الجمهوريـة الجزائـريـة الديمقراطيـة الشعبيـة
République Algérienne Démocratique et Populaire
وزارة التعليـم العالـي والبحـث العلمـي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

N° Réf :……

Centre Universitaire
Abd elhafid Boussouf Mila

Institut des sciences et de la technologie **Département de Mathématiques et Informatiques**

Mémoire préparé en vue de l'obtention du diplôme de Master

En : Informatique

Spécialité : Sciences et Technologies de l'information et de la communication (STIC)

Vérification d'une approche basée transformation de graphes en IDM

Préparé par :

- Boucheloukh Ibtissem

Soutenue devant le jury

Encadré par : Lalouci Ali…….....M.A.B

Aouag Mouna……M.C.B

Président : Boubakir Mohammed……M.A.B

Examinateur : Djaaboube Salim……M.A.A

Année universitaire : 2015/2016

Remerciements

- En premier lieu, nous tenons à remercier notre DIEU "allah", notre créateur de nous avoir donné la force pour accomplir ce travail.
- Nos remerciements à nos très chers parents, frères, soeurs, collègues et amis respectives qui nous ont encouragés, soutenu durant tout notre
- Nous adressons nos vifs remerciements à notre encadreur consultant Mr.Lalouci Ali pour nous avoir guidés tout au long de ce travail, pour sa compréhension, sa patience, sa compétence, et ses remarques qui nous ont été précieuses.
- Un grand merci sera également adressé au M.Aouag Mouna.

- Nous présentons nos chaleureux remerciements aux enseignants du jury qui m'ont fait l'honneur de présider et d'examiner ce modeste travail.

Remerciements parcours

Ibtissem Bouchelouk...

DEDICACES

Avec toute l'ardeur de mes sentiments je dédie ce modeste mémoire à ma famille. A toi celle qui la présence l'affection la tendresse celle qui a fait de moi ce que je suis aujourd'hui à toi ma mère sans oublier mon cher père.

- *A Mes chers frères, et Mes Chères sœurs, je vous réserve toujours une place dans mon cœur et mes pensées.*
- *A mes chers neveux et nièces : Seifeeddinne, Charafeddinne , Malak,Nada ,Ritej,Rihem, Batoul,Assil et bien sur Elmoatassimbiallah*
- *A la mémoire de ma grande mère Tourkia que dieu l'accueille en son vaste paradis.*
- *A mes oncles Abdallah et Ismail paix à leurs ames.*
- *Et à toutes mes amis spécialement : Amel, Souad, Amina, Ikram, Amel, Sana, Safa, Maissa,*
- *A tous les membres de la promotion 2016 et Surtout : Asma, Amel, Amira, Samira,*
- *A tous ceux qui j'ai oublié de citer leurs noms et qu'ils sont Importants pour moi.*

Ibtissem Boucheloukh...

Résumé

L’ingénierie dirigée par les modèles a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s’agit d’une forme d’ingénierie générative dans laquelle tout ou partie d’une application est engendrée à partir de modèles. L’IDM se focalise sur l’exploitation de modèles, méta-modèles, et la transformation de modèle qui définit l’automatisation qui peut être utilisé dans le processus de développement de logiciels. Afin d’assurer la qualité du produit final, Il est donc nécessaire d’augmenter la vérification d’une telle transformation de modèles. L’objectif principal de notre travail est de définir un modèle de transformation, afin de vérifier la consistance de la transformation entre les diagrammes de communications orienté objet et les diagrammes de communications orienté aspect en utilisant l’outil USE et le langage OCL.

Mots clés : Ingénierie Dirigée par les Modèles, Transformation de modèle, modèle de transformation, Modélisation, Vérification, Validation, Langage OCL, L’outil USE.

Table des matières

Chapitre 3 : Vérification des transformations de modèles

Chapitre 4 : *Vérification d'une approche de transformation de modèles*

Tables des figures

Acronymes

IDM : Ingénierie **D**irigée par les **M**odèles.

MDA : Modèle **D**riven **A**rchitecture.

JMI : Java **M**etadata **I**nterface.

LHS : Left **H**and **S**ide.

RHS : Right **H**and **S**ide.

OMG : Object **M**anagement **G**roupe.

CIM : Computation **I**ndependent **M**odel.

PIM : Platform **I**ndependent **M**odel.

PDM : Platform **D**escription **M**odel.

PSM : Platform **S**pecific **M**odel.

UML : Unified **M**odeling **L**anguage.

MOF : Meta **O**bject **F**acility.

XMI : XML **M**etadata **I**nterchange.

OCL : Object **C**onstraint **L**anguage.

SPEM : Software **P**rocess **E**ngineering **M**etamodel.

CWM : Common **W**arehouse **M**etamodel.

