, 1999.

[7] A. Andrieux and al. Web services agreement specification (ws-agreement). In *Global Grid Forum*, 2004.

[8] J. M. T. Antonio Vallecelio, Juan Hernandez. Component interoperability. Technical Report ITI-2000-37, université de Malaga, 2000.

[9] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii : Technical concepts of component-based software engineering. Technical Report SEI-2000-TR-08, Carneggie Mellon University, Pittsburgh, 2000.

[10] R.-J. Back and J. W. von. Contracts, games and refinement. Technical Report TUCS-TR-138, Turku Centre for Computer Science, Finland, 1997.

[11] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2–3) :131–146, May 2005.

[12] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – java with assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, july 2001.

[13] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[14] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Series in Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., 1991.

[15] J.-Y. L. Boudec. The asynchronous transfer mode : A tutorial. *Computer Networks and ISDN Systems*, 24(4) :279–309, 1992.

[16] L. Briand, W. J. Dzidek, and Y. Labiche. Using aspect-oriented programming to instrument ocl contracts in java. Technical Report SCE-04-03, Software Quality Laboratory, Carleton University, 2004.

[17] E. Bruneton, T. Coupaye, and J.-B. Stefani. The fractal component model. Technical report v1, v2, The ObjectWeb Consortium, 2002,2003.

[18] L. Burdy and al. An overview of jml tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[19] P. Chalin. Improving jml : For a safer and more effective language. Technical report, Concordia University, 2003.

[20] H. Chang and P. Collet. Négociation de contrats : des systèmes multi-agents aux composants logiciels. *RSTI - Série L'Objet (RSTI-Objet)*, 12(4), Dec. 2006.

[21] H. Chang and P. Collet. Compositional Patterns of Non-Functional Properties for Contract Negotiation. *Journal of Software (JSW)*, 2(2) :12, Oct. 2007.

[22] P. Collet, A. Ozanne, and N. Rivierre. Enforcing different contracts in hierarchical component-based systems. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 50–65, Vienna, Austria, 2006. Springer.

[23] P. Collet and R. Rousseau. Etat de l'art des approches contractuelles. Technical report, France Telecom, I3S, 2003.

[24] P. Collet, R. Rousseau, T. Coupaye, and N. Rivierre. A contracting system for hierarchical components. In G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2005.

[25] A. Dan, H. Ludwig, and G. Pacifici. Web services differentiation with service level agreements. Technical report, IBM, May 2003.

[26] P.-C. David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Université de Nantes / École des Mines de Nantes, 2005.

[27] C. DellaTorreCicalese and S. Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7) :46–53, 1999.

[28] V. Dignum, J.-J. C. Meyer, Frank, Dignum, and H. Weigand. Formal specification of interaction in agent societies. In *FAABS*, pages 37–52, 2002.

[29] A. Duncan and U. Holzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA, Dec. 1998.

[30] F. P. Dusan Balek. Software connectors and their role in component deployment. In *DAIS'01*, 2001.

[31] O. Enseling. icontract : Design by contract in java. *JavaWorld*, 02 2001. http ://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

[32] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, pages 1–15, 2001.

[33] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC / SIGSOFT FSE*, pages 229–236, 2001.

[34] R. J. Frantisek Plasil, Dusan Balek. Sofa/dcup : Architecture for component trading and dynamic updating. *ICCDS, Annapolis, Maryland, USA*, 1998.

[35] L.-A. Fredlund. Implementing ws-cdl. In *JSWEB 2006 (II Jornadas Científico-Técnicas en Servicios Web)*, Santiago de Compostelle, Espagne, 2006.

[36] S. Frolund and J. Koistinen. Quality of service aware distributed object systems. Technical Report HPL-98-142, Hewlett Packard, Software Technology Laboratory, Aug. 1998.

[37] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, 1998.

[38] A. D. Fuxman. A survey of architecture description languages. In *Reports from CSC2108 Automatic Verification*. University of Toronto, 2000.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[40] C. Garion and L. van der Torre. Design by contract deontic design language for multiagent system. In *ANIREM*, 2005.

[41] T. Genssler and C. Zeidler. Rule-driven component composition for embedded systems, 1999.

[42] J. V. Guttag. Abstract data type and the development of data structures. *Communications of the ACM*, 20(6) :396–404, 1977.

[43] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts : Specifying behavioral compositions in object-oriented systems. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 169–180, Oct. 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[44] T. Henzinger, S. Qadeer, and S. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In I. C. S. Press, editor, *International Conference Computer-aided Design*, pages 245–252, 2000.

[45] C. Hoare. An axiomatic basis for computer programming. In *ACM 12*, pages 576–580, 1969.

[46] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting ocl. In A. Evans, S. Kent, and B. Selic, editors, *Proc. 3rd International Conference on the Unified Modeling Language (UML)*, volume 1939 of *LNCS*, pages 278–293. Springer-Verlag, 2000.

[47] S. E. Jeff Magee, Naranker Dulay and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conference ESEC'95*, 1995.

[48] V. C. Jim Dowling. The k-component architecture meta-model for self-adaptive software. In *Reflection 2001, The Third Int. Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, 2001.

[49] J. Jin and K. Nahrstedt. Qos specification languages for distributed multimedia applications : A survey and taxonomy. *IEEE MultiMedia*, 11(3) :74–87, 2004.

[50] C. Jones. Tentative steps toward a development method for interfering programs. In *ACM Transactions on Programming Languages and Systems*, pages 569–619, 1983.

[51] J.-M. Jézéquel, O. Defour, and N. Plouzeau. An MDA approach to tame component based software development. In *Formal Methods for Components and Objects : Second International Symposium, FMCO 2003*, pages 260–275, 2003.

[52] M. Karaorman, U. Hölzle, and J. Bruno. jcontractor : A reflective java library to support design by contract. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection*, volume 1616 of *LNCS*, pages 175–196, Berlin, 1999. Springer Verlag.

[53] R. S. Karine Arnout. the .net contract wizard : Adding design by contract to languages other than eiffel. *IEEE*, 2001.

[54] A. Keller, G. Kar, H. Ludwig, A. Dan, and J. Hellerstein. Managing dynamic services : A contract-based approach to a conceptual architecture. In *Proc. NOMS 2002 : 8th Network Operations and Management Symposium, Florence, Italy, 15-19 Apr. 2002.*, 2002.

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

[56] M. H. Kerry Trentleman. Extending jml specifications with temporal logic. In *Algebraic Methodology and Software Technology (AMAST)*, 2002.

[57] J. Kiniry. Semantic component composition. *CoRR*, cs.SE/0204036, 2002.

[58] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained : The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.

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

[60] R. Kramer. icontract – the java design by contract tool. In *Proceedings of TOOLS 26 : Technology of Object-Oriented Languages and Systems*, page 295, Santa Barbara, CA, august 1998. IEEE.

[61] L. Lamport and F. B. Schneider. The "hoare logic" of csp, and all that. In *ACM Transaction on Programming Language and Systems*, pages 281–296, 1984.

[62] R. Land. A brief survey of software architecture. Technical report, Malardalen University, 2002. Available : citeseer.nj.nec.com/land02brief.html.

[63] G. T. Leavens and Y. Cheon. Design by contract with jml. Draft, available from jmlspecs.org., 2005.

[64] G. T. Leavens, Y. Cheon, C. Clifton, and al. How the design of jml accomodates both runtime assertion checking and formal verification. In *FMCO 2002*, 2002.

[65] J. P. Loyall, D. E. Bakken, R. E. Schant, and al. Qos aspect languages and their runtime integration. In *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering*, 2001.

[66] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. Web service level agreement (wsla) language specification, version 1.0. Technical report, IBM, 2003.

[67] J. Magee and J. Kramer. *Concurrency : state models & Java programs*. John Wiley & Sons, Inc., 1999.

[68] J. Malenfant, N. Plouzeau, and J.-M. Jézéquel. The design of gccl : a generalized common contract language. Technical Report RR-4502, INRIA, Rennes, July 2002.

[69] W. W. Martin Büchi. The greybox approach : when blackbox specifications hide too much. Technical report, TUCS, 1999.

[70] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. Technical report, University of California, 1997.

[71] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10) :40–51, Oct. 1992.

[72] B. Meyer. *Eiffel : the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[73] W. S. Mike Barnett. Contracts, components, and their runtime verification on the .net platform. Technical report, Microsoft Research, 2002.

[74] N. Milanovic, V. Stantchev, and M. Malek. Definition of contracts for service composition. 2004.

[75] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4) :417–426, jul 1981.

[76] D. J. Murray and D. E. Parson. Automated debugging in java using ocl and jdi. In M. Ducasse, editor, *proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, Munich, August 2000.

[77] L. P. Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[78] S. P. Nikunj R. Mehta, Nenad Medvidovic. Towards a taxonomy of software connectors. *ICSE*, 2000.

[79] OASIS. Reference model for service oriented architecture version 1.0. Technical report, OASIS, 2006.

[80] Object Modeling Group. *Object Constraint Language Specification, version 1.1*, september 1997.

[81] *UML 2.0 Object Constraint Language (OCL) Specification*, 2003. http ://www.omg.org/docs/ptc/03-10-14.pdf.

[82] OMG. Uml 2.0 ocl specification. Technical report, OMG, 2003.

[83] OMG. OMG model driven architecture. `http://www.omg.org/mda/`, May 2006.

[84] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10) :46–52, 2003.

[85] H. Peng and S. Tahar. A survey on compositionnal verification. Technical report dept of ece, Concordia University, 1998.

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

[87] L. Petre, R.-J. Back, and I. Paltor. Analysing uml use cases as contracts. In *UML*, pages 518–533, 1999.

[88] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng*, 28(11), 2002.

[89] R. Plösch. Evaluation of assertion support for the java programming language. *Journal Of Object Technology*, 2002.

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

[91] R. H. Reussner. An enhanced model for component interfaces to support automatic and dynamic adaption. In J. Hernandez, A. Vallecillo, and J. M. Troya, editors, *New Issues in Object Interoperability – Proceedings of the ECOOP' 2000 Workshop on Object Interoperability*, pages 33–42, 6 2000. Published by Universidad de Extremadura Dpto. Informatica.

[92] M. Richters and M. Gogolla. A metamodel for ocl. In *UML*, pages 156–171, 1999.

[93] R. M. Robert T. Monroe, Andrew Kompanek and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 1997.

[94] S. Röttger and S. Zschaler. CQML+ : Enhancements to CQML. In J.-M. Bruel, editor, *1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56, Toulouse, France, 2003. Cépaduès-Éditions.

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

[96] M. Sereno. *Performance Models for Discrete Event Systems with Synchronisations : Formalisms and Analysis Techniques*, chapter Modelling and analysis of time constrained and hierarchical systems (MATCH). 1998. ERB Project CHRX-CT-940452.

[97] A. Sjörgen. A method for support design by contract on the .net platform. Technical report, 2002.

[98] E. W. Stark. A proof technique for rely/guarantee properties. In Springer-Verlag, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206, pages 369–391, Berlin, 1985.

[99] K. Stolen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical Report SFB 342/2/93, Technische Universität München, 1993.

[100] Sun. *Programming With Assertions*. http ://-java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html.

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

[102] H.-M. Tran, P. Bedu, L. Duchien, H.-Q. Nguyen, and J. Perrin. Toward structural and behavioral analysis for component models. In *SAVBCS 2004, 12th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, NewPort Beach, California, USA, November 2004.

[103] N. Tran, C. Mingins, and D. Abramson. Design and implementation of assertions for the common language infrastructure. *IEEE*, 2003.

[104] G. J. Vecellio, W. M. Thomas, and R. M. Sanders. Containers for predictable behavior of component-based software. In *Proceedings of the Fifth ICSE Workshop on Component Based Software Engineering*, May 2002.

[105] W3C. Web service architecture. Technical report, W3C, February 2004. http ://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[106] W3C. Xml path language 2.0. http ://www.w3.org/TR/xpath20/, january 2007.

[107] Q. Xu and S. Mohalik. Compositional reasoning using the assumption-commitment paradigm. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 565–583. Springer, 1997.

[108] J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Inf. Syst*, 29(2) :97–125, 2004.

[109] P. Ziemann and M. Gogolla. An extension of ocl with temporal logic. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 53–62. Technische Universität München, Institut für Informatik, 2002.

[110] M. Zulkernine and R. E. Seviora. Assume-guarantee supervisor for concurrent systems. In *IPDPS*, page 151, 2001.