REPUBLIQUE ALGERIENNE DEMOQRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR

ET DE RECHERCHE SCIENTIFIQUE

Université Saad Dahleb Blida 1

Faculté des Sciences

Département d'Informatique

Mémoire présenté par :

M^r Mansour Abdeselem

M^r Debbache Faouzi

En vue d'obtenir le diplôme de Master

Domaine : Mathématique et Informatique

Filière : Informatique

Spécialité : Ingénierie de logiciel

Sujet:

Expression des contraintes OCL dans la spécification de mission d'un système de systèmes

Soutenu le septembre 2017 devant les jurys composé de :

M^me Cherfa.I	promotrice.
M^me Chikhi.I	Président.
M^me Guessoum.D	Examinateur.

Remerciements

*Cet opuscule serait incomplet si nous ne témoignerons pas de notre profonde gratitude envers grand **DIEU** qui nous a conduits à réaliser l'un de nos rêves par sa bienveillance et sa générosité.*

Toutes les Expressions de remerciements ne suffiront jamais pour traduire notre reconnaissance et notre profonde gratitude.

*On Remercie **M^{me} Cherfa.I** notre promotrice qui nous a été le guide durant cette année, on la remercie de nous avoir fait confiance et de nous avoir donné le courage et la force pour vaincre toutes les difficultés.*

On remercie infiniment Nos deux familles pour leur compréhension et leur présence tout au long de cette dure année.

Nos remerciements s'adressent aux membres du jury qui ont bien accepté de juger et d''évaluer ce travail.

Nos remerciements s'adressent à tous les enseignants qui ont contribué à notre formation durant les cinq ans d'études à cette université.

Merci.

Table des Matières

Chapitre III. La Méta-Modélisation

PARTIE II : CONRIBUTION

Chapitre IV. Méta-Modèle pour La Spécification des Contraintes et des Missions

Chapitre V. Implémentation

Liste des Figures

Liste des Tableaux

ملخص

ويمكن فهم نظام النظم على أنه نتيجة للتفاعل بين النظم القائمة المستقلة وغير المتجانسة القائمة التي تتعاون لتشكيل نظام أكبر وأكثر تعقيدا لإنجاز مهمة ما. والهدف من هذا العمل هو اقتراح تدوين يسمح للتعبير عن القيود أوكل بيانيا عند تحديد مهمة من نظام الخدمة الاجتماعية (سوس).
تم التعبير عن بناء الجملة المجردة بواسطة نموذج ميتا، في حين تم التعبير عن بناء الجملة الخرسانية باستخدام عناصر الرسم. قمنا بتنفيذ أداة مواصفات باستخدام إمف (إكليبس إطار النمذجة) والتحقق من صحة الأداة من خلال دراسة حالة.

الكلمات الرئيسية: أنظمة الأنظمة، كائن تقييد اللغة، المهمة، النموذج الفوقي، القيد

Résumé

Un système de systèmes peut être compris comme le résultat de l'interaction entre les systèmes constitutifs indépendants et hétérogènes existants qui coopèrent pour former un système plus vaste et plus complexe permettant d'accomplir une mission. Le but de ce travail est de proposer une notation permettant d'exprimer des contraintes OCL graphiquement lors de la spécification d'une mission d'un SoS (System of System).
La syntaxe abstraite a été exprimé par un méta-modèle, tandis que la syntaxe concrète a été exprimée à l'aide d'éléments graphiques. Nous avons implémenté un outil de spécification à l'aide d'EMF (Eclipse Modeling Framework)-GMF (Graphical Modeling Framework) et nous avons validé l'outil par une étude de cas.

Mots Clés : Systèmes de Systèmes SoS, Object Constraint Language OCL, Mission, Méta-modèle, contrainte.

Introduction Générale

Contexte Générale

Les systèmes de systèmes sont des systèmes qui nécessitent des adaptations récurrentes en raison de leur environnement changeant. De tels systèmes doivent disposer d'architectures facilitant le travail de l'architecte lors de l'adaptation. La complexité du travail de l'architecte réside dans le fait qu'il doit modifier l'architecture du système tout en respectant sa mission. Cependant, la mission du système est définie au stade de la spécification des exigences et n'est souvent pas explicite dans l'architecture. Ainsi, elle devient inaccessible à l'architecte chargé de l'adaptation.

Problématique

Lors de l'accomplissement d'une mission, un système de systèmes peut être soumis à des contraintes opérationnelles et environnementales pouvant affecter positivement ou négativement sa mission. Il est donc indispensable de spécifier ce genre de contraintes dans les premières phases de construction d'un SoS. Cependant, les techniques actuelles ne permettent pas de spécifier les contraintes dans la spécification de la mission d'un système de systèmes.

Objectif

L'objectif de ce travail est de définir un méta-modèle qui permet de prendre en charge la définition de contraintes associées à une mission. Le langage OCL (Object Constraint Language) peut être utilisé à la base pour définir ces contraintes.

Organisation du mémoire

Le présent mémoire est composé de cinq chapitres structuré en deux parties :

- La première partie intitulée état de l'art est composé de trois chapitres :
  - Nous présentons dans le premier chapitre les systèmes de systèmes et leurs caractéristiques ainsi que les différents types, nous présentons aussi quelques langages pouvant être utilisés pour définir les missions dans les SoS et on aborde leurs limites.
  - Dans le deuxième chapitre nous présentons les classifications des contraintes, ainsi que le langage OCL et quelques formalismes graphiques d'expression de contraintes.

Partie I

ETAT DE L'ART

Chapitre I

Les Systèmes de Systèmes

I. Introduction

La complexité croissante des organisations humaines complémentées par leurs outils technologiques et la nécessaire collaboration entre ces organisations repose aujourd'hui sur un véritable maillage informationnel de la société. Il a conduit à interconnecter ces systèmes pour les faire collaborer, ce qui conduit au concept de système de systèmes (SoS).
Dans ce premier chapitre nous allons parler des systèmes de systèmes, de leurs types et caractéristiques, ainsi sur leurs missions

II. Les systèmes de systèmes

1. Définitions de système de systèmes

Il n'y a pas encore une définition unique généralement acceptée pour les SoS (System of Systems), il existe dans la littérature plusieurs définition des SoS, nous citons celles qui sont les plus utilisées.

1.1. Définition 1

L'approche système de systèmes est une méthode pour visant le développement, l'intégration, l'interopérabilité et l'optimisation des systèmes pour améliorer la performance dans un domaine d'application [1].

1.2. Définition 2

On parle de système de systèmes lorsqu'il y' a la présence de la majorité des cinq caractéristiques suivantes : l'indépendance opérationnelle et managériale, la distribution géographique, le comportement émergent, et le développement évolutif [2].

1.3. Définition 3

Les systèmes de systèmes sont des systèmes concourants et distribués à grande échelle qui sont composés des systèmes complexes [3].

A partir de ces définitions et d'autre dans la littérature, nous choisissons la définition suivante: Un Système de Système (SoS) est une collection de systèmes dédiés qui mettent en commun leurs ressources et leurs capacités afin de créer un nouveau système, plus complexe qui offre plus de fonctionnalités et de performances que la somme des systèmes constitutifs [4].

2. Les caractéristiques de SoS

Un grand nombre de chercheurs ont tenté de documenter les caractéristiques de SoS comme [5] et [6].

Ces caractéristiques permettent de faire la distinction entre les systèmes de systèmes et les systèmes monolithiques complexes. Les caractéristiques sont les suivantes [7] :

- **L'indépendance opérationnelle des systèmes constitutifs**

Les systèmes constitutifs d'un SoS doivent être capables de fonctionner indépendamment. Cela signifie que chacun de ces systèmes est indépendant et dispose de son propre usage (Exemple de la force navale militaire et ses composants, tels que des véhicules ou des avions).

- **L'indépendance managériale des systèmes constitutifs**

Cela signifie que les systèmes constituants sont acquis séparément, puis intégrés dans un système plus large. Ils ont une existence et une organisation séparée dans leur développement, ainsi que dans leur maintenance.

- **Développement évolutif**

Puisque la configuration du système est évolutive, cela signifie que le système global n'est pas complètement formé. Son développement est évolutif avec des fonctions et des objectifs ajoutés, supprimés et modifiés.

- **Comportement Emergent**

L'émergence des comportements signifie que le système global propose des propriétés et des fonctionnalités non présentes dans l'un de ses systèmes constitutifs, puisque ces comportements émergents ne peuvent pas être attribués à l'un de ces systèmes constitutifs. De plus, ces comportements ne peuvent pas être toujours prévus, ce qui conduit à des difficultés pour la validation du système global. Un exemple de service émergent est le routage IP (protocole

Internet), qui est à l'origine de l'Internet. Aucun routeur IP connaît la topologie complète des interconnexions de l'Internet, ou encore la configuration des interconnexions locales dans son propre voisinage. Et pourtant, le routage IP est un processus efficace qu'on peut faire confiance, et qui transfère de manière prévisible les messages sources à la destination prévue. Chaque routeur dans le chemin du message décide quel est le routeur voisin qui constituera le point suivant, même s'il n'a pas connaissance des routeurs ou des chemins potentiels accessibles dans son voisinage immédiat. Le routage IP est donc un service émergent, qui travaille également avec des informations incomplètes, imprécises et obsolètes, tout en offrant une fonctionnalité efficace et prévisible.

- **Distribution géographique**

La répartition géographique de ces systèmes constitutifs signifie qu'ils sont situés dans des endroits différents; cette extension géographique est relative et dépend largement des technologies disponibles et des moyens de communication. Les systèmes peuvent échanger des informations, mais ne peuvent pas échanger des quantités importantes d'énergie ou de matière.

3. Les types de systèmes de systèmes

Dahmann et Baldwin et Maier ont identifiés quatre types de SoS (System of System) selon la manière dont ils sont gérés:

- **Virtuel :** Ce type de SoS manque d'une autorité centrale de gestion et d'un objectif clair de SoS. Il est souvent ad hoc[1] et les systèmes constitutifs ne sont pas nécessairement connus [8].
- **Collaboratif :** Dans le cadre d'un SoS collaboratif, les équipes d'ingénierie des systèmes constituants travaillent ensemble plus ou moins volontairement pour atteindre des objectifs communs convenus. Dans ce type de SoS, il n'y a pas d'équipe d'ingénierie SoS pour guider ou gérer les activités liées aux SoS des systèmes constituants [8].
- **Reconnu** : Les SoS (System of System) ont des objectifs reconnus, un gestionnaire désigné et des ressources pour le SoS. Toutefois, les systèmes constitutifs conservent leur propriété indépendante, leurs objectifs, leur financement et leurs approches de développement et de maintien en puissance. Un exemple de ce type de SoS pourrait être un SoS de contrôle et de

[1] **Ad hoc** est une locution latine qui signifie « pour cela ». Elle s'emploie de nos jours pour « qui a été institué spécialement pour répondre à un besoin ».

commande militaire qui a fait la transition d'un SoS collaboratif à un SoS reconnu en raison de l'importance des missions soutenues par le SoS ou des complexités des capacités transversales de SoS [9].

- **dirigé** : Un SoS dirigé est géré de façon centralisée par une équipe gouvernementale, d'entreprise ou d'intégrateur de systèmes de plomb (LSI [Lead System Integrator]) et construit pour remplir des objectifs spécifiques. Les systèmes constitutifs maintiennent leurs capacités et continuent à fonctionner de façon indépendante, mais l'évolution est principalement contrôlée par l'organisation de gestion de SoS. Par exemple un réseau intégré de défense aérienne est généralement géré de manière centralisée pour défendre une région contre les systèmes ennemis bien que ses systèmes composants conservent la capacité de fonctionner indépendamment, et de faire le nécessaire lors d'un combat [8].

III. La Description des Missions

1. Définition d'une mission

Une mission est un ensemble d'activités permettant d'atteindre les objectifs du SoS. Une spécification de la mission d'un SoS doit comporter clairement les buts et les contraintes de la mission [10].

2. Langage de description de mission

Pour décrire la mission d'un SoS, nous avons besoin d'un langage permettant la description de tous les concepts qui caractérisent une mission. Nous avons remarqué qu'il existe vraiment peu de travaux qui abordent la spécification de la mission d'un SoS. Cependant, nous avons remarqué que le concept de mission apparaît dans les domaines de l'ingénierie logicielle et de l'ingénierie des systèmes sous différentes définitions: des buts à atteindre ou des d'activités ordonnées à exécuter dans un ordre prédéfini (processus). Nous avons analysé les travaux visant à décrire les exigences et les buts [11, 12,13], ainsi que les travaux qui décrivent les processus [14,15]. Dans ce qui suit, nous présentons les quelques approches pouvant être utilisées pour décrire une mission.

2.1. Approches d'ingénierie des exigences

Comme la mission est en quelque sorte similaire à la notion d'objectif, nous avons analysé les langages les plus connues liées à la description du but :

- **Kaos**

KAOS (Keep All Objectives Satisfied) est une méthode pour la spécifiction des exigences.Cette méthode permet aux analystes de construire des modèles des exigences en s'appuyant sur un raisonnement orienté par les buts. Le modèle des exigences KAOS se compose de cinq sous-modèles fortement liés par des règles de cohérence (le modèle de buts, le modèle objets, le modèle des responsabilités, le modèle des opérations et le modèle des comportements). Le principe de la méthode KAOS est de décomposer le but le plus abstrait en des sous buts plus concrets jusqu'au point ou on pourrait attribuer aux buts des acteurs qui peuvent les satisfaire [13].

- **Tropos**

Tropos[2] est une méthodologie, pour la construction des systèmes orientés agent. Elle repose sur deux idées clés. Tout d'abord, la notion d'agent et toutes les notions de mentalité connexes (par exemple les objectifs et les plans) sont utilisées dans toutes les phases du développement de logiciels, de l'analyse précoce jusqu'à la mise en œuvre effective. Deuxièmement, Tropos couvre également les premières phases de l'analyse des besoins, permettant ainsi une compréhension plus approfondie de l'environnement où le logiciel doit fonctionner et du type d'interactions qui devraient se produire entre les logiciels et les agents humains. Tropos adopte une approche transformationnelle, dans le sens où, à chaque étape, les modèles vont être raffinés de manière itérative par ajout ou suppression d'éléments ou relations dans les modèles [16].

- **mKaos**

MKAOS est un langage de description de mission dans un SoS (System of System). Le langage comprend plusieurs concepts spécifiques au domaine tel que les comportements émergents et les systèmes constitutifs identifiés dans un travail antérieur comme essentiels pour la description de

[2] [2]Tropos est déviré du terme grec *tropé* signifiant "facilement modifiable ou adaptable".

la mission.

Le langage permet de décrire les Missions dans le contexte du SoS à partir d'une perspective de haut niveau, représentant des missions globales et individuelles ; à des concepts de bas niveau tels que les capacités et les entités. Dans MKAOS, on ne peut pas exprimer cependant les contraintes [17].

Les concepts dérivés des approches d'ingénierie des exigences sont ceux qui permettent de décomposer la mission globale en sous-missions.

2.2 Langage de Définition de Processus

La mission peut également être considérée comme un processus, nous choisissons d'analyser les langages largement utilisés pour décrire les processus :

- **BPEL**

Le Business Process Execution Language (BPEL) est un langage utilisé pour définir les processus métier d'entreprise dans les services Web. BPEL étend le modèle d'interaction des services Web et lui permet de prendre en charge les transactions commerciales. Dans BPEL, chaque processus métier impliqué est supposé être mis en œuvre en tant que service Web.

Les processus écrits dans BPEL peuvent organiser des interactions entre les services Web en utilisant des documents XML de manière standardisée. Ces processus peuvent être exécutés sur n'importe quelle plate-forme ou produit conforme à la spécification BPEL [15].

- **BPMN**

Business Process Modeling Notation (BPMN) est une notation permettant de dresser un de diagramme de flux qui modélise les étapes d'un processus commercial planifié de bout en bout. Le principal avantage de BPMN est qu'il représente visuellement une séquence détaillée des activités commerciales et des flux d'information nécessaires pour compléter un processus [14].

3. Comparaison

Nous avons trouvé le concept Mission sous forme d'exigence dans la méthode MKAOS et sous forme des buts dans les méthodes KAOS et TROPOS ainsi que sous forme des processus dans les processus BPMN et BPEL.

Le concept soumission est présenté dans la méthode MKAOS et KAOS et existé sous forme de but dans la méthode TROPOS et il est présenté également sous forme de processus dans les processus BPMN et BPEL.

Le concept Dépendance temporelle est présenté sous forme de processus dans les processus BPMN et BPEL et il n'existe pas dans les méthodes MKAOS, KAOS et TROPOS.

Le concept Dépendance structurelle est présenté dans les méthodes MKAOS, KAOS, TROPOS et absent dans les processus BPMN et BPEL.

Le concept Conflit de la mission est présenté dans les méthodes MKAOS, TROPOS et existé sous forme de conflit dans la méthode KAOS, et absent dans les processus BPMN et BPEL.

Le tableau I.1 résume les concepts retenus dans notre étude sur les langages / approches existantes. Pour chaque langage / approche, nous indiquons la présence (V dans le tableau), ou l'absence (- dans le tableau) à l'égard de chaque concept

Concepts	MKAOS	Kaos	Tropos	BPMN	BPEL
Mission	V (exigence)	V (but)	V (but)	processus	processus
Soumission	V	V	V (but)	processus	processus
Dépendance temporelle	-	-	-	V	V
Dépendance structurelle	V	V	V	-	-
Conflit de la mission	V	V (conflit)	V	-	-

Tableau I.1 : les Concepts dans les Langages Existants

Voici la définition que nous avons retenue pour chacun de ces concepts:

- Mission et soumission: Fonction temporaire et déterminée qu'un système de système doit accomplir. Une mission peut être considérée comme un objectif à atteindre, ou comme une activité composée d'un ensemble d'actions ordonnées. Nous avons trouvé ce concept sous forme d'exigence dans la méthode MKAOS et sous forme des buts dans les méthodes KAOS et TROPOS ainsi que des processus dans les processus BPMN et BPEL.
- Dépendance temporelle: c'est un concept qui indique la relation temporelle entre les missions.

- Dépendance structurelle: concept indiquant qu'une mission peut être décomposée dans les soumissions.
- Conflit de la mission: situation définissant que certaines missions ne peuvent pas être totalement complétées.

IV. Conclusion

Dans ce chapitre nous avons parlé sur les systèmes de systèmes et leurs caractéristiques ainsi que leurs types et les missions. Nous avons présenté aussi quelques langages qui nous ont permis de tirer les concepts nécessaires de notre langage. Nous avons synthétisé ces langages par un tableau indiquant la présence des concepts qu'on a retenus dans notre langage.

Dans le chapitre suivant, nous allons présenter quelques langages qui permettent d'exprimer des contraintes.

Chapitre II

La Spécification des Contraintes

I. Introduction

Une contrainte est une relation sémantique liant un ou plusieurs éléments du modèle conceptuel et qui spécifie les conditions et les propositions qui doivent être respectées. Les contraintes peuvent être prédéfinies ou écrite en langages naturelles ou exprimée avec un langage formel comme OCL (Object Constraint Language).

Dans ce chapitre, nous allons présenter quelques langages de spécification de contraintes, nous nous sommes intéressés aux formalismes graphiques vu qu'ils sont plus faciles à utiliser. Puisque la majorité de ces langages s'appuient sur le langage OCL, nous allons tout d'abord présenter le langage OCL.

II. Classification des contraintes

Une contrainte est une relation sémantique liant un ou plusieurs éléments du modèle conceptuel (instances, entités,…) et qui spécifie les conditions et les propositions qui doivent être respectées. La contrainte peut être décrite en langage naturel pour être compréhensible par les différents acteurs ou exprimée avec un langage formel permettant une meilleure modélisation de l'information. Les systèmes de systèmes intègrent souvent plusieurs acteurs ou utilisateurs de domaine d'application qui interagissent et expriment des besoins et des exigences différentes. Il ressort de l'analyse de ces besoins que lors de la modélisation du problème, un ensemble de contraintes de différentes natures doit être pris en compte. Il existe trois types de contraintes [18] :

1. Contrainte utilisateur

La première classification vise le niveau d'expertise décideur et permet de distinguer les contraintes suivant la vision de l'utilisateur du système.

2. Contrainte métier

Cette grille vise le niveau d'expertise métier et permet d'établir un meilleur dialogue entre le concepteur et les décideurs.

3. Contrainte système

Cette expertise vise une décomposition informatique plus fine permettent d'analyser et de gérer la complexité des contraintes.

III. Langage de contraintes (OCL)

OCL (Object Constraint Language) est un langage déclaratif formel permettant d'exprimer des contraintes dans un modèle objet.
Le langage OCL a été utilisé dans le document de présentation de la sémantique du langage UML« UML semantics »[1] pour spécifier les règles de bonne formation du méta modèle UML. Chaque règle de bonne formation dans les sections concernant les modèles statiques des documents de spécification UML contient une expression OCL qui est un invariant de la classe concernée [19].

1. Pourquoi OCL ?

En modélisation orientée objet, un modèle graphique comme la représentation d'une classe n'est parfois pas suffisant à exprimer la réalité, dans le cadre d'une spécification précise et non ambiguë.

Il est souvent nécessaire de décrire des contraintes supplémentaires sur les objets du modèle. De telles contraintes sont souvent rédigées en langage naturel. La pratique a montré que cela aboutissait souvent à des ambiguïtés.
Afin d'écrire des contraintes sans ambiguïtés, des langages formels ont été développés. De tels langages présentent un inconvénient majeur. Ils ne peuvent être utilisés que par des personnes ayant une bonne pratique de la notation mathématique, ce qui constitue souvent un handicap dans les domaines de la gestion ou de la modélisation des systèmes [19].
OCL a été développé pour palier cette difficulté. C'est un langage formel qui reste facile à lire et à écrire. Il a été développé comme un langage de modélisation de gestion dans le département IBM et prend ses racines dans la méthode Syntropy[2]
OCL est un langage d'expressions. De plus, une expression OCL est garantie comme étant sans effet de bord. Elle ne peut pas provoquer de changements dans le modèle. Cela signifie que l'état du système ne sera jamais modifié du fait de l'évaluation d'une expression OCL.
A chaque fois qu'une expression OCL est évaluée, elle délivre simplement une valeur. OCL n'est pas un langage de programmation. Ainsi, il n'est pas possible d'écrire une logique de programme ou un contrôle de flux en OCL. On ne peut pas appeler des processus ou activer

1 [http://www-306.ibm.com/software/rational/uml/resources/documentation.html]
2 [http://www.syntropy.co.uk/syntropy/].

des opérations au sens algorithmique. Parce qu'OCL est avant tout un langage de modélisation, il ne contient pas d'instructions qui soient directement exécutables [19].
Par ailleurs, OCL est un langage typé. Chaque expression OCL possède un type. Par conséquent, dans une expression OCL correcte, les types utilisés doivent être conformes (on ne peut pas comparer un entier avec une chaîne de caractères) [19].

2. Utilisation d'OCL

OCL peut être utilisé avec des objectifs différents [19]:

· Pour spécifier des invariants sur des classes ou des types dans un modèle de classes.
· Pour spécifier un type invariant pour un stéréotype.
· Pour décrire des pré et post conditions sur des opérations et des méthodes.
· Pour décrire des gardes
· Comme langage de navigation dans un diagramme.
· Pour spécifier des contraintes sur des opérations.

3. Typologie des contraintes

3.1 Notion de contexte

Toute contrainte OCL est liée à un contexte spécifique, l'élément auquel la contrainte est attachée, ce contexte est une classe.

Syntaxe

context mon contexte
<Stéréotype> nom contrainte : expression de la contrainte
Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clef *self.* [20]

3.2 Les stéréotypes

Le stéréotype peut prendre les valeurs suivantes :
-**inv** invariant de classe
- **pre** pré condition
- **post** post condition
- **body** indique le résultat d'une opération query
- **init** indique la valeur initiale d'un attribut
- **derive** indique la valeur dérivée d'un attribut [20]

Exemple d'application :

Application bancaire :

-des comptes bancaires

-des personnes (clients)

-des banques

Spécification :

-un compte doit avoir un solde toujours positif

-un client peut posséder plusieurs comptes

-un client peut être client de plusieurs comptes

-un client d'une banque possède au moins un compte dans cette banque

-une banque gère plusieurs comptes

-une banque possède plusieurs clients

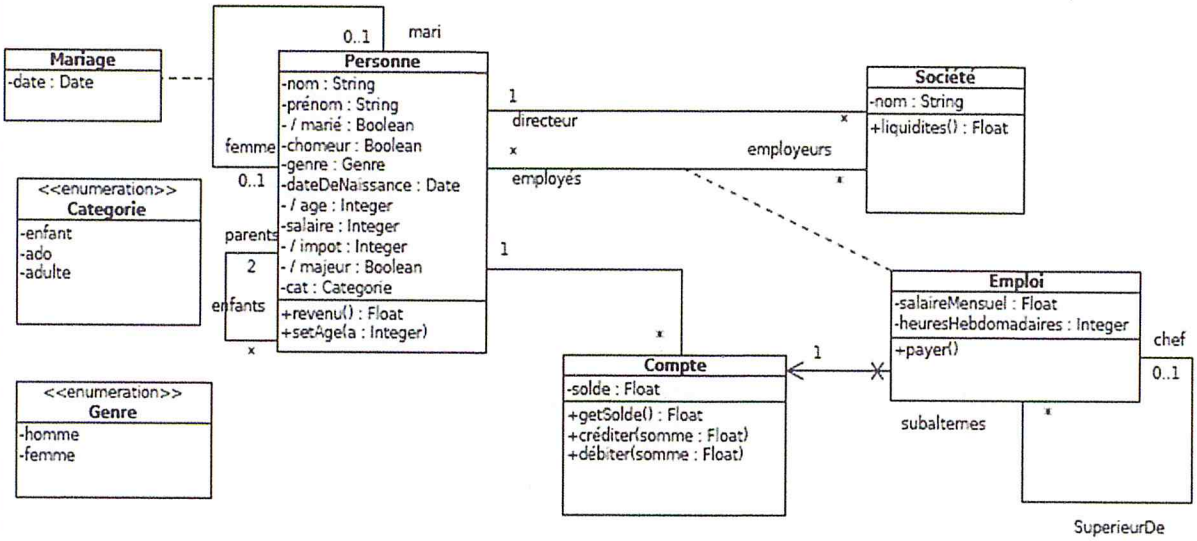


Fig II.1 : Exemple de diagramme de classe

*context*Compte : l'expression OCL s'applique à la classe compte c'est-à-dire à toutes les instances de cette classe.

*context*Compte :: getSolde() : Real : l'expression OCL s'applique à la méthode getSolde().

*context*Compte :: créditer(somme :Real) : l'expression OCL s'applique à la méthode créditer.

*context*Compte : solde : Real : l'expression OCL s'applique à l'attribut solde.

3.2.1 Invariants

Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence.

<StéréoType> : **inv**
context Compte
inv : solde > 0 -- solde d'un compte doit toujours être positif
On peut placer des commentaires dans les expressions OCL, ils débutent par deux tirets et se prolongent jusqu'à la fin de la ligne. [20]

3.2.2 Pré-conditions et Post-conditions

Spécifie une contrainte qui doit être vérifie avant/après l'appel d'une opération

Pré condition : état qui doit être respecté avant l'appel de l'opération
Post condition : état qui doit être respecté après l'appel de l'opération
Mots-clés : ***pre***et ***post*** [20]

3.2.3 Retour d'une opération (body)

Elle spécifie le corps d'une opération et définie directement le résultat de l'opération. [20]
Exemple:
context Compte:getSolde (): Real
Body : solde

3.2.4 Valeur initiale et dérivée d'un attribut (init et derive)

Spécifier la valeur initiale ou dérivée d'un attribut [20]
Exemple :
*context*Personne::age : entier
init : 0
context Personne ::age : entier
derive: Date::current ().getYear() - dateDeNaissance.getYear ()

3.3 Nommage de la contrainte

*context*Personne
*inv*ageBorné :(age <= 140) and (age>=0) -- l'âge est compris entre 0 et 140 [21]

3.4 Utilisation de mot-clef self

*context*Personne

inv : (***self***.age<=140) and (***self***.age>=0) -- l'âge est compris entre 0 et 140

. Permet d'accéder à une caractéristique d'un objet (méthode, attribut, terminaison d'association)

Selfobjet désigné par le contexte. [22]

3.5 Utilisation d'un nom d'instance formel

*context*p : Personne

*inv*ageBorné: (p.age<=140) and (p.age>=0) [21]

3.6 Syntaxe pour préciser la signature de l'opération

context ma_classe:mon_op (liste_param) : type_retour [22]

Exemple :

context Personne:setAge (a : entier)

Pre: (a <= 140) and (a >=0) and (a >= age)

Post : age = a -- on peut écrire également a=age

4. les opérateurs

4.1 Opérateurs spéciaux

Dans la post condition, deux éléments particuliers sont utilisables :

-Pseudo-attribut ***result***: référence la valeur retournée par l'opération.

-mon_attribut***@pre***: référence la valeur de *mon_attribut* avant l'appel de l'opération [21].

Exemple :

context Compte: débiter (somme : Real)

Pre: somme > 0

Post: solde = solde***@pre*** - somme

context Compte:getSolde () : Real

Post: ***result*** = solde

4.2 Opérateurs conditionnels

Certaines contraintes sont dépendantes d'autres contraintes

If *expr1* ***then*** *expr2* ***else*** *expr3* ***endif***

Si l'expression expr1 est vraie alors expr2 doit être vraie sinon expr3 doit être vraie, Il n'existe pas de if … then sans la branche else, il faut utiliser le implies pour cela [20].

Exemple :

context Personne

inv : ***if*** age >=18 ***then***

majeur=vrai

else majeur=faux ***endif***

expr1 ***implies*** *expr2*

Si l'expression expr1 est vraie, alors expr2 doit être vraie également.

Si expr1 est fausse, alors l'expression complète est vraie

Exemple :

context Personne

inv : marié ***implies*** majeur

On peut définir plusieurs contraintes pour un invariants, une pré ou post condition, il ya **and** : « et logique » : l'invariant, pré ou post condition est vrai si toutes les expressions reliées par le **and** sont vraies, il existe aussi en OCL les autres opérateurs logiques classiques que l'on combine comme on veut : **or**, **not**, **xor**. Possibilité d'utiliser les parenthèses pour changer les priorités ou éviter des ambiguïtés [20].

5. Définitions de variables et d'opérations

5.1. L'expression *let*

Définir des variables pour simplifier l'écriture de certaines expressions, l'expression ***let*** permet de définir un attribut ou une opération qui peut être utilisé dans une contrainte.

Syntaxe

let variable : type = expression1 ***in*** expression2 [19]

Exemple :

context Personne

inv: ***let*** montantImposable : Real = salaire*0.8 ***in***

if (montantImposable >= 100000)

then impot = montantImposable*45/100

else if (montantImposable >= 50000)

then impot = montantImposable*30/100

else impot = montantImposable*10/100

endif

endif

5.2. L'expression *def*

Si on veut définir une variable/opération utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction ***def***.

Syntaxe

def variable : type = expression1 [19]

Exemple :

context Personne

def : montantImposable : Real = salaire*0.8 --définition d'une variable

context Personne

def : ageCorrect(a :Real) :Boolean = (a>=0) and (a<=140) -- définition d'une operation

On peut réécrire l'invariant sur l'age :

*context*Personne

inv: ageCorrect (age) -- l'age ne peut dépasser 140 ans

6. Les types

6.1. Types d'Assertion

Il y a trois types qui permettent de faire la conception par contrat invariants, pré et post conditions ça se fait de la manière suivante, Si l'appelant respecte les contraintes de la pré condition alors l'appelé s'engage à respecter la post-condition, Si l'appelant ne respecte pas la pré condition, alors le résultat de l'appel est indéfini, Pour l'exemple précédent : Si l'appelant de l'opération débiter passe une somme positive en paramètre, alors le compte est bien débité de cette somme[20].

6.2. Types de Base et Opérations

Il y a 4 types de base en OCL integer,real,string,boolean[22].

6.2.1. Le Type Entier (integer)

Les valeurs s'écrivent de manière ordinaire : 1 ; -5 ; 2 ; 34 ; 26524 ; ...

Les opérateurs sont les suivants : =, *, +, -, /, abs(), div(), mod(),max(), min()

6.2.2. Le Type Réel (real)

Les valeurs s'écrivent de manière ordinaire : 1,5 ; 3,14 ; ...

Les opérateurs sont les suivants : =, *, +, -, /, abs (), floor (), round (), max (), min (), >, <, <=, >=,....

6.2.3. Le Type Chaine de Caractères (string)

Les chaines s'écrivent entre deux quotes : "Ceci est une chaine OCL ..."

Les opérateurs sont (notamment) les suivants : =, size(), concat(), substring(),toInteger(), toReal(), toUpper(), toLower()

6.2.4. Le Type Booléen (Boolean)

Exemple de valeurs : True, false

Les opérateurs sont les suivants : or, xor, and, not, implies, if-then-else-endif ; ... [22]

6.3. Types Enumérés

Il ya deux types d'utilisation d'une valeur d'une énumération

- NomEnum::valeur

Leurs valeurs apparaissent précédées de ::

Exemple :

context Personne

inv : *if* age <=12 *then* cat =Categorie::enfant

else if age <=18 *then* cat =Categorie::ado

else cat=Categorie::adulte

endif endif

- Ancienne notation : #valeur

Leurs valeurs apparaissent précédées de # [22] [21]

Exemple :

context Personne

inv: genre = #femme

7. Navigation et Accès

Dans une contrainte OCL associée à un objet, on peut :

-Accéder à l'état interne de cet objet (ses attributs) : self.attribut

-Accéder aux opérations de cet objet : self.operation

-Naviguer le long des liens : accéder de manière transitive à tous les objets ou groupe d'objets (et leur état) avec qui l'objet désigné par le contexte est en association

Syntaxe : nom de la classe associée avec minuscule (si pas d'ambiguïté) OU nom du rôle

Ambigüité : s'il s'agit de multiples associations entre objet désigné par le contexte et objet associé [21].

Il est possible de naviguer à travers les associations, en utilisant le rôle opposé [20].

Exemple

context Société

inv: self.personne.age >= 50 --ambigüe car il ya multiple associations

context Société

inv: self.directeur.age >= 50--utilisation de rôle opposé

IV. Les Formalismes Graphiques d'expression des contraintes

Il existe quelques formalismes d'expression de contraintes qui peuvent être utilisés aussi pour exprimer des contraintes. Nous en verrons ici trois formalismes différents qui utilisent une forme graphique d'expression de contraintes: les *constraint diagrams* [23], le *Visual First Order Logic* [24], les *visual OCL constraints* [25], [26], [27].

1. Constraint diagrams

Les diagrammes de contraintes [23] sont des *spider diagrams* augmentés par des flèches et des *universal spiders*. Les flèches représentent les relations binaires entre les ensembles et les *universal spiders* dénotent la quantification universelle, donc il est nécessaire d'expliquer très rapidement la théorie des *spider diagrams* avant d'aller à l'explication des *constraint diagrams. U*n spider ou araignée est un arbre avec des nœuds (appelés pieds) placés dans différentes zones ; Les bords de connexion (appelés jambes) sont des lignes droites. Une araignée touche une zone si l'un de ses pieds apparaît dans cette zone. Une araignée peut toucher une zone au plus une fois. On dit qu'une araignée Habite la région qui est l'union des zones qu'elle touche. Pour toute araignée *S*, l'habitat de *S*, noté η *(s)*, est la région habité par l'araignée . Il y a une légère différence sémantique entre les araignées avec des cercles comme des pieds et des araignées avec des carrés en pieds. Une araignée dont les pieds sont des cercles correspond à la quantification existentielle d'un élément dans l'ensemble désigné par l'habitat de l'araignée. Une araignée dont les pieds sont des carrés représente un élément donné. Une araignée Schrödinger désigne un ensemble dont la taille est zéro ou un [23].

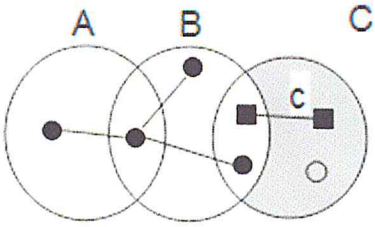


Fig II.2 : Exemple de Spider Diagram, tiré de [23]

Deux araignées distinctes désignent des éléments distincts, à moins qu'elles ne soient jointes par une *cravate* ou par un *brin* (fil). Une *cravate* est une double ligne droite (comme un signe égal) reliant deux pieds de différentes araignées, placé dans la même zone. Un *brin* (fil) est une ligne ondulée reliant deux pieds, de différentes araignées, placé dans la même zone. Le nid d'araignées s et t, écrit τ (s, t), est l'ensemble des zones z ayant la propriété que les pieds de s et t sont reliés par une *cravate* dans z .Deux araignées qui ont un *nid* non vide sont désignées comme des compagnons. Si les deux éléments indiqués par les araignées s et t sont dans l'ensemble désigné par la même zone dans le *nid* de s et t, alors s et t désignent le même élément. La *toile* d'araignées s et t, écrit ζ (s, t) représente l'union des zones z dont les araignées sont reliées soit par une *cravate* soit par un *brin* (fil). Deux araignées avec une toile non vide sont appelées des amis, deux araignées s et t peuvent (mais pas nécessairement doivent) désigner le même élément si cet élément est dans l'ensemble désigné par la *toile* de s et t. Chaque région est une union de zones. Une région est ombragée si chacune de ses zones composantes est ombragée, La sémantique d'une zone ombrée indique qu'il ne peut pas contenir d'éléments autres que ceux indiqués et ceci permet de limiter la cardinalité des éléments de la zone [23].

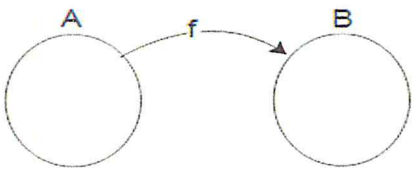


Fig II.3 : Flèche Simple [23]

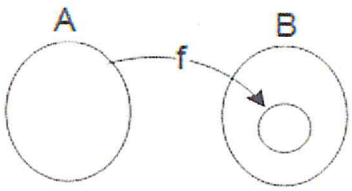


Fig II.4 : Contour Dérivé [23]

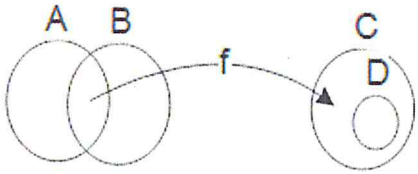


Fig II.5 : Flèche avec Zone comme Source et Cible [23]

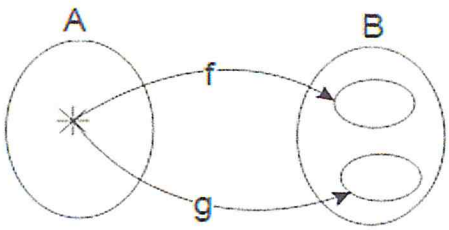


Fig II.6 : Flèche avec des Araignées comme Source et Cible [23]

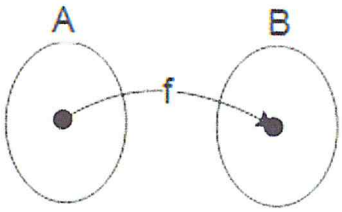


Fig II.7 : Existentiel et Araignées Dérivées [23]

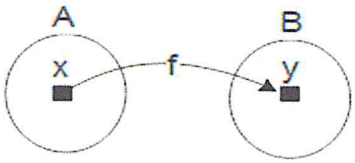


Fig II.8 : Flèches avec une Araignée Universelle comme Source [23]

Une flèche a une étiquette, une source et une cible. La source d'une flèche peut être un contour, une zone ou une araignée. La cible d'une flèche peut être un contour, une zone ou une araignée, il est interprété comme l'image relationnelle de l'ensemble représenté par la source sous la relation représentée par la flèche. Dans la Fig. 3 la source de la flèche f est le contour A et sa cible est le contour B. Son interprétation est A.f = B. Dans la Fig. 4, la cible de la flèche f est un contour dérivé. L'interprétation est A.f⊆B. Dans la Fig. 5, la source et la cible d'une flèche peut être une zone, l'interprétation est (A∩B) .f = (C-D), c'est-à-dire que l'intersection des ensembles Aet B donne avec f l'ensemble représenté par la différence des ensembles C et D. La source et la cible de la flèche de la Fig. 6 sont des araignées, l'interprétation est x.f = y, c'est-à-dire si on applique à x, la relation f, on obtient y. L'expression de navigation dans la Fig. 7 est quantifiée existentiellement, car sa source est une araignée existentielle. Son interprétation est ∃x∈A • x.f∈B c'est-à-dire il existe au moins un élément de A qui par f donnera un élément de B.Dans la Fig. 8, l'araignée universelle s'étend sur tous les éléments de l'ensemble A. L'interprétation de ce diagramme est que pour chaque x dans A, x.f et x.g sont disjoints, c'est-à-dire ∀x∈A • x.f∩x.g = □ [23].

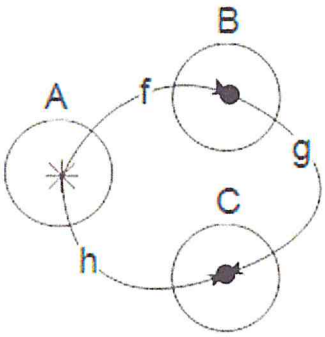


Fig II.9 : Navigation dans un Constraint Diagram, tiré de [23]

Il existe une expression de navigation à deux flèches ∀x∈A • x.f.g. L'expression x.f.g est interprétée comme (x.f) .g qui est égal à U y∈x.f y.g, Dans la Fig. 9, l'ensemble obtenu en naviguant à partir de l'araignée universelle en A par les flèches f et g est identique à celui obtenu en naviguant à partir de lui-même par h donc nous avons ∀x • x.f.g = x.h.

Lorsque les *constraint diagrams* sont utilisés dans la modélisation orientée objet, nous utilisons des rectangles pour les classes (Ou les types), on peut trouver aussi des ellipses à l'intérieur d'un rectangle qui représentent les différents états qu'elle peut prendre cette classe [23].

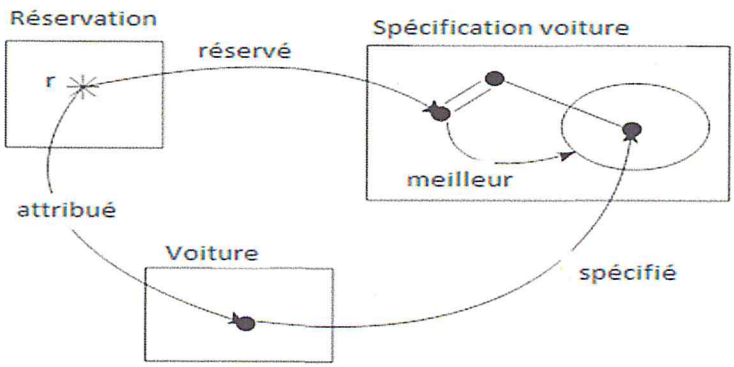


Fig II.10 : Exemple de Constraint Diagram de Location Voiture, inspiré de [23]

Le *constraint diagram* de la Fig. 10 exprime, entre autres contraintes, une invariante sur un modèle d'entreprise de location de voitures:
La spécification de la voiture attribuée à une réservation doit être égales ou supérieure à la spécification réservée. On note :

∀ r ∈ Réservation, r.attribué.spécifié = r.réservé ∨ r.attribué.spécifié ∈ r.réservé.meilleur

2. Visual First Order Logic

Nous présentons dans cette partie un langage schématique appelé Visual First Order Logic (VFOL) qui est née du travail sur les constraint diagrams.VFOL est susceptible d'être utile pour les spécifications logicielles dans le contexte d'UML, car il est similaire aux constraint diagrams. Un autre domaine d'application potentiel est pour l'enseignement de la logique des prédicats du premier ordre.VFOL conserve de nombreuses fonctionnalités des constraint diagrams qui sont utiles pour la modélisation des systèmeslogiciels [24].

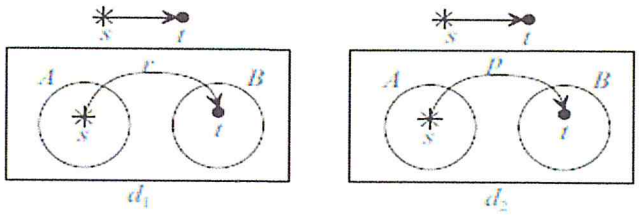


Fig II.11 : Exemple de deux Constraint Diagrams, tiré de [24]

Dans la figure 11, il existe deux constraint diagrams. Les astérisques étiquetés s, représentent la quantification universelle et les nœuds étiquetés t, ils représentent une quantification existentielle.

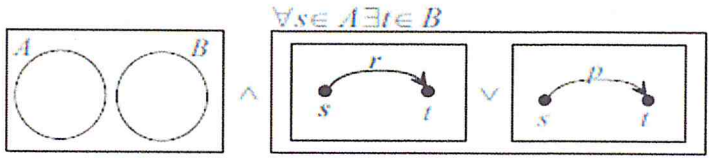


Fig II.12 : Exemple de Diagramme VFOL avec Disjonction et Conjonction, tiré de [24]

Un exemple de diagramme VFOL est illustré à la figure 12, qui est équivalent sémantiquement à l'instruction (1) ci-dessus

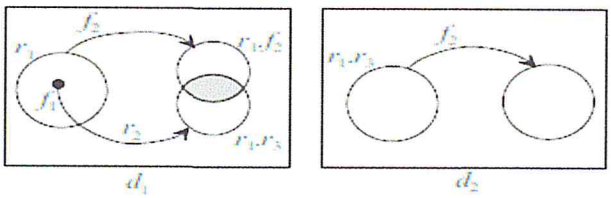


Fig II.13 : Deux Diagrammes VFOL, tiré de [24]

Dans cette section, nous présentons un alphabet de VFOL, tout d'abord, nous avons un ensemble infini de variables V = { x1, x2,….}, Nous définissons un ensemble de symboles

de fonction F = {f1,f2,...} et un ensemble de symboles de relation R = { r1,r2,....} les ensembles F et R peuvent être finis , une fonction α : F ∪ R ⟶ □ retourne l'arité de chaque symbole , nous présentons maintenant le syntaxe de diagramme VFOL qu'on peut le voir clairement dans l'exemple de la figure 13 , nous avons les symboles de relation avec l'arité égale 1 , R1 = { ri ∈ R : α (ri) = 1 } seront utilisés pour étiqueter les contours et on lui appelé des étiquettes de contour données , il y a aussi Les symboles de fonction avec l'arité égale 0 , F0 = {fi ∈ F : α(fi) = 0 } sont des constantes , les symboles de relation et de fonction restants seront utilisés pour étiqueter des flèches , un symbole spécial *U*, représente l'ensemble universel [24].

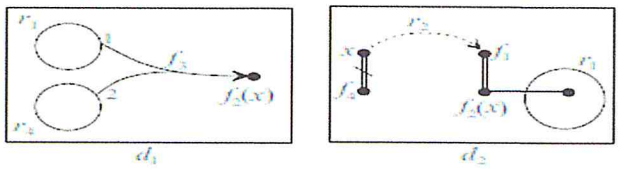


Fig II.14 : Flèches avec Multi-sources et Egalité, tiré de [24]

Le diagramme d1 de la figure 14 contient une étiquette de fonction f3 qui a une arité égale à 2, la flèche comporte deux sources et l'ordre dans lequel elles sont lues est indiqué par l'étiquetage de la flèche, Le diagramme exprime que r1 ∩ r2 = {} et (r1 x r2).f3 = f2(*x*), où *x* une variable libre dans d1 [24].

Le diagramme d2 exprime que f1 = f2(*x*), à l'aide d'une paire de segments de ligne droite parallèles comme un signe égal. Nous disons que f1 et f2 (x) sont identifiés de même, *x*≠ f4 donc on dit que *x* et f4 sont séparés, la flèche pointillée exprime que {*x*}.r2 comprend f1, en d'autre terme *x* est lié à au moins f1 sous r2 [24].

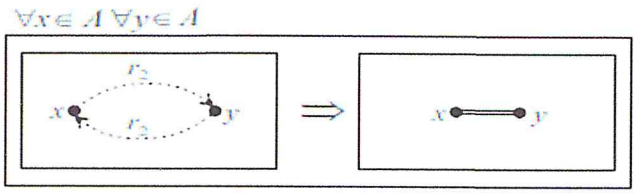


Fig II.15 : Un Diagramme Composé, tiré de [24]

La figure 15 présente la contrainte sur la relation r2 de la classe A, c'est-à-dire si *x* et *y* ont une relation qui s'appelle r2 et que cette relation est antisymétrique alors *x* égale *y* [24].

3. Visual OCL Constraints

Un autre moyen de représenter les contraintes, c'est Visual OCL Constraint, il suit la notation UML. VisualOCL est un langage graphique mais qui se base sur le langage OCL, donc typé et orienté objet, . Les nouveaux types de données et les opérations telles que les collections et les opérations comme *forall*, *select* et *union* sont représentés par des graphiques simples mais significatifs. Les expressions logiques sont désignées comme des graphiques Peircian à l'aide de boîtes imbriquées pour exprimer des disjonctions et des conjonctions. Une contrainte OCL est visualisée sous forme de rectangle arrondi avec deux sections ,la section de contexte contient le contexte du mot-clé suivi du nom de type de l'élément modèle (principalement une classe ou méthode) de la contrainte suivie du type de contrainte, par exemple Inv, pre, post ou def par exemple dans la figure 16 chaque personne a un prénom différent du nom de famille . Ainsi, le contexte est spécifié comme dans OCL. Dans la section du corps, le corps de la contrainte est visualisé [27].

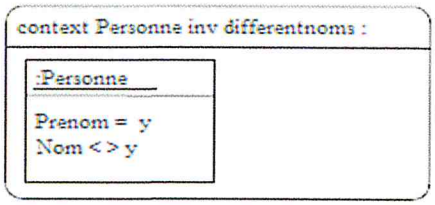


Fig II.16 : Exemple de Contrainte Visuelle OCL, tiré de [27]

Dans la figure 17 on a deux exemple de contrainte visuelle, sachant que la contrainte la plus à gauche vérifié qu'il existe au moins un employé de l'entreprise ayant plus de 18 ans, et la contrainte la plus à droite vérifié que pour l'état estMarie est vrai il faut que tous les employés sont mariés , dans un diagramme pareille lorsque on trouve une ligne pleine comme dans la contrainte la plus à gauche signifié qu'il existe au moins un chemin possible . Si une partie d'un chemin représenté pointillé comme dans la contrainte la plus à droite signifié que le chemin de doit pas exister[25].

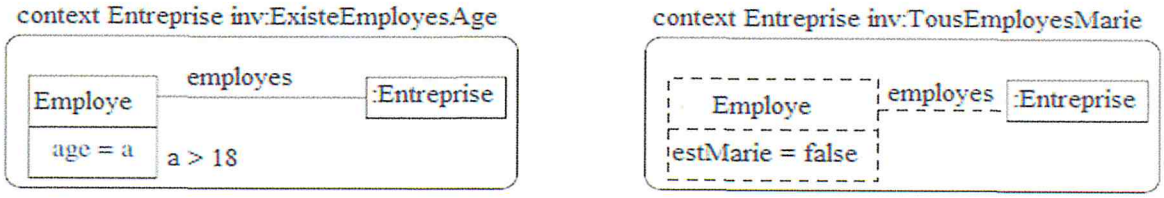


Fig II.17 : Exemples de deux Contraintes Visuelles, inspirées de [25]

Voici la traduction textuelle des contraintes présenté dans la figure 17 en OCL :

context Entreprise *inv* : ExisteEmployesAge

self.employes ->*select*(age>18) ->*notEmpty* -- contrainte plus à gauche

context Entreprise *inv* : TousEmployesMarie

self.employes ->*forall*(estMarie=true)

On peut exprimer des contraintes plus complexes avec les contraintes visuelles OCL en les séparant en plusieurs petites sous-contraintes, dans la figure 18 on va voir une contrainte plus complexe séparé en deux sous-contraintes,ces deux contraintes décrit que si une personne est marié alors elle a au moins 18 ans donc elle ne peut pas être marié à quelqu'un qui n'as pas 18 ans[25].

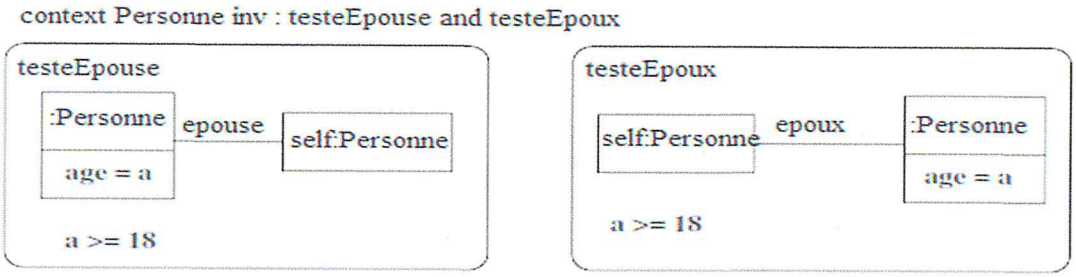


Fig II.18 : Exemple d'une Contrainte Complexe, inspirée de [25]

Voici la traduction textuelle des contraintes présenté dans en OCL :

context Personne *inv* :*self*.epouse ->*notEmptyimpliesself*.epouse.age>= 18 *and*

self.epoux ->*notEmpty impliesself*.epoux.age>= 18

Nous présentons maintenant un exemple tiré d'un projet «eGovernment» à Berlin l'objectif du projet est d'enregistré des résidents voici le diagramme de classe correspondant dans la figure 19 :

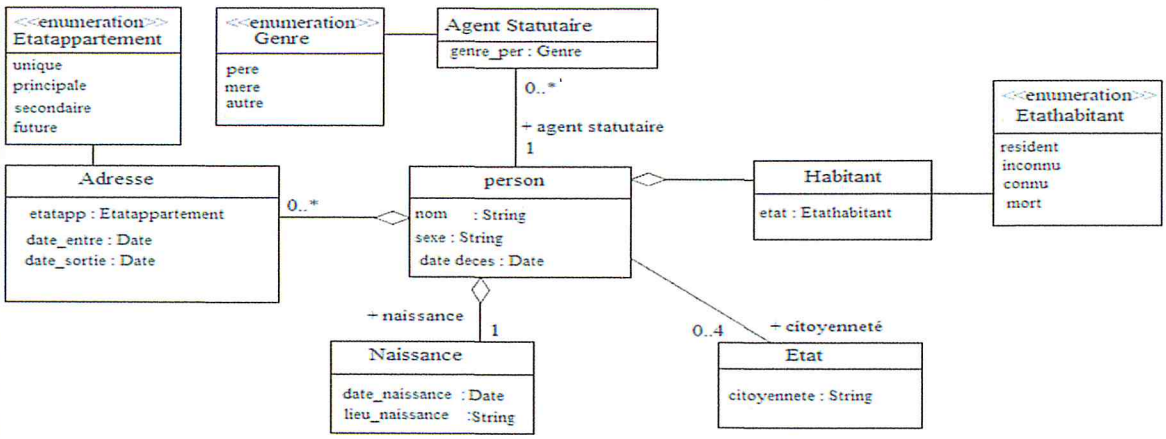


Fig II.19 : Exemple de Projet eGovernement, inspirée de [26]

A partir de diagramme on peut tirer la première contrainte :

- La date de naissance d'une personne vient avant la date d'entrée dans un appartement.

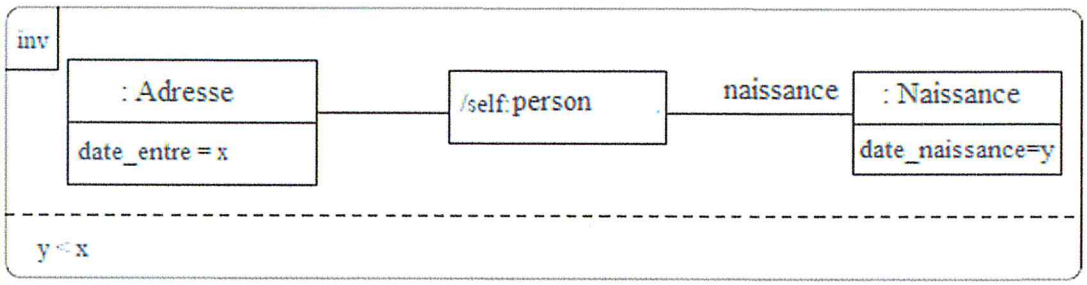


Fig II.20 : Exemple de Contrainte Visuelle, inspirée de [26]

La traduction en OCL :

Context Person inv :

Self.naissance.date_naissance < self.adresse.date_entre

Une autre contrainte :

- La date de naissance doit être inférieure à la date d'aujourd'hui.

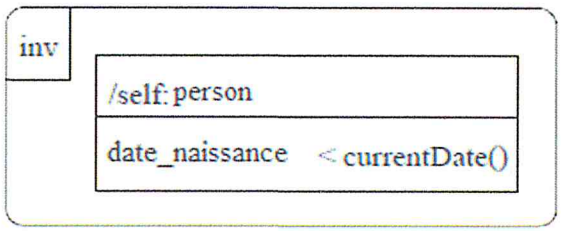


Fig II.21 : Exemple de Contrainte Visuelle sur un Attribut, inspirée de [26]

1. Navigation

Pour [26], la partie la plus intéressante de la visualisation de contraintes OCL est la navigation à travers une structure d'objet qui est beaucoup plus facile car le chemin est visualisé directement comme dans la figure 20, si la multiplicité de l'extrémité de l'association à un maximum de 1 alors la cible est un objet sinon si la multiplicité est supérieur à 1 donc ça sera une collection. Lorsqu'il y a plus d'une association possible entre 2 classes, le rôle est obligatoire, la navigation à partir des classes d'association est similaire à la navigation habituelle (ce qu'on a vu précédemment).

2. Comparaison Entre les trois Formalismes Graphiques

VisualOCL prend en charge l'OCL 2.0 version 1.5 complète, par contre les *constraints diagrams* prennent en charge seulement les contraintes statiques (invariants) et les contraintes dynamiques (pré et post condition),les *constraint diagrams* sont entièrement schématiques, tandis que le *VisualOCL* contient des contraintes textuels, les contraintes définies par *VisualOCL* sont faciles à lire et à apprendre en raison de sa nature liée à UML et OCL, tandis que les *constraint diagrams* sont plus difficiles à lire en raison de leur nature entièrement schématique. Le VFOL permet d'exprimer tout le langage des prédicats du premier ordre sous une forme relativement visuelle. En fait, il s'agit plutôt de *Constraint Diagrams* augmentés de prédicats du premier ordre ou combinés par des conjonctions, disjonctions ou implication, les contraintes définies par VFOL sont faciles a lire et apprendre par rapport au *constraint diagrams,* et moins faciles par rapport au *VisualOCL.* Pour exprimer les contraintes sur les missions, nous avons décidé de s'inspirer du formalisme *VisualOCL,* et d'utiliser OCL textuel quand la notation graphique est trop compliquée. Tous les détails de notre proposition vont être expliqués dans le chapitre 4.

V. Conclusion

Dans ce chapitre nous avons présenté les différents types de contraintes, le langage OCL qui va être utilisé pour exprimer des contraintes sur les missions ainsi que quelques formalismes graphiques utilisés pour exprimer les contraintes. Dans le chapitre qui suit, nous allons présenter les principes de la méta-modélisation.

Chapitre III

La méta-modélisation

I. Introduction

Puisque notre travail a pour but de proposer un méta-modèle pour la spécification d'une mission dans un système de système, nous allons aborder dans ce chapitre la méta-modélisation, ainsi que les principaux concepts de cette dernière.

II. Hiérarchie de Modélisation à 4 Niveaux

L'OMG[1] (Object Management Groupe) définit 4 niveaux de modélisation comme indiqué sur la figure ci dessous :

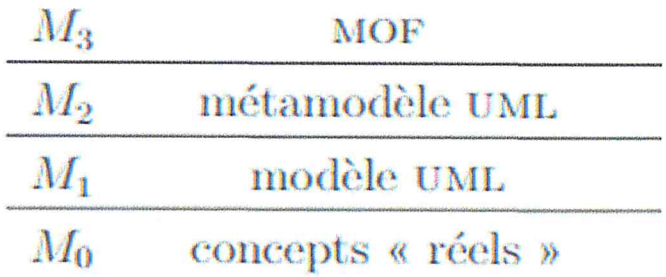


Fig III.1 : Architecture de Méta-Modélisation d'UML [28]

– M0 : système réel, système modélisé

– M1 : modèle du système réel défini dans un certain langage

– **M2: méta-modèle définissant ce langage**

– M3 : méta-méta-modèle définissant le méta-modèle

• Le niveau M3 est le MOF [2](Meta-Object Facility)

– Dernier niveau, il est méta-circulaire : il peut se définir lui-même

• Le MOF est – pour l'OMG – le méta-méta-modèle unique servant de base à la définition de tous les métas modèles [28].

La figure 2 illustre un exemple de présentations de modélisation de 4 niveaux :

[1] L'**Object Management Group (OMG)** est une association américaine à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

[2] Le **Meta-Object Facility** (MOF) est un standard de l'Object Management Group (OMG) s'intéressant à la représentation des métamodèles et leur manipulation.

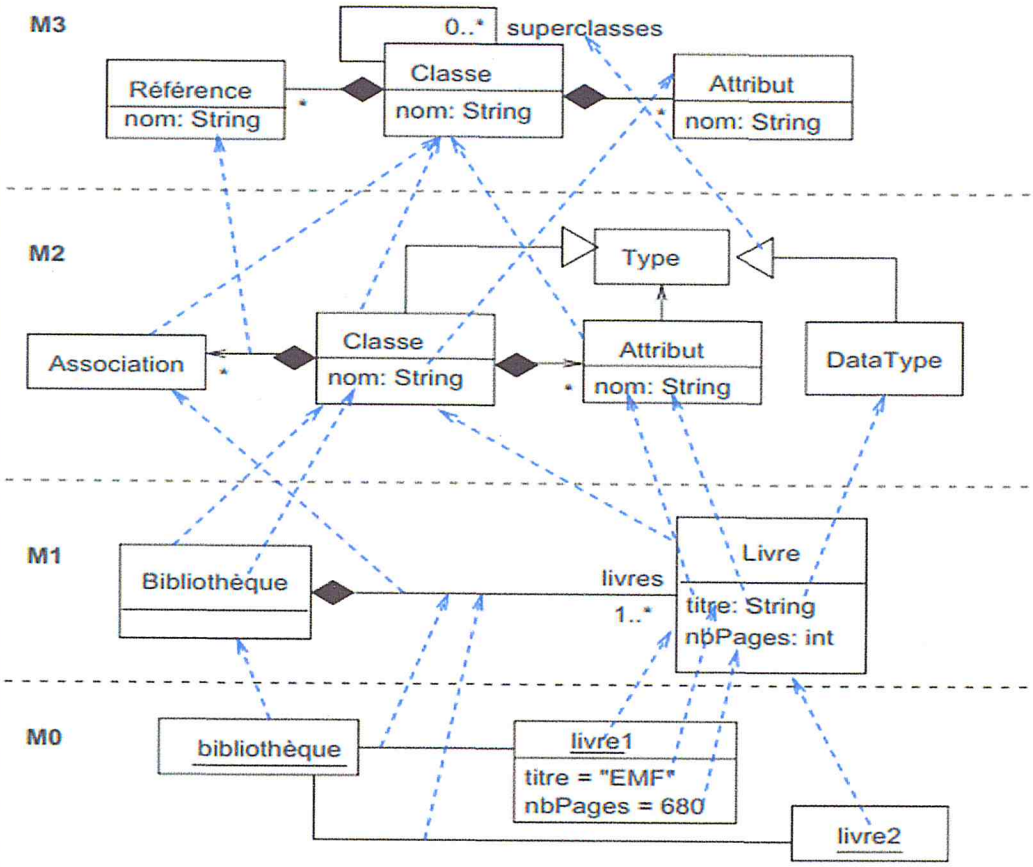


Fig III.2 : Exemple des 4 Niveaux de Modélisation.

III. Méta-Modèles

La sémantique d'un langage de programmation est définie sur le langage M2, pas sur le programme M1, Dans cette partie on va expliquer et détailler le niveau M2 de Meta-modèles.

1. C'est Quoi un Modèle ?

Un modèle est une vue simplifiée de la réalité pour une application particulière : il définit le sous-ensemble des données intéressantes et des comportements possibles, de la même manière qu'un programme définit un ensemble d'exécutions ou de traces possibles.
De manière simple un modèle est une simplification, une abstraction du système, On utilise des modèles pour mieux comprendre un système [29].

2. Définitions de Méta Modèles

2.1. Définition 1

La manière la plus courante de définir un méta-modèle est de le désigner comme un modèle du modèle [29].

2.2. Définition 2

Selon MOF, un méta modèle définit la structure que doit avoir tout modèle conforme à ce méta modèle. Autrement dit, tout modèle doit respecter la structure définie par son méta modèle. Par exemple, le méta modèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc.
La figure 3 illustre la relation entre un méta modèle et l'ensemble des modèles qu'il structure. Les métas modèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. MOF les représente sous forme de diagrammes de classes [30].

A partir de ces définitions et d'autre dans la littérature en déduit : Un méta-modèle est une définition formelle d'un modèle qui aide à le comprendre et qui facilite le raisonnement sur sa structure, sa sémantique et son usage. La méta-modélisation, qui est l'activité de construire des métas modèles, est très utilisée dans le domaine de l'ingénierie des systèmes d'information et particulièrement dans l'ingénierie des modèles et des méthodes. Dans l'ingénierie des méthodes, c'est un outil conceptuel indispensable pour définir et raisonner sur de nouveaux modèles. Dans l'ingénierie des outils, les méta-modèles sont une définition formelle nécessaire pour concevoir et construire les outils pour éditer, vérifier, transformer et éventuellement exécuter des instances de ces modèles [31].

Conformité : On dit qu'un modèle est conforme `à un méta modèle si :
chaque ´élément du modèle est instance d'un ´élément du méta modèle et, chaque contrainte exprimée sur méta modèle est respectée sur le modèle.

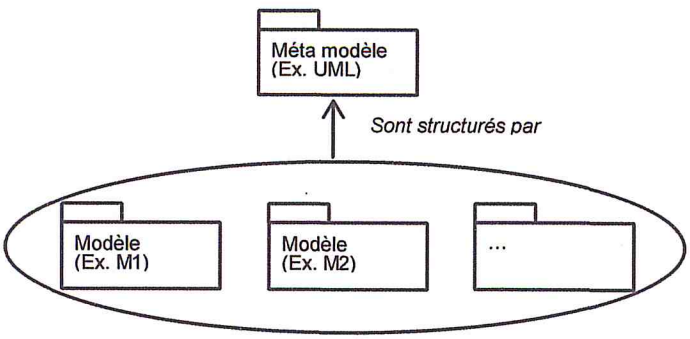


Fig III.3 : Relation entre Modèles et Méta-Modèles, tiré de [30]

Rappelons que les diagrammes de classes permettent de représenter les notions d'un domaine et leurs propriétés, que ces notions ou entités soient organisées ou non sous forme d'objets. Cette utilisation des diagrammes de classes a le double avantage de permettre de définir très précisément les métas modèles et de les rendre eux aussi pérennes et productifs.
Un méta modèle est donc une sorte de diagramme de classes qui définit la structure d'un ensemble de modèles [30].

3. Exemples de Méta-Modèles

Avant de définir précisément et théoriquement ce que sont les métas modèles, il nous paraît important d'en présenter un exemple de diagramme de classes. Nous montrerons ainsi à quoi servent les métas modèles et comment ils sont élaborés conceptuellement [30].

Nous allons donner un exemple de diagramme de classes simplifiés (package, classes, attributs).
L'information qui nous intéresse est la suivante :
« Un diagramme de classes contient des **packages.** Un package à un nom et contient des **classes.** Un package peut *importer* un autre package. Une classe a un nom et peut *contenir* des **attributs.** Une classe peut aussi *hériter* d'une autre classe. Un attribut a un nom et une visibilité qui peut être soit public soit private. Un attribut *a* un **type** qui peut être soit un **type de base (string, integer, boolean),** soit une classe du diagramme. »
La figure 4 illustre le méta modèle des diagrammes de classes. Il est composé de huit classes. La classe package contient un attribut nommé nom, dont le type est une chaîne de caractères. La classe class contient un attribut nommé nom, dont le type est une chaîne de caractères. La classe class hérite de la classe type. La classe attribute contient un attribut nommé nom, dont le type est une chaîne de caractères, et un attribut nommé visibilité, dont le type est une énumération (public ou private). Les classes string, integer et boolean héritent de la classe basicType, qui hérite de la classe type [30].
Il existe une association nommée import, qui a pour source et pour cible la classe package, ainsi qu'une association nommée super, qui a pour source et pour cible la classe class, une association nommée type, entre les classes attribute et type, et deux associations d'agrégation entre les classes package et class et entre les classes class et attribute [30].
Ce méta modèle définit que les modèles conformes ne peuvent contenir que des packages qui contiennent des classes contenant des attributs. Packages, classes et attributs ont des noms. Les packages peuvent importer d'autres packages et les classes hériter d'autres

classes, tandis que le type d'un attribut peut être soit un type de base, soit une classe.
Ce méta modèle définit donc bien la structure des diagrammes de classes telle que nous l'avons énoncée en langage naturel précédemment [30].
Cet exemple nous a permis de démystifier le méta modèle. Nous avons vu qu'un méta modèle était une sorte de diagramme de classes permettant de représenter les concepts fondamentaux d'un langage de modélisation sous forme de classes et de représenter les relations existantes entre ces concepts sous forme d'associations.

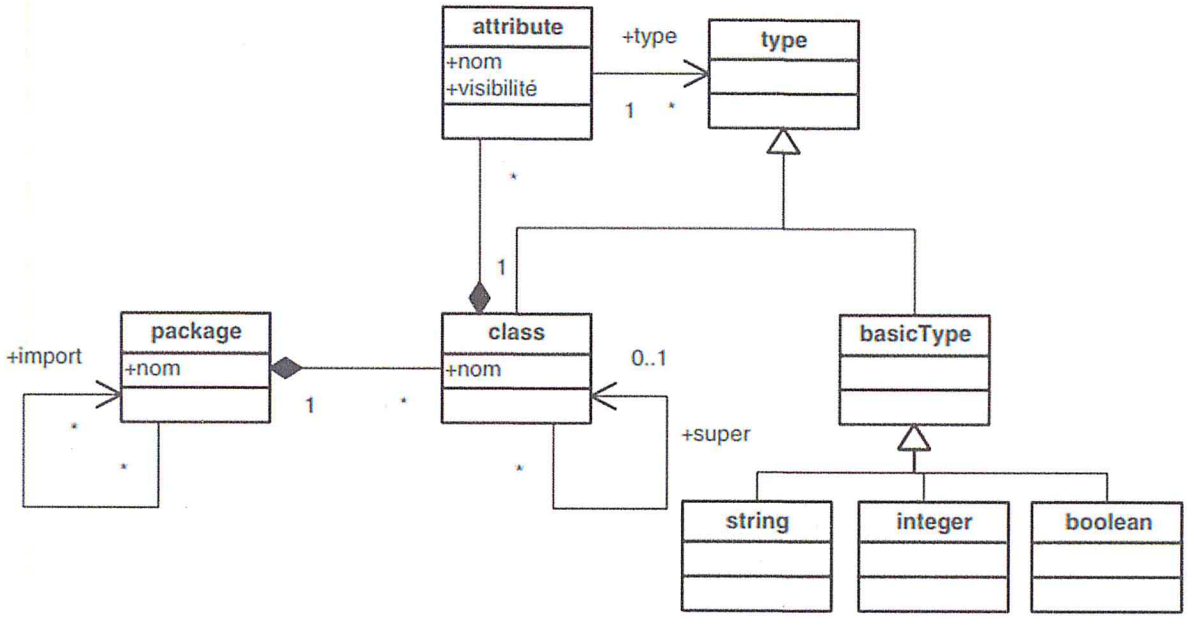


Fig III.4 : Méta-Modèle des Diagrammes de Classes, tiré de [30]

4. Présentation de Méta-Modèle MOF1.4

Après avoir vu qu'un méta modèle était une sorte de diagramme de classes et en avoir présenté deux exemples, il est temps d'expliquer précisément ce qu'est un méta modèle. Étant donné qu'il existe plusieurs façons de faire des métas modèles (MOF1.3, MOF1.4, MOF2.0, EMF, etc.) et que nous ne pouvons toutes les présenter, nous avons retenu la façon MOF version 1.4. Tous les métas modèles publics de l'OMG sont réalisés avec cette version, assez simple d'utilisation, contrairement à la 1.3, qui nécessite une connaissance de CORBA, ou à la 2.0, très complexe. Nous présentons la version 2.0 du MOF en fin de chapitre car c'est elle qui permettra d'élaborer les futurs métas modèles [30].
Pour MOF1.4, un méta modèle définit la structure d'un ensemble de modèles. Cette structuration est semblable à la structuration orientée objet. Les métas modèles MOF1.4

sont donc définis sous forme de classes. Les modèles conformes aux métas modèles sont considérés comme des instances de ces classes [30].

Afin de discerner les classes constituantes d'un méta modèle des autres classes, telles que les classes Java, MOF1.4 propose d'utiliser le terme *méta classe.* Un méta modèle est ainsi constitué d'un ensemble de méta classes. De même, afin de discerner les objets instances des métas classes des autres objets, MOF1.4 propose d'utiliser le terme *méta-objet.* Ainsi, un modèle est constitué d'un ensemble de méta-objets instances de méta classes [30].

Un méta classe a un nom et contient des attributs et des opérations, aussi appelés *méta attributs* et *méta-opérations.* Un méta-attribut représente une propriété d'un élément du modèle. Une méta-opération représente un traitement applicable à un élément du modèle.

Pour typer les attributs et les paramètres des opérations, MOF1.4 propose le concept de *type de donnée* (data Type). Un type de donnée permet de spécifier un type qui n'est pas un type d'objet. Les types tels que les booléens, les chaînes de caractères, les tableaux et les structures sont des types de données [30].

Pour ce qui concerne l'expression des relations entre méta classes, MOF1.4 propose le concept de *méta-association.* Une méta-association est une association binaire entre deux métas classes. Une méta-association a un nom, et chacune de ses extrémités peut porter un nom de rôle et une multiplicité [30].

Pour grouper entre eux les différents éléments d'un méta modèle, MOF1.4 propose le concept de *package.* Un package, aussi appelé méta package, est un espace de nommage servant à identifier par des noms les différents éléments qui le constituent.

Pour résumer, on peut dire que les concepts de base de MOF1.4 sont les suivants (en appellation anglaise) [30] :

- **Class :** Un méta classe permet de définir la structure de méta-objets. Un ensemble de méta-objets constitue un modèle. Un méta classe contient des méta-attributs et des méta-opérations.
- **Data Type :** Un type de donnée permet de spécifier le type non-objet d'un méta-attribut ou d'un paramètre d'une méta-opération.
- **Association :** Une méta-association permet de spécifier une relation binaire entre deux métas classes.
- **Package :** Un méta package permet de regrouper sous un même espace de nommage différents éléments d'un méta modèle.

4.1. Méta Classe (class)

Comme expliqué précédemment, un méta classe sert à décrire la structure d'un ensemble de méta-objets.

Un méta classe a les propriétés suivantes :

• Possède un nom.

• Peut hériter d'une ou de plusieurs autres méta classes. L'héritage signifie que la sous-classe possède tous les méta-attributs et toutes les méta-opérations de la superclasse. Si un méta classe hérite de plusieurs méta classes, il est impératif que ces méta classes n'aient pas de méta-attributs ni de méta-opérations de mêmes noms.

• Peut être abstraite. Si tel est le cas, aucun méta-objet ne peut être une instance directe de ce méta classe.

• Peut être considérée comme feuille (leaf) ou racine (root) d'un arbre d'héritage. Si le méta classe est feuille, cela signifie qu'aucun autre méta classe ne peut en hériter. Si le méta classe est racine, elle ne peut hériter d'un autre méta classe.

- **Méta-attribut (attribute) :**

  Un méta classe peut posséder zéro ou plusieurs méta-attributs.

  Un méta-attribut a les propriétés suivantes :

  • Possède un nom.

  • Possède une portée (scope). Peut-être de niveau méta-objet ou méta classe. Si laportée est de niveau méta-objet (instance_level), cela signifie que le méta-objet porte la valeur du méta-attribut. Si la portée est de niveau méta classe(classifier_level), cela signifie que leméta classe porte la valeur du méta-attribut pour tous les méta-objets instances.

  • Possède un type. Peut-être unméta classe ou un type de donnée *(voir la section suivante sur les types de données).*

  • Peut être ou non modifiable (isChangeable). S'il est non modifiable, cela signifie en quelque sorte que le méta-attribut a une valeur constante.

  • Peut être considéré comme dérivé (isDerived). Si le méta-attribut est dérivé, cela signifie que sa valeur peut être calculée, par exemple, grâce aux valeurs d'autres méta-attributs du méta classe.

  • Possède une multiplicité (multiplicity). La multiplicité est spécifiée par trois informations. La première contient les bornes maximale (upper) et minimale (lower)de la multiplicité. Par exemple, un méta-attribut ayant 0 comme borne minimale et 1

comme borne maximale est un méta-attribut optionnel, tandis qu'un attribut ayant 1 comme borne minimale et l'infini (représenté par le caractère *) comme borne maximale est un méta-attribut obligatoire pouvant avoir une infinité de valeurs. Les deux autres informations ne concernent que les méta-attributs ayant une borne maximale supérieure à 1.

L'information « ordonné » (is_ordered) permet de spécifier que l'ensemble des valeurs du méta-attribut est ordonné et l'information « unique » (is_unique) que l'ensemble des valeurs du méta-attribut ne doit pas contenir de doublon.

- **Méta-opération (operation)**

Un méta classe peut contenir zéro ou plusieurs méta-opérations.

Une méta-opération a les propriétés suivantes :

• Possède un nom.

• Possède une portée. Peut-être de niveau méta-objet ou méta classe. Si la portée est de niveau méta-objet, il est possible de demander à un méta-objet de réaliser le méta opération. Si la portée est de niveau méta classe, il est possible de demander directement au méta classe de réaliser la méta-opération.

• Peut contenir zéro ou plusieurs méta paramètres d'entrée. Un méta paramètre, comme un méta-attribut, a un nom, un type et une multiplicité. De plus, il a une direction (direction). Cette direction est soit monodirectionnelle, soit bidirectionnelle. Si elle est monodirectionnelle, elle peut être soit de l'appelant vers l'appelé (in), l'appelant donnant la valeur au méta paramètre, soit de l'appelé vers l'appelant (out), l'appelé donnant la valeur au méta paramètre. Si elle est bidirectionnelle (in_out), c'est indifféremment l'appelant ou l'appelé qui donne la valeur au méta paramètre.

• Peut optionnellement définir un type de retour. Ce dernier peut être soit un méta classe, soit un type de donnée. Si une méta-opération n'a pas de type de retour, on considère qu'elle ne retourne rien et non pas qu'elle retourne un ensemble vide (void).

• Peut jeter une ou plusieurs exceptions. Une exception a un nom et peut contenir zéro ou plusieurs attributs permettant de la renseigner.

4.2. Type de Donnée (data Type)

Les types de données permettent de définir des types non-objet. MOF1.4 permet de définir deux sortes de types de données : les types primitifs et les types construits.

Concernant les types primitifs, MOF1.4 définit les types suivants :

• boolean : la valeur est soit vrai (true), soit faux (false).
• integer : la valeur est un entier compris-en entre – 231 et + 231–1.
• long : la valeur est un entier compris entre – 261 et + 261 – 1.
• float : la valeur est définie par la norme ANSI/IEEE 754-1985.
• double : la valeur est définie par la norme ANSI/IEEE 754-1985.
• string : la valeur est une chaîne de caractères encodée en 16 bits.

Concernant les types construits, MOF1.4 propose les énumérations, les alias et les structures. Lorsqu'on élabore un méta modèle, on peut soit utiliser les types primitifs existants, soit construire ses propres types grâce aux constructions proposées par MOF1.4.

4.3. Méta-Association (association)

Une méta-association permet de définir une relation entre deux métas classes. En fait, une méta-association permet de définir la structure des liens entre les méta-objets instances des métas classes reliées par la méta-association.
Une méta-association a les propriétés suivantes :
• Possède un nom.
• Contient obligatoirement deux extrémités (associationEnd). Ce sont les extrémités d'une méta-association qui sont reliées aux métas classes.

➢ **Extrémité d'association (associationEnd)**

Une extrémité de méta-association a les propriétés suivantes :
• Possède un type. Le type de l'extrémité identifie le méta classe relié par le méta association.
• Possède un nom. Correspond au nom du rôle que joue le méta classe relié.
• Possède une multiplicité (même multiplicité que les méta-attributs). Cette multiplicité permet de spécifier le nombre d'instances de le méta classe identifiée par le type de l'extrémité pouvant être liées à exactement une instance du méta classe de l'autre extrémité de l'association. Par exemple, dans le méta modèle des diagrammes de cas d'utilisation *(voir figure 2.2),* la méta-association cas relie les métas classes acteur et cas d'utilisation. Une extrémité de cette méta-association a pour type cas d'utilisation.
La multiplicité de cette extrémité est *. Cela signifie que plusieurs instances de cas d'utilisation peuvent être liées à une instance d'acteur (rappelons que * signifie zéro ou plusieurs). Pour savoir combien d'instances d'acteur peuvent être liées à une instance de cas d'utilisation, il faut regarder l'autre extrémité de l'association (de nouveau *).

Une extrémité de méta-association dispose d'une spécification d'agrégation. Celle-ci peut être soit composite (composite), soit non existante (non-aggregate). Une spécification d'agrégation composite entraîne que les méta-objets (instances du méta classe identifié par le type de l'extrémité) contiennent les méta-objets instances du méta classe identifié par le type de l'autre extrémité (extrémité opposée) Concrètement, cela signifie que si le méta-objet est détruit, tous les méta-objets qu'il contient sont aussi détruits. Par exemple, dans le méta modèle des diagrammes de cas d'utilisation *(voir figure 2.2),* l'extrémité de la méta-association qui relie le méta classe système au méta classe cas d'utilisation et qui a pour type le méta classe système à une spécification d'agrégation composite (illustrée par le losange noir). Dans cet exemple, cela signifie que les méta-objets instances du méta classe système contiennent les méta-objets du méta classe cas d'utilisation. Si un méta-objet instance du méta classe système est détruit, tous les méta-objets instances du méta classe cas d'utilisation qu'il contient doivent être détruits.

Les contraintes suivantes doivent être respectées lorsqu'il existe des extrémités de méta association ayant des spécifications d'agrégation composite :

• Il ne faut pas que les deux extrémités d'une méta-association aient une spécification d'agréation composite. Cela empêche que deux méta-objets soient composés l'un de l'autre.

• Il ne faut pas qu'un méta classe soit relié par deux méta-associations dont les extrémités opposées (celles dont les types sont les autres métas classes) aient des politiques d'agrégation composite. Cela empêche qu'un méta-objet soit inclus dans deux méta-objets instances de deux métas classes différentes.

• Il faut que la multiplicité de l'extrémité ayant une spécification d'agrégation composite ait 1 comme maximum. Cela empêche qu'un méta-objet soit inclus dans deux métas objets instances de la même méta-classe. Une extrémité de méta-association peut être ou non modifiable (isChangeable). Si l'extrémité est changeable, cela signifie qu'il est possible de modifier n'importe quand les liens des méta-objets correspondant aux extrémités des méta-associations.

Une extrémité de méta-association peut être ou non navigable (isNavigable). Si l'extrémité est navigable, cela signifie qu'il est possible d'atteindre les valeurs des méta-attributs du méta-objet lié et de lui appeler ses méta-opérations. Bien sûr, l'accès n'est possible que pour les méta-objets liés par les extrémités opposées de la méta-association. Dans l'exemple du méta modèle des diagrammes de cas d'utilisation, la méta-association cas

à son extrémité dont le type est cas d'utilisation, qui est navigable (symbolisé par une flèche). Cela signifie que les méta-objets instances du méta classe acteur peuvent atteindre les valeurs des méta-attributs et appeler les méta-opérations des méta-objets instances du méta classe cas d'utilisation avec lesquels ils sont liés.

- **Référence (reference)**

  Afin de faciliter les navigations entre méta-objets *via* les méta-associations dont les extrémités sont navigables, MOF1.4 propose le concept de *référence* (reference). Une référence est une sorte de méta-attribut permettant essentiellement de naviguer *via* les méta-associations.

  Comme un méta-attribut, une référence à un nom, un type, une multiplicité, etc. La différence avec les méta-attributs réside dans le fait que le type de la référence ne peut être qu'un méta classe relié par une méta-association avec le méta classe qui contient la référence. De plus, l'extrémité qui pointe vers l'autre méta classe (celle qui ne contient pas la référence) doit être navigable. La multiplicité de la référence doit être la même que a multiplicité de cette extrémité. La référence est en fait une sorte d'alias pour atteindre les méta-objets liés.

4.4 Package

MOF1.4 propose le concept de *package,* qui permet de grouper différents éléments d'un même méta modèle. L'objectif est, d'une part, de regrouper les éléments d'un méta modèle portant sur un même domaine, par exemple, les éléments relatifs aux diagrammes de classes, et, d'autre part, de gérer plus facilement les méta-objets instances d'un ensemble de méta classes. Il est ainsi possible de stocker dans un même espace mémoire des méta-objets dépendant les uns des autres.

Un package a les propriétés suivantes :

• Possède un nom.

• Contient zéro ou plusieurs méta classes, méta-associations reliant ces classes et types de données nécessaires.

• Peut contenir zéro ou plusieurs autres packages.

• Lorsqu'un élément est dans un package (méta classe, méta-association, type de donnée, package), il dispose d'un nom complet. Ce nom correspond au nom complet du package, suivi du caractère «. », suivi du nom de l'élément. Si, par exemple, le méta classe C est contenue dans le package B, qui est lui-même contenu dans le package A, le

nom complet du méta classe est A.B.C.

• Un package peut hériter d'un ou de plusieurs autres packages. Dans ce cas, le sous package acquiert tous les éléments du super package (méta classes, méta-associations, types de données, packages).

• Un package peut importer un autre package. Le package qui importe l'autre package (le package importé) peut utiliser tous les éléments du package importé.

IV. Conclusion

Dans ce chapitre nous avons présenté les concepts fondamentaux des méta-modèles qui vont nous servir à proposer notre méta modèle.

Dans le chapitre suivant nous allons présenter notre contribution à savoir un méta-modèle pour exprimer les contraintes d'une mission.

Partie II
Contribution

Chapitre IV

Méta-Modèle pour La Spécification des Contraintes et des Missions

I. Introduction

Une mission est une séquence d'activités qui se déroulent dans un certain ordre. La mission d'un SoS (System of Systems) est finalement sa raison d'être, et elle est réalisée sous certaines contraintes. Dans ce chapitre, nous allons présenter notre méta-modèle permettant la spécification d'associer à une spécification de mission, les contraintes associées.

II. Description de la démarche suivie

Nous élaborons notre modèle en suivant la démarche proposée par Moody [32] comprenant deux phases illustré dans la figure IV.1. La première phase consiste à faire une analyse ontologique qui nous permettra d'identifier les concepts principaux, afin d'obtenir les constructions sémantiques, et dans la deuxième phase nous allons introduire des constructions graphiques.

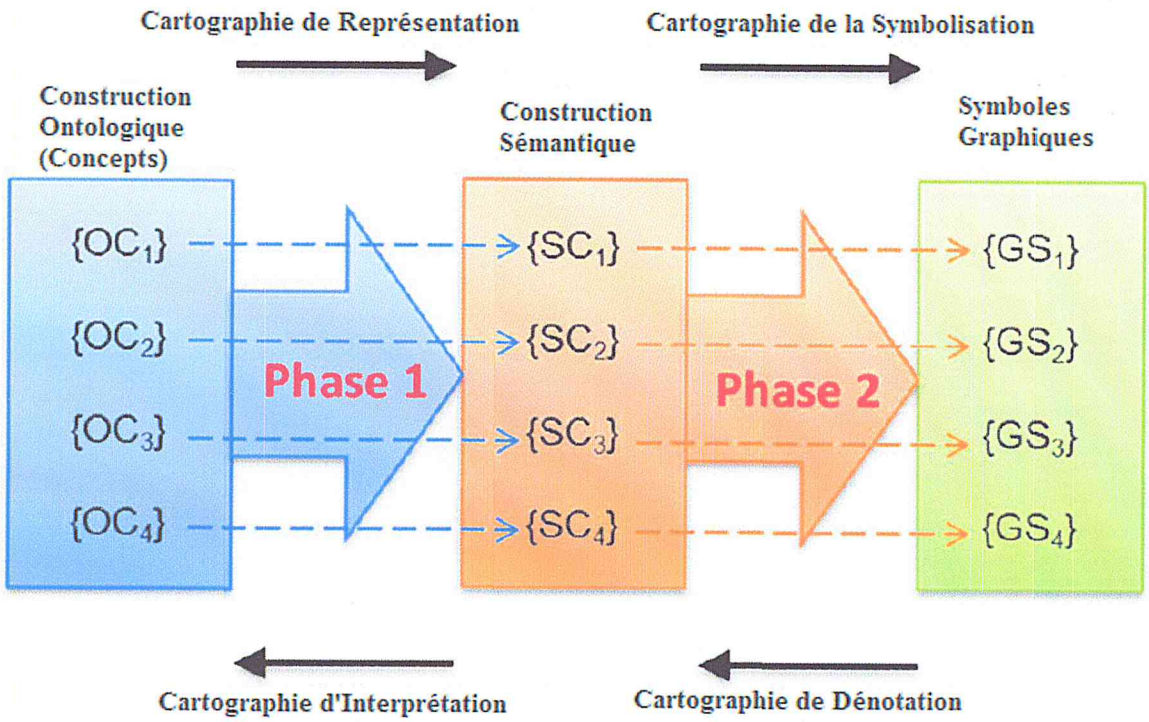


Fig IV.1 : Le Processus Global Requis pour Développer la Nouvelle Notation [33]

1. Phase I: Définir les constructions sémantiques nécessaires

Dans cette phase, nous devons faire une analyse ontologique du domaine des systèmes de systèmes, et ce pour identifier les constructions sémantiques nécessaires pour décrire les contraintes d'une mission. L'objectif est de réaliser un mappage un à un entre les constructions ontologique et sémantique comme il est illustré dans la figure IV.1. Pour se faire, nous devons éviter les quatre anomalies suivantes de se produire:

- La surcharge de construction : qui se produit lorsqu'un concept ontologique particulier n'a pas de construction correspondante dans la notation.
- La surcharge de construction : qui existe lorsque plusieurs concepts ontologiques peuvent être représentés par une seule construction de notation.
- La redondance de construction : qui existe lorsqu'un concept ontologique unique peut être représenté par de multiples constructions dans la notation.
- L'excès de construction : qui existe lorsqu'une construction dans la notation ne correspond à aucun concept ontologique.

Nous avons déduit nos concepts à partir des langages que nous avons présentés dans l'état de l'art, et à partir des caractéristiques des SoS (System of Systems).

- Mission et soumission: Fonction temporaire et déterminée qu'un système de système doit accomplir. Une mission est considérée dans notre travail comme un ensemble d'activités ordonnées. Une mission peut être atomique ou composée d'autres missions.
- Dépendance temporelle: c'est un concept qui indique la relation temporelle entre les missions (l'ordonnancement)
- Dépendance structurelle: concept indiquant qu'une mission peut être décomposée en s soumissions.
- Conflit entre missions: situation définissant que certaines missions ne peuvent pas être totalement complétées.
- Le déclencheur : c'est un évènement permettant de déclencher une mission particulière.
- La contrainte: c'est une condition ou une restriction sémantique exprimée sous forme d'instruction dans un langage. Le langage OCL a été utilisé dans notre cas.

- Le paramètre: un élément d'information qui est en rapport avec la mission et que nous devons considérer pour prendre une décision.

1.1. Description du Méta-Modèle Proposé

Le méta-modèle proposé, servant de base à l'expression des contraintes OCL pour la spécification d'une mission, combine les avantages des langages basés sur les objectifs et des langages basés sur les processus. Les concepts retenus par notre méta-modèle pour caractériser une mission constituent donc une synthèse de ceux trouvés dans les langages et les approches que nous avions vus dans le premier chapitre.

Nous présentons la syntaxe abstraite de notre méta-modèle afin de mettre en évidence les concepts impliqués.

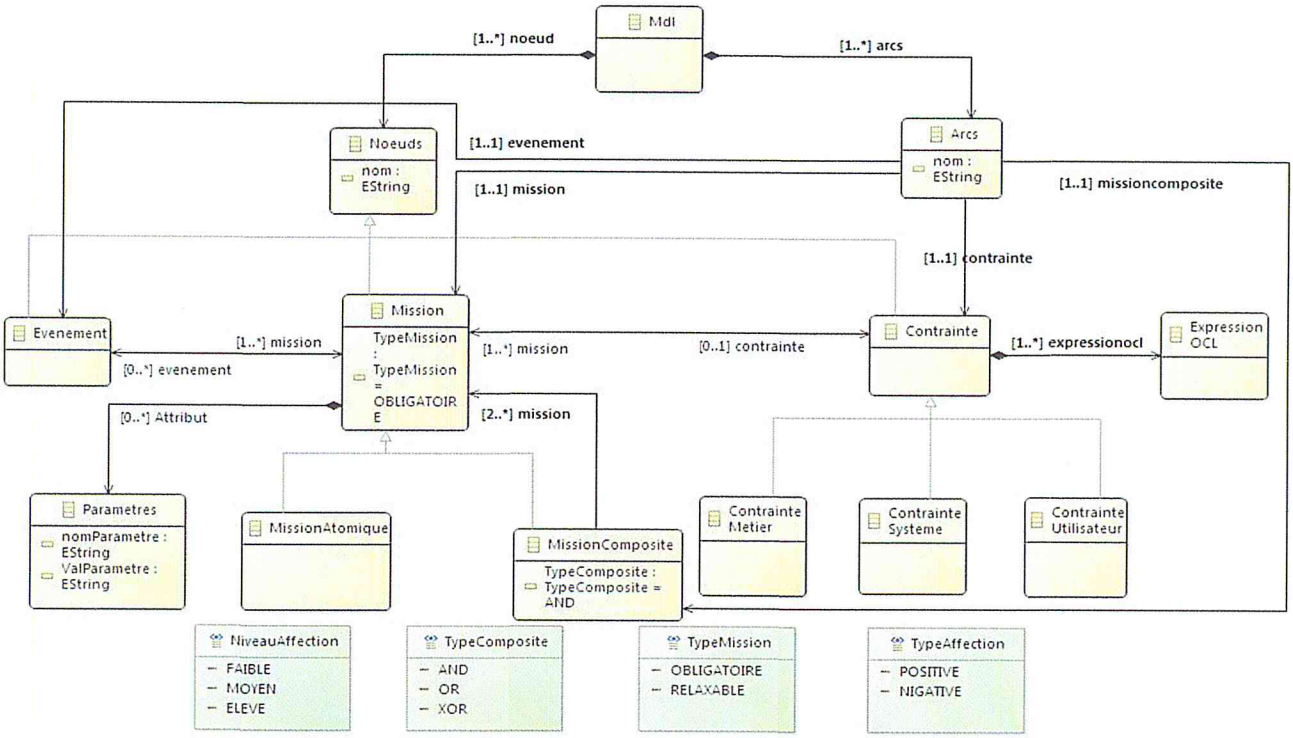


Fig IV.2 : Méta-Modèle Proposé

La figure IV.2 illustre notre méta-modèle où la méta-classe Mission est le concept principal. Nous définissons une mission comme un ensemble de fonctions temporaires et déterminées qu'un système doit accomplir, nous suggérons de décomposer cette méta-classe de haut niveau (généralement abstraite) en missions plus concrètes, et cela est exprimé par l'existence

de deux méta-classes (Mission Composite et Mission Atomique). La nature de la collaboration entre les missions composites est décrite à l'aide des opérateurs AND, OR et XOR, inspirés de [34], nous avons utilisé ces opérateurs à l'aide d'une énumération. Cependant, l'utilisation des énumérations nous permettra d'ajouter de nouveaux opérateurs si nécessaire. Une mission est composée de zéro ou plusieurs paramètres.

La racine du diagramme sera décrite à l'aide de la méta-classe MDL (pour représenter Mission Description Language), qui est composé de :

- Un ou plusieurs Arcs, ce dernier est un super méta-classe qui représente l'ensemble des arcs que nous pouvons utiliser dans les différentes relations entre les méta-classes.
- Un ou plusieurs Nœud, ce dernier représente l'ensemble des nœuds qui peut être soit :
  - ✓ Evénement : cette méta-classe représente l'ensemble des événements que nous pouvons avoir pour chaque mission.
  - ✓ Mission : cette méta-classe représente la mission principale qui peut être soit :
    - Mission Composite : cette méta-classe représente l'ensemble des missions composites.
    - Mission Atomique : cette méta-classe représente l'ensemble des missions atomiques.
  - ✓ Contrainte : cette super méta-classe représente la contrainte que nous pouvons associer à chaque mission, cette contrainte peut être soit :
    - Contrainte Utilisateur : cette méta-classe représente une contrainte utilisateur.
    - Contrainte Systeme : cette méta-classe représente une contrainte système.
    - Contrainte Metier : cette méta-classe représente une contrainte métier.

De plus, cette contrainte est composée d'un ou plusieurs expressions OCL, cette dernière représente les expressions OCL qui peuvent construire une contrainte.

2. Phase II: Développer les Constructions Graphiques Nécessaires

Dans cette phase, nous introduisons de nouvelles constructions graphiques (Symboles) qui correspondent aux constructions sémantiques développées au cours de la phase précédente, la

conception de constructions graphiques (Symboles) est apparemment une question de goût et d'esthétique. Le tableau 1 présente les nouveaux symboles graphiques proposés.

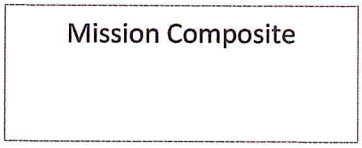
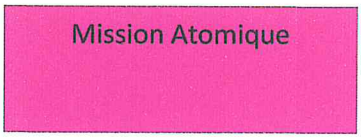
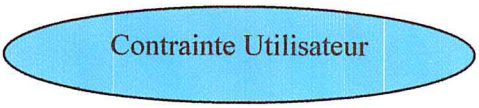
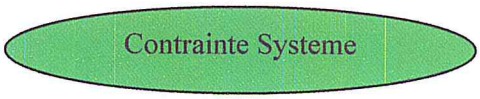
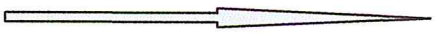
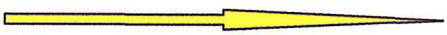
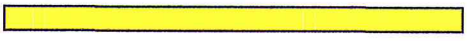
Constructions Sémantiques		Symboles Graphiques
Evénement		
Mission Composite		
Mission Atomique		
Contrainte	Contrainte Utilisateur	
	Contrainte Système	
	Contrainte Metier	
Arcs	Mission → Mission	
	Mission → Evénement	
	Mission → Contrainte	
Parametre		

Tableau IV.1 : Symboles Graphiques

Cette phase nécessite une analyse des relations entre les nouvelles constructions sémantiques, cela constitue la base de développement d'un langage visuel conforme aux principes présentés dans [39] illustré dans la figure IV.3

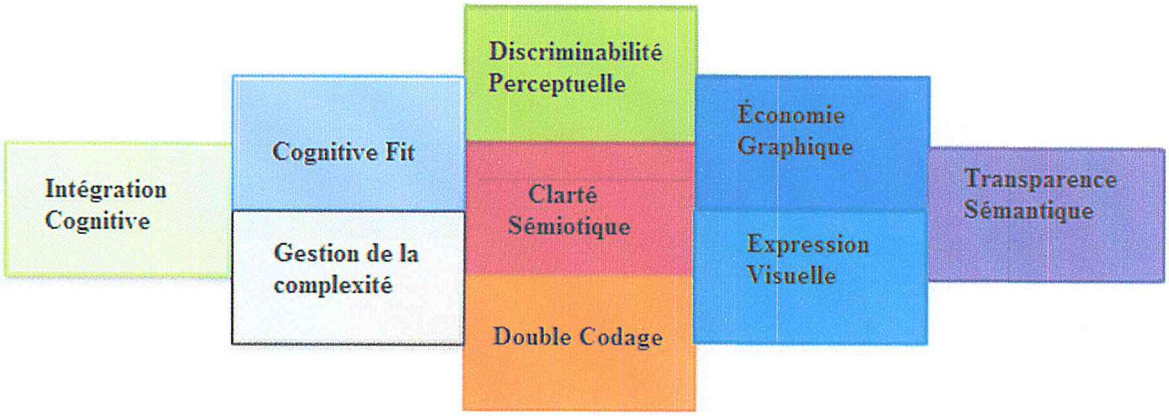


Fig IV.3 : Principes pour la Conception des Notations Visuelles, tiré de [33]

Nous allons présenter quelques principes que nous avons respectés pour réaliser cette phase :

2.1. Principe de La Clarté Sémiotique

Comme pour l'analyse ontologique, le principe de la clarté sémiotique indique qu'il devrait y avoir un mappage un à un entre les constructions sémantiques et les constructions graphiques. La clarté sémiotique est donc considérée comme le principe premier des neuf principes de Moody's [32]. Le non-respect de ce principe résulte en une des quatre anomalies suivantes [32]:

- La redondance des symboles: qui se produit lorsqu'une construction peut être représentée par plusieurs symboles graphiques.
- La surcharge de symboles: qui se produit lorsque deux constructions différentes peuvent être représentées par le même symbole graphique.
- L'excès de symbole: qui survient lorsqu'un symbole graphique ne correspond à aucune construction sémantique.
- Le déficit symbolique: qui se produit lorsqu'une construction sémantique n'a pas de symbole graphique correspondant.

2.2. Principe de Codage Double

Bien que l'utilisation du texte dans les diagrammes soit visiblement interdite, elle est réellement encouragée à utiliser le texte pour compléter, non pas pour remplacer les graphiques. Selon la double théorie du codage, l'utilisation d'une combinaison de graphiques et de textes est plus efficace que l'utilisation de l'un ou l'autre seul [35].

2.3. Principe de l'Economie Graphique

Le principe de l'économie graphique se réfère au nombre de différentes catégories de symboles utilisées dans une notation. Des études ont montré que la capacité humaine de discriminer entre les symboles perceptuellement distincts est d'environ six catégories [33] illustré dans la figure IV.4

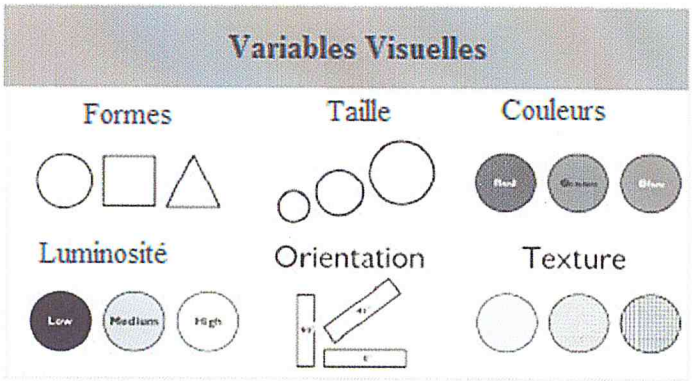


Fig IV.4 : Variables Visuelles, tiré de [32]

2.4. Principe de La Discrimination Perceptive

Le principe de la discrimination[1] perceptuelle concerne la facilité et la précision avec lesquelles différents symboles du même ensemble de notation peuvent être différenciés les uns des autres. Un diagramme ne peut être interprété avec précision que si ses symboles peuvent être discernés avec précision. Une conception de notation devrait viser à augmenter la distance visuelle entre ses symboles afin de maximiser la discrimination. La discrimination est mesurée par le nombre et la portée des variables visuelles utilisées. Plus les différences de nombres et de gammes de variables visuelles utilisées par différents symboles sont grandes,

[1] fait de distinguer entre des individus, des objets; ségrégation

plus la distance visuelle est grande. La figure IV.4 présente les six variables visuelles élémentaires qui peuvent être utilisées pour décoder l'information [33]. La variable de forme est la plus influente de toutes les variables visuelles. La forme est le facteur prédominant utilisé par les humains pour classer les objets dans le monde réel [32].

III. Cas d'Utilisation

Dans notre application, l'utilisateur final est l'expert de domaine d'application car personne ne peut mieux connaitre les missions dans les SoS (System of Systems) que lui, et pour décrire les différentes fonctionnalités de notre système, un diagramme de cas d'utilisation est utilisé et illustré dans la figure IV.5.

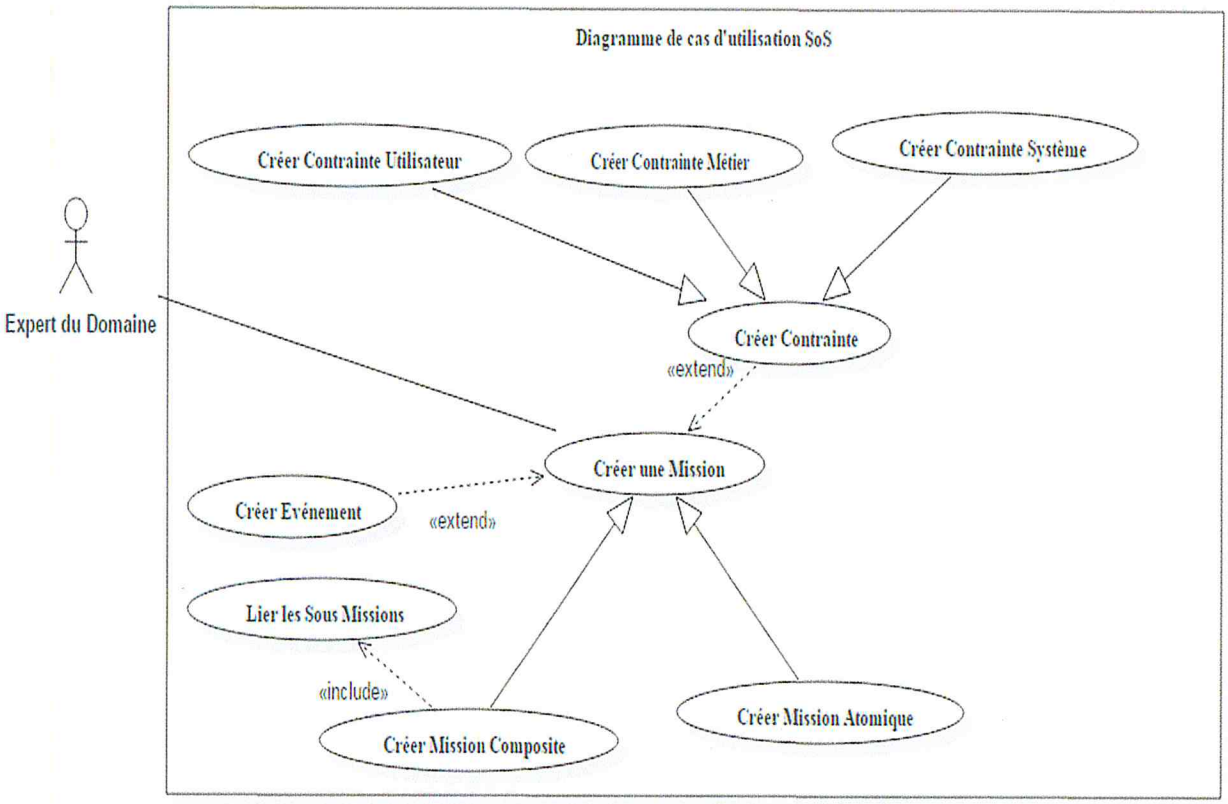


Fig IV.5 : Cas d'Utilisation de la Chaine d'Outils

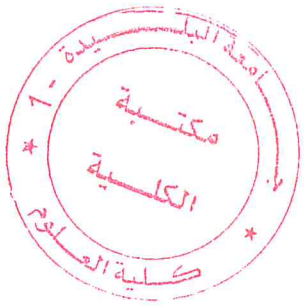
Notre diagramme de cas d'utilisation intitulé SoS composé d'un cas d'utilisation *Créer une Mission*, cette fonctionnalité permet à l'expert du domaine de créer une mission qui peut être soit une mission atomique ou bien une mission composite, cette dernière qui est composée de plusieurs missions que ça soit des missions composites ou des missions atomiques ou bien les deux, donc il doit lier ces sous missions avec la super mission, de plus il peut crée pour

chaque mission une ou plusieurs événements, et il peut également crée une contrainte qui peut être soit une contrainte système ou une contrainte métier ou bien une contrainte utilisateur.

IV. Conclusion

Dans ce chapitre nous avons présenté notre méta-modèle à travers une démarche, cette dernière est composée de deux phases, dans la première nous avons fait une analyse ontologique afin de déduire les concepts nécessaires. A partir de cette dernière nous avons identifié les constructions sémantiques qui nous ont permis d'élaborer notre méta modèle. Et enfin nous avons conçu dans la deuxième phase les différentes constructions graphiques, à l'aide des quatre principes de Moody.

Dans le chapitre suivant nous allons présenter l'implémentation de notre méta modèle. Nous présentons aussi l'environnement de notre travail, ainsi que les outils de développement utilisés avec une étude de cas.



Chapitre V

Implémentation

I. Introduction

Après la proposition de notre méta modèle, nous allons dans cette partie présenter son implémentation et sa mise en œuvre. En première partie nous présentons l'environnement de travail ainsi que les outils de développements utilisées, ensuite nous présentons notre application et son utilisation à l'aide d'une étude de cas.

II. Environnement de travail

1. Langages utilisées

Le choix du langage de programmation représente une étape très importante dans la réalisation de n'importe quelle application.
Pour la réalisation de notre application nous avons utilisé :

1.1. Le langage de programmation JAVA

Java est un langage de programmation et une plate-forme informatique qui ont été créés par Sun Microsystems en 1995. Beaucoup d'applications et de sites Web ne fonctionnent pas si Java n'est pas installé et leur nombre ne cesse de croître chaque jour. Java est rapide, sécurisé et fiable. Des ordinateurs portables aux centres de données, des consoles de jeux aux superordinateurs scientifiques, des téléphones portables à Internet, la technologie Java est présente sur tous les fronts.

2. Plugins Eclipse utilisés

2.1. Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) est un ensemble de plugins Eclipse qui peut être utilisé pour modéliser un modèle de données et générer un code ou une autre sortie en fonction de ce modèle. EMF a une distinction entre le méta-modèle et le modèle actuel. Le méta-modèle décrit la structure du modèle. Un modèle est une instance concrète de ce méta-modèle. EMF permet au développeur de créer le méta-modèle par différents moyens, par exemple XMI, annotations Java, UML ou un schéma XML. Il permet également de persister les données du modèle [36].

2.2. Graphical Modeling Framework (GMF)

GMF fournit l'infrastructure pour le développement d'éditeurs graphiques en se basant sur EMF et GEF (*Graphical Editing Framework)* [37]. Ainsi, GMF établie une liaison entre ces deux derniers *frameworks*[38].

III. Présentation de l'application

1. Créer Un Nouveau Diagramme

La figure V.1 illustre la création d'un nouveau diagramme, pour ce faire on doit cliquer sur : File → New → Mdl Diagram

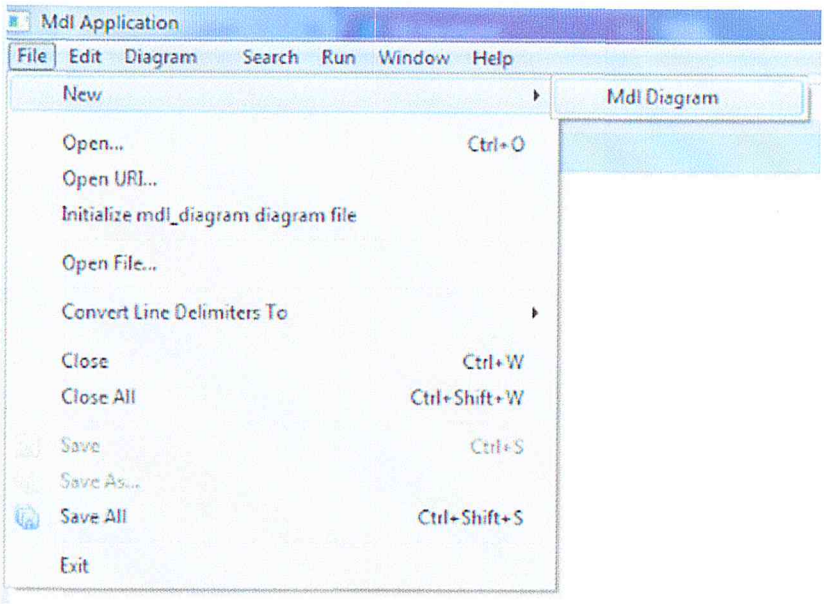


Fig V.1 : Création d'un Nouveau Diagramme Mdl

2. Dessiner un Diagramme Mdl

La figure V.2 illustre comment dessiner un diagramme Mdl, il y a deux partie dans la figure V.2, la partie 1 c'est notre palette qui contient une boite à outil des concepts nécessaires. Pour commencer on doit glisser l'un des éléments de la palette vers la partie 2 (la zone de modélisation.

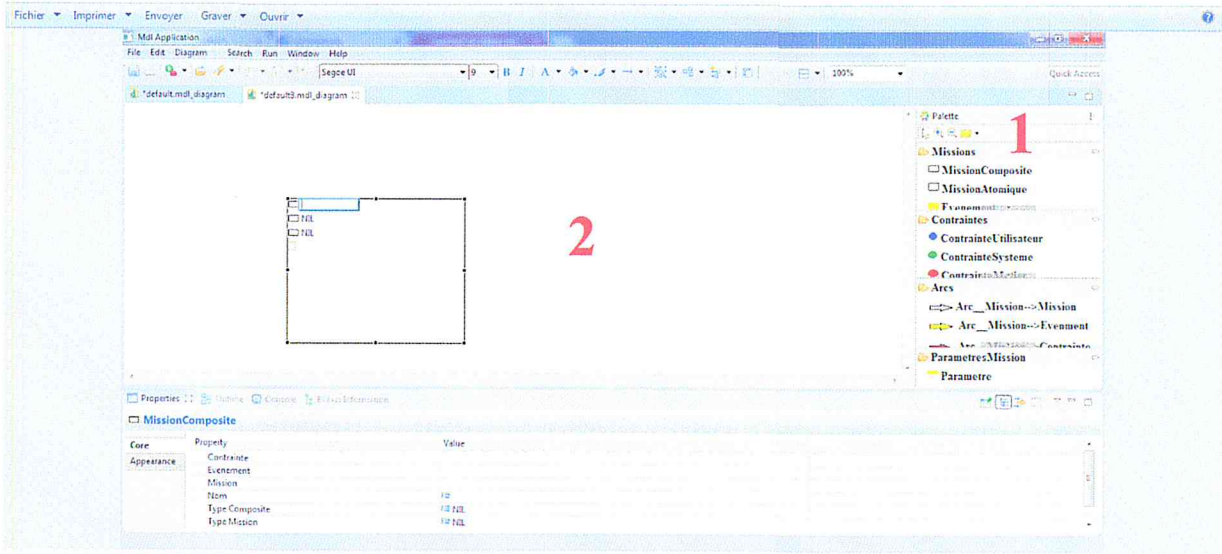


Fig V.2 : Dessiner Un Diagramme Mdl

3. Remplissage Des Propriétés Et La Validation

La figure V.3 illustre le remplissage des champs vide des attributs de concept choisi dans la zone des propriétés (partie 3), puis on valide la sémantique de notre diagramme par un clique droit sur le concept et puis OCL → validate (partie 4).

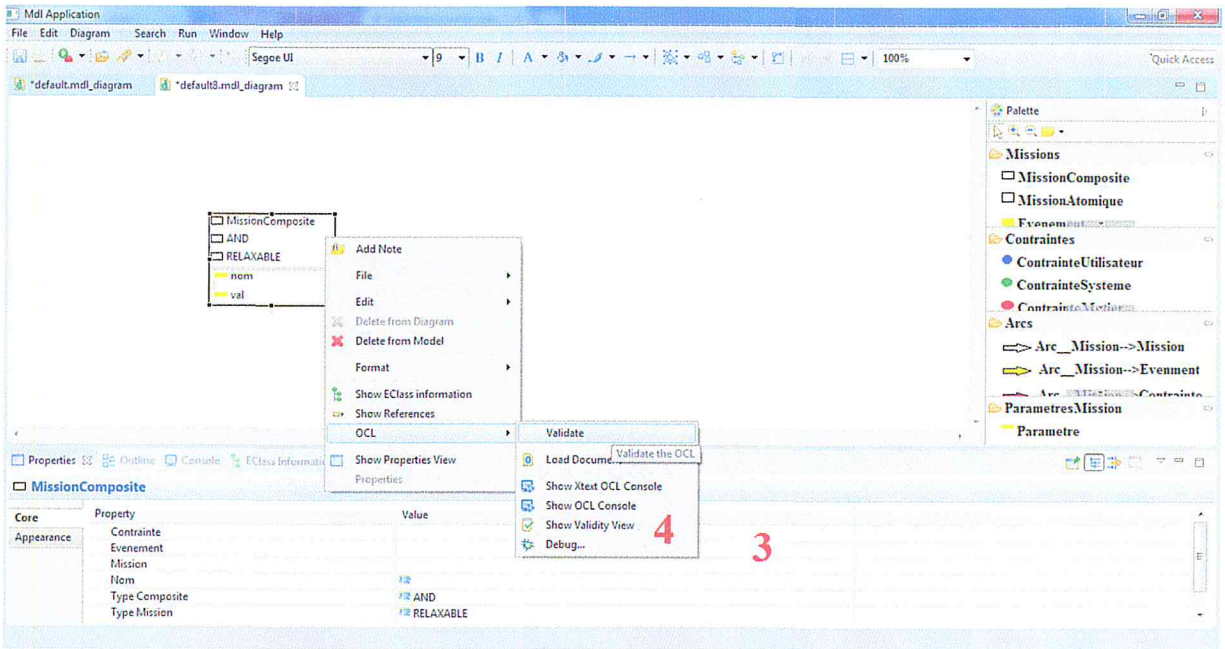


Fig V.3 : Remplissage et Validation

4. L'expression des contraintes

OCL fournit un langage de modélisation qui permet d'intégrer le comportement dans les méta-modèles structuraux ou fourni en complément de ces méta-modèles. En tant que langue de modélisation, OCL comprend les modèles et donc le code OCL est beaucoup plus compact que le Java équivalent. Le code OCL peut être vérifié de manière statique, tandis que le code Java correspondant utilise souvent la réflexion et ne peut donc pas être vérifié. Eclipse OCL est une implémentation de la spécification OMG OCL 2.4 à utiliser avec les méta-modèles Ecore et UML.

Dans notre projet nous avons utilisé le plugin OCL de Package Eclipse Modeling Tools pour vérifier la sémantique des modèles crées et la console OCL pour exprimer les contraintes sur les missions.

La figure V.4 illustre la console OCL qu'est composé de deux zones, une pour écrire la contrainte et l'autre pour le résultat.

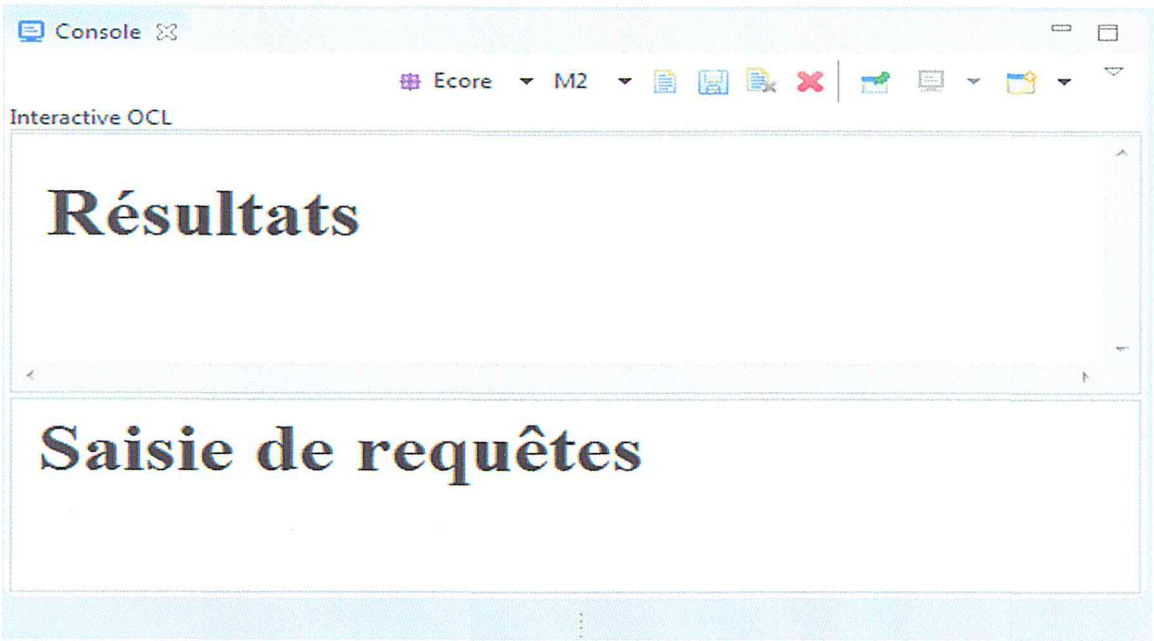


Fig V.4 : La Console OCL

La figure V.5 illustre un exemple de contrainte utilisateur appliqué sur un paramètre d'une mission.

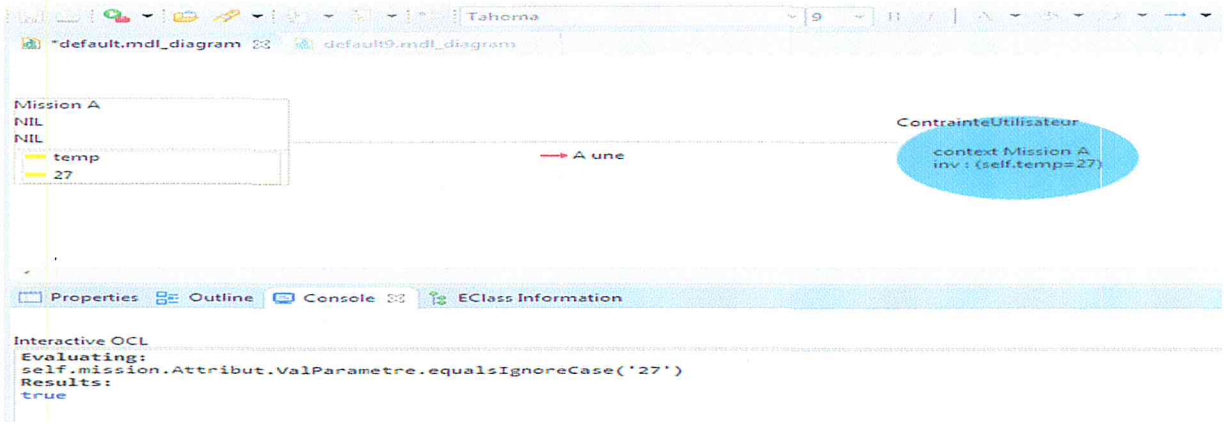


Fig V.5 : Exemple de Contrainte Utilisateur

IV. Etude de Cas « CROSS »

1. Présentation de CROSS

Le centre régional opérationnel de surveillance et de sauvetage(CROSS) est un centre qui prend en charge toute opération de sauvetage dès que l'incident a lieu sur le domaine maritime. Matériellement, les CROSS sont équipés de moyens de communication radios performants, de cartes précises sur lesquelles sont répertoriés les opérations en cours, les moyens disponibles, et les dangers particuliers. Ils sont dirigés par des administrateurs des Affaires Maritimes et sont armés par des personnels civils ou militaires de la Marine Nationale.

2. Modélisation de CROSS

Dans notre étude de cas nous considérons les différents systèmes comme des missions que ça soit des missions composites ou bien des missions atomiques comme il est illustré dans la figure V.6.

Fig V.6 : Structure Globale de CROSS

La mission CROSS est la super mission, elle est considérée comme une mission composite, donc elle est composée de cinq sous missions :

- La mission recherche et sauvetage maritime(RSM), cette mission est considérée comme prioritaire (obligatoire). Une mission RSM débute par la réception d'un signalement (alerte) d'un incident qui est considéré comme un événement, de plus l'impact de cette mission sur les autres missions est positif et élevé.
- La mission surveillance de la navigation maritime(SNM), c'est une mission atomique. Elle consiste à détecter et à identifier le trafic maritime, elle est considérée comme une mission obligatoire. L'impact de cette mission sur les autres missions est positif et élevé.
- La mission radiodiffusion d'information sur la sécurité maritime (RISM), c'est une mission atomique. Elle consiste à couvrir les zones maritimes dans le domaine de responsabilité de CROSS, elle est considérée comme une mission obligatoire. L'impact de cette mission sur les autres est positif et élevé.
- La mission surveillance des pêches maritimes(SDM), c'est une mission atomique .Les CROSS exercent le contrôle opérationnel des moyens nautiques et aériens engagés dans la surveillance des pêches, cette mission est considérée comme une mission secondaire (relaxable).l'impact de cette mission aux autres missions est positif mais faible.
- La mission surveillance de pollution(SDP), c'est une mission atomique. Elle consiste à faire la surveillance des pollutions maritimes, les CROSS reçoivent des informations de la part des douanes et la marine nationale qui est considéré comme un événement dans notre cas. L'impact de cette mission sur les autres missions est positif faible.

Afin de bien clarifier les détails de notre application, nous allons modéliser la mission RSM qui est une mission composite, cette dernière composée de deux sous missions :

- La mission de coordination des opérations(CDO), c'est une mission qui permet de faire la coordination entre les différents systèmes de RSM. La mission CDO est une mission atomique, cette dernière est obligatoire mais son impact est moyen positif.
- La mission de localisation de lieu, elle permet de localiser le lieu où le secours doit être mené. C'est une mission composite obligatoire, ainsi que son impact est positif élevé.

La figure V.7 illustre le détail de la mission RSM.

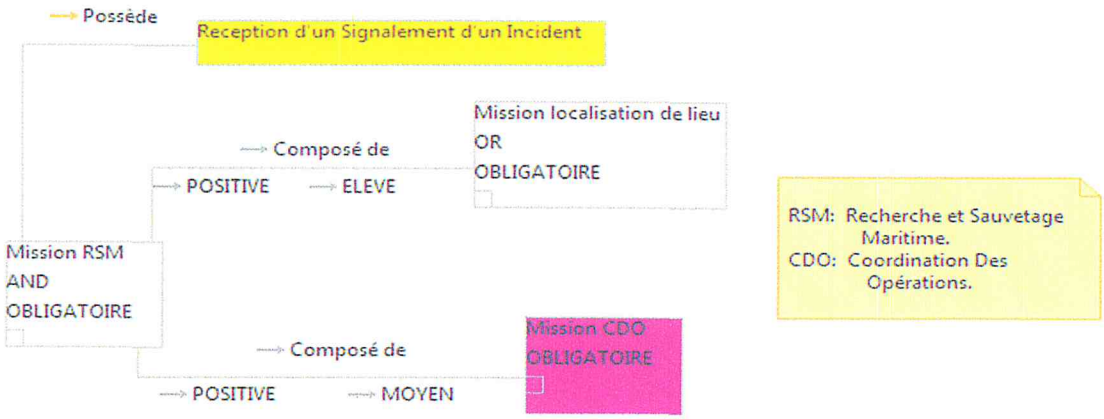


Fig V.7 : Structure de Mission RSM

Nous allons détailler dans ce qui suit la mission localisation de lieu qui est composée de deux sous missions composites illustré dans la figure V.8 :

- La mission géo localisation, nous aurons cette mission dans le cas ou nous pouvons identifier la zone précise où l'accident à eu lieu, cette dernière est une mission obligatoire possède un paramètre « GPS », ainsi qu'avec un impact positif élevé.
- La mission délimiter un périmètre de recherche (DPR), nous aurons cette mission dans le cas ou nous arrivons plus à établir les coordonnées géographiques, ce périmètre délimite la zone dans laquelle les recherches seront conduites. La mission DPR est une mission obligatoire avec un impact négatif faible. Elle contient deux paramètres, l'un qui donne la vitesse de vent et l'autre le courant et nous avons bien contrôlé ces deux paramètres par une contrainte métier.

Ces deux missions sont des missions composites ont les même sous missions car nous ne pouvons pas démarrer les deux sous missions à la fois, donc ils sont composées des sous missions atomiques suivantes :

- La mission service de soin (SS), c'est une mission atomique obligatoire, elle possède deux événements, la fin de la localisation et la réception d'une information auprès des autorités compétentes, cette mission est lancée lorsque ces deux événements se produisent. L'impact de cette mission est positif moyen.
- La mission déterminer les moyens adéquats (DMA), ces moyens peuvent être adaptés en fonction de l'évolution de la situation, elle possède un paramètre de l'adaptation

des moyens et une contrainte métier sur ce paramètre, elle possède également un événement de fin de localisation avec un impact positif élevé.

- La mission rédaction des comptes rendus (RCR), cette mission met fin à toutes les actions en cours et en rédigeant un compte rendu final, c'est une mission obligatoire avec un impact positif faible et un événement de fin de RSM.
- La mission prévenir les familles et fournir le moyen ad hoc[1] (PFFM), c'est une mission obligatoire avec un impact positif moyen et un événement de fin de la localisation.

La figure V.8 illustre tous les détails que nous avons parlés.

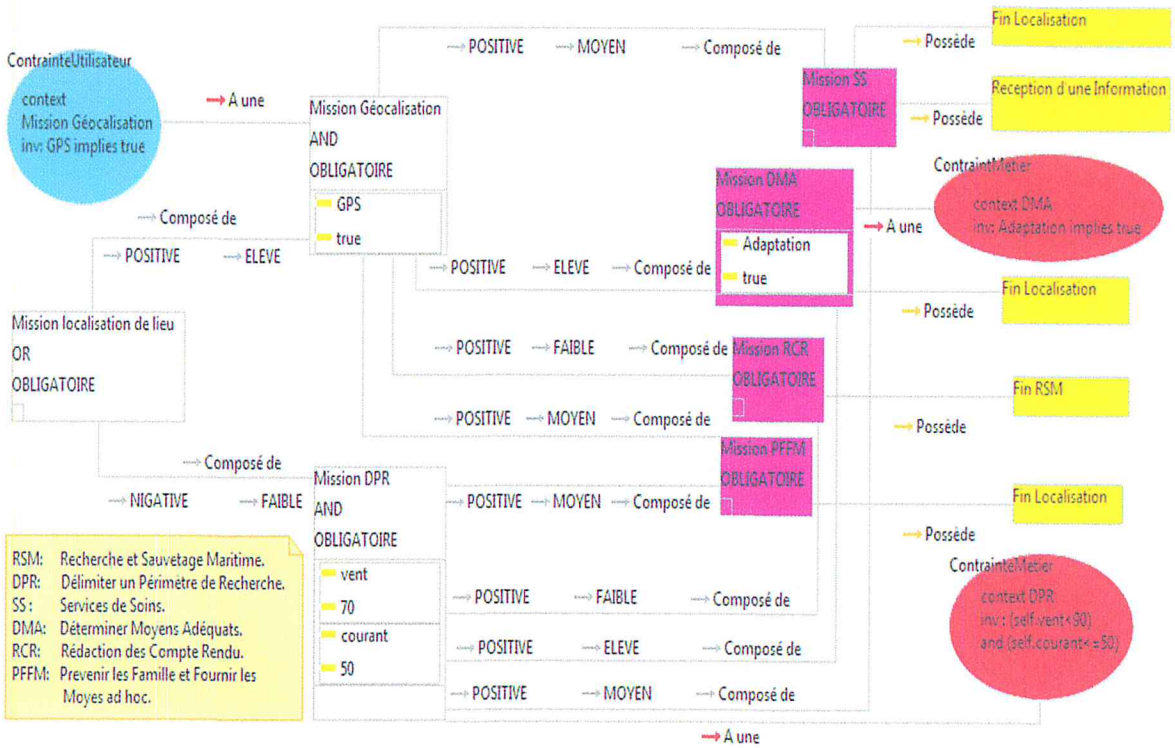


Fig V.8 : Détails de La mission localisation de lieu

V. Conclusion

Nous avons présenté dans ce chapitre l'implémentation de notre méta modèle. Nous avons entamé ce chapitre par définir l'environnement de travail, ainsi que les différents outils de développements utilisés. Nous avons présenté également l'application que nous avons développée en utilisant une étude de cas.

1 Adéquat

Conclusion Générale

Conclusion Générale

Les travaux présentés dans ce mémoire se situent dans le domaine de génie logiciel, ils portent exactement le volet de spécification des systèmes de systèmes.

Ce mémoire présentait notre proposition d'expression des contraintes OCL. Notre proposition avait pour but de proposer un méta modèle en prenant en considération la définition des contraintes dans la spécification d'une mission.

Afin d'élaborer ce projet, nous avons commencé le travail par faire une analyse ontologique qui nous permettra d'identifier les concepts principaux afin d'obtenir des constructions sémantiques (notre méta modèle finale), a partir de ces derniers nous avons proposé des constructions graphiques correspondants aux constructions sémantiques obtenues.

Nous avons développé une application dédiée pratiquement aux experts de domaine d'application afin de représenter les missions qui peuvent être composites ou atomiques et d'exprimer pour chacune ses propres contraintes. La sémantique du modèle crée est vérifiée par le plugin OCL qui est intégré dans éclipse.

L'apport que nous avons essayé d'apporter au domaine des systèmes de systèmes est intéressant, mais n'est en réalité qu'une ouverture vers d'autres travaux. .

Plusieurs perspectives pour ce travail sont à envisager :

- Etendre le méta-modèle pour qu'il soit capable d'exprimer les sous systèmes qui réalisent la mission.

- Etendre notre outil pour qu'il soit capable de générer l'architecture abstraite des systèmes.

BIBLIOGRAPHIE

Bibliographies:

[1] Pei, R. S. (2000). System of Systems Integration (SoSI)-A" SMART" Way of Acquiring Army C412WS Systems. In *Summer Computer Simulation Conference* (pp. 574-579). Society for Computer Simulation International; 1998.

[2] Jamshidi, M. (2005). Theme of the IEEE SMC 2005, Waikoloa, Hawaii, USA. *2005-10-10]. http://ieeesmc2005. unm. edu.*

[3] Carlock, P. G., & Fenton, R. E. (2001). System of Systems (SoS) enterprise systems engineering for information-intensive organizations. *Systems engineering*, *4*(4), 242-261.

[4] Stanley N. H.(2013,November). Modeling and Simulation for Systems and System-of-Systems Engineering.

[5] Maier, M. W. (1996, July). Architecting principles for systems-of-systems. In *INCOSE International Symposium* (Vol. 6, No. 1, pp. 565-573).

[6] Sage, A. P., & Cuppan, C. D. (2001). On the systems engineering and management of systems of systems and federations of systems. *Information knowledge systems management*, *2*(4), 325-345.

[7] Dominique L,Jean-René R.(2010). *System of Systems*.

[8] Maier, M. (1998). Architecting principles for systems-of-systems. *Systems Engineering* 1, no. 4: 267-284.

[9] Dahmann, J. and K. Baldwin. (2008). Understanding the current state of US defense systems of systems and the implications for systems engineering. *Proceedings of the IEEE Systems Conference*, April 7-10, in Montreal, Canada.

[10] Beale, D., & Bonometti, J. (2016). Chapter 2: Systems Engineering (SE)–The Systems Design Process. *Auburn University. Available in: https://goo. gl/YtuUan. Accessed in June 28th.*

[11] OMG Systems Modelling Language Group.(2015). Omg systems modeling language specification. Technical Report ptc/06-05-04, Object Management Group.

[12] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, *8*(3), 203-236.

[13] van Lamsweerde, A. (2008, November). Requirements engineering: from craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (pp. 238-249). ACM.

[14] P. Model and N. Group.(2011). Business Process Model and Notation (BPMN). Technical report, Object Management Group.

[15] Højsgaard, E., & Hallwyl, T. (2012, March). Core BPEL: syntactic simplification of WS-BPEL 2.0. In *Proceedings of the 27th annual ACM symposium on applied computing* (pp. 1984-1991). ACM.

[16] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems, 8(3), 203-236.

[17] Silva, E., Batista, T., & Cavalcante, E. (2015, May). A mission-oriented tool for system-of-systems modeling. In *Software Engineering for Systems-of-Systems (SESoS), 2015 IEEE/ACM 3rd International Workshop on* (pp. 31-36). IEEE.

[18] Jaziri, W. (2004). *Modélisation et gestion des contraintes pour un problème d'optimisation sur-contraint: application à l'aide à la décision pour la gestion du risque de ruissellement* (Doctoral dissertation, INSA de Rouen).

[19] Claude, B. Le langage UML 2.0 OCL (Object Constraint Language). [document électronique].France, https://www.google.dz/?gws_rd=cr,ssl&ei=77NuV-7MKoSuU8XggYgK#q=01_lelangageocl

[20] Laetitia, M. OCL - Object Constraint Language. [document électronique].France,2012, https://fr.slideshare.net/niazi2012/cours-ocl

[21] Marianne, H. Object Constraint Language (OCL) Une introduction. [document électronique].France,2008, https://www.google.dz/?gws_rd=cr,ssl&ei=77NuV-7MKoSuU8XggYgK#q=coursOCL20

[22] Philippe,C.OCL : contraintes sur métamodèles. [document électronique].France,2014, https://www.google.dz/?gws_rd=cr,ssl&ei=77NuV-7MKoSuU8XggYgK#q=3_contraintes_ocl

[23] Gil, J., Howse, J., & Kent, S. (2001). Towards a formalization of constraint diagrams. In *Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on* (pp. 72-79). IEEE.

[24] Stapleton, G., Thompson, S., Fish, A., Howse, J., & Taylor, J. (2005). A new language for the visualisation of logic and reasoning.

[25] Bottoni, P., Koch, M., Parisi-Presicce, F., & Taentzer, G. (2000, October). Consistency checking and visualization of OCL constraints. In *International Conference on the Unified Modeling Language* (pp. 294-308). Springer Berlin Heidelberg.

[26] Bottoni, P., Koch, M., Parisi-Presicce, F., & Taentzer, G. (2001, October). A visualization of OCL using collaborations. In *International Conference on the Unified Modeling Language* (pp. 257-271). Springer Berlin Heidelberg.

[27] Fish, A., Howse, J., Taentzer, G., & Winkelmann, J. (2005). Two visualizations of OCL: A comparison.

[28] Eric,C. Du modèle au métamodéle Syntaxes abstraites et concrètes.[document électronique].France,2008, https://www.google.dz/?gws_rd=cr,ssl&ei=77NuV-7MKoSuU8XggYgK#q=MetaModelisation2008par6

[29] Eric,C. Ingénierie des Modèles Méta-modélisation.[document électronique].France,2008, https://www.google.dz/?gws_rd=cr,ssl&ei=77NuV-7MKoSuU8XggYgK#q=Ing%C3%A9nierie+des+Mod%C3%A8les+M%C3%A9ta-mod%C3%A9lisation

[30] Blanc, X. (2005). *MDA en action: ingénierie logicielle guidée par les modèles*. Editions Eyrolles.

[31] Mallouli, S. D., Assar, S., & Souveyet, C. (2011, May). Pour une perspective comportementale dans les méta-modèles de processus. In *INFORSID* (pp. 351-366).

[32] Moody, D. (2009). The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756-779.

[33] El-Attar, M., Luqman, H., Karpati, P., Sindre, G., & Opdahl, A. L. (2015). Extending the UML statecharts notation to model security aspects. *IEEE Transactions on Software Engineering*, *41*(7), 661-690.

[34] Czarnecki, K., Eisenecker, U. W., Goos, G., Hartmanis, J., & van Leeuwen, J. (2000). Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, *15*.

[35] Paivio, A. (1990). *Mental representations: A dual coding approach*. Oxford University Press

[36] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

[37] Projet Eclipse. « Eclipse Graphical Modeling Framework ». http://www.eclipse.org/modeling/gmp/?project=gmf-notation#gmf-notation (Page consultée le 01 juin 2017).

[38] Eclipse Wiki, « GMF Tutorial », http://wiki.eclipse.org/index.php/GMF_Tutorial (Page consultée le 01 juin 2017).