MOFM2T : MOF Model-**to**-**T**ext language.

EDOC : Enterprise **D**istributed **O**bject **C**omputing.

CASE : Case **A**ided **S**oftware **E**ngineering.

MIC : Model **I**ntegrated **C**omputing.

MOA : Modélisation **O**rientée **A**spect.

USE : UML based **S**pecification **E**nvirrenement.

KMF : Kent **M**odeling **F**ramework.

Introduction

Générale

Problématique

De nos jours les systèmes informatiques jouent un rôle important dans plusieurs domaines d'applications. Ils deviennent de plus en plus complexes. De ce fait, le développement de ces systèmes est toujours plus complexe, et a mis en évidence la nécessité de disposer d'outils, de méthodes et de processus permettant d'assurer la maîtrise tout au long de leur cycle de vie. A ce titre, l'Object Management Group à proposer l'ingénierie dirigée par les modèles.

L'ingénierie dirigée par les modèles (IDM) est une forme d'ingénierie générative, qui se singularise par une démarche par laquelle tout ou partie d'un système informatique est généré à partir de modèles. IDM se base sur les deux notions de méta-modélisation et de transformation de modèles permettant d'assister les développeurs au cours du cycle de développement des systèmes complexes. Une transformation de modèles, quel que soit son type, s'apparente toujours comme une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un autre ensemble de modèles. Les modèles en entrée et en sortie sont tous structurés par leur méta-modèle. Pour assurer une meilleure transformation, il faut que les modèles soient suffisamment précis pour être interprétés ou transformés de façon automatique ou semi-automatique. Mais, le défi de l'IDM est de répondre aux problèmes de la vérification de modèles et de transformations.

Objectif (Contribution)

Avec le succès d'UML, la possibilité de transformer des modèles vers des programmes ou des spécifications formelles devient une clé pour la génération de code automatique ou la vérification dans le processus de développement de logiciels.

La plus part des travaux dans le cadre de transformation de modèles UML ne prend pas en charge les problèmes de vérification et validation. Dans [6] l'auteur ont proposé une approche pour la transformation des diagrammes UML 2.0 vers les diagrammes orientés aspect à l'aide de transformation de graphes. Mais, l'auteur n'a pas vérifié et valider sa travail.

Dans la littérature, ils existent plusieurs méthodes pour la vérification et la validation de la transformation de modèles. L'objectif de notre travail est d'appliquer l'approche USE proposée par Martin Gogola et al [21] pour la vérification formelle de ces transformations. Pour cela, nous allons proposer un modèle de transformation modélisant la transformation entre les diagrammes de communication orienté objet et les diagrammes de communications orienté aspect afin de vérifier la propriété de consistance avec leurs trois options.

Structure

Dans le reste, nous détaillons ces idées à travers le plan du présent manuscrit :

- **Le premier chapitre** détaille le contexte de l'ingénierie dirigée par les modèles en présentant les concepts de base ainsi que la transformation de modèles et par la suite, l'intégration des méthodes formelles ou nous présentons une définition simple de ces méthodes et les différentes classifications existantes de ces méthodes.
- **Le deuxième chapitre** aborde en détaille l'approche orienté objet en présentant les concepts de base, et on met en particulier l'accent sur les deux langages objets UML et OCL. Dans la dernière partie de ce chapitre nous présenterons l'approche orientée aspect. nous allons voire les concepts de base ainsi que les techniques et les technologies Orientées Aspect.
- **Le troisième chapitre** présente les différentes vues des transformations ainsi que les propriétés et les techniques de vérification existante dans la littérature concernant les deux vues. En particulier, nous allons présenter l'approche USE.
- **Le quatrième chapitre** présente l'approche étudiée et décrit en détail notre modèle de transformation proposé pour cette dernière. Finalement, Nous allons vérifier la propriété de consistance ainsi que leurs trois options.

Chapitre 01

Ingénierie Dirigée par Les Modèles

Au sommaire de ce chapitre

1. **Introduction**
2. **Concepts de base de l'IDM**
3. **Transformation demodèles**
4. **Les approches de l'ingénierie dirigée par les modèles**
5. **Intégration des méthodes formelles à l'IDM**
6. **Conclusion**

1. Introduction

L'ingénierie dirigée par les modèles (IDM) a permet de prendre en charge la croissance de la complexité des systèmes logicielle développé, où la modélisation de ces systèmes est basée sur la définition de modèles qui décrivent la transformation de modèles ainsi que l'intégration des méthodes formelles.

Dans ce chapitre nous allons présenter les concepts de base de l'IDM. Ensuite, nous décrivons la transformation de modèles. Nous abordons les critères de transformation de modèles ainsi que la classification de différentes approches de transformation de modèles. Enfin, nous allons voir l'intégration des méthodes formelles en IDM.

2. Concepts de base de l'IDM

L'IDM est une approche de développement mettant à disposition des outils, des concepts et des langages pour créer et transformer des modèles afin de mécaniser le processus de développement. L'IDM se concentre sur une préoccupation plus abstraite que la programmation classique ce qui permet d'obtenir plusieurs améliorations dans le développement de systèmes complexe [1].

2.1. Système complexe

Un Système complexe est un ensemble d'éléments organisé en interaction permanente entre eux. Cet ensemble forme un tout cohérent et intégré pour assurer une ou plusieurs fonctions correspondant à la finalité du système [2].

2.2. Modèle et Méta-modèle

Pour modéliser un système dans le domaine de l'IDM, il est intéressant de bien comprendre et de bien maitriser les notions de modèle et Meta modèle.

Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé [3].

Un métamodèle est un modèle qui définit le langage d'expression d'un modèle c.-à-d. le langage de modélisation. La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé [3].

Donc on peut dire qu'un méta modèle est souvent défini comme un modèle d'un modèle. Un modèle n'est utilisable que si son langage de modélisation est précisé. Dans le sens de l'IDM, un modèle doit être conforme à un méta modèle pour avoir une bonne définition d'un système (voir figure 1.1).

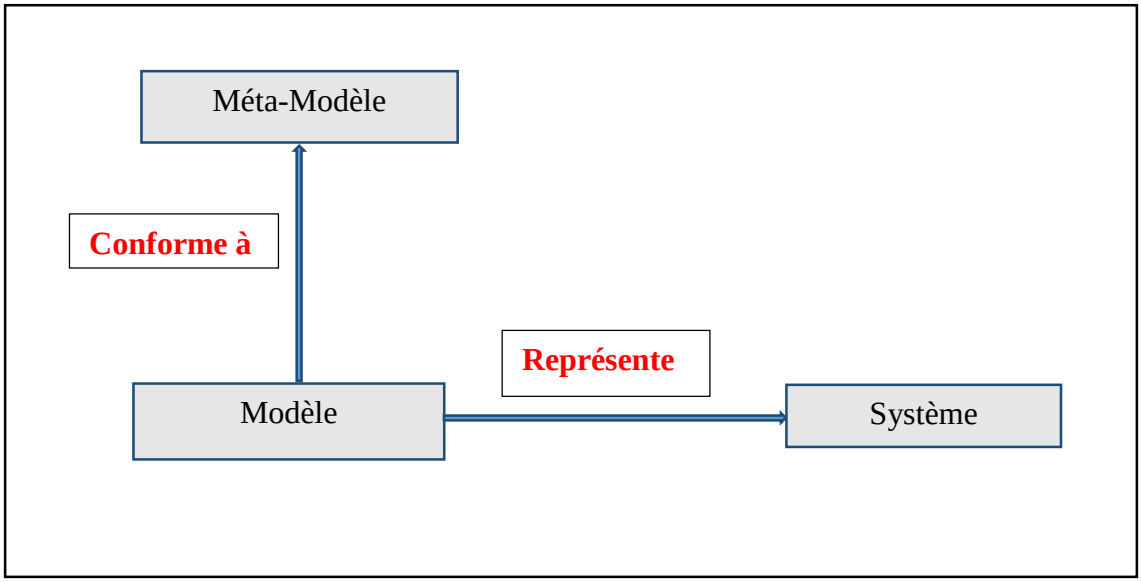

Figure 1.1 : Concepts de base en ingénierie des modèles.

2.3. Méta modélisation

Le méta modélisation est une activité de modélisation qui consiste à mettre en œuvre un langage. Pour cette activité, il est nécessaire de définir sa structure, sa sémantique et son langage de modélisation. Ces derniers sont représentés par un méta modèle qui illustre une définition, formelle d'un modèle. En d'autres termes, le méta modèle exprime les éléments, la structure ainsi que la sémantique de ces modèles. En plus défini la relation existante entre un modèle et le système à modéliser. La mise en œuvre de cette relation représente alors le méta modélisation. Le méta modélisation est une activité qui consiste à définir des métas modèles contemplatifs qui reflètent la structure statique des modèles [4].

2.4. Langage de modélisation

Un langage de modélisation est tout langage artificiel utilisé pour exprimer une information ou une connaissance ou pour décrire une structure ou un système défini par un ensemble consistant de règles. Ces dernières pouvant être utilisées pour l'interprétation du sens des composants dans la structure. Chaque langage possède une syntaxe et une sémantique. La syntaxe décrit les différentes constructions du langage, alors que la sémantique permet de donner un sens à chacune des constructions du langage [3].

3. Transformation de modèles

3.1. Définitions ou concept de bases

La transformation de modèles est une opération qui consiste à générer un ou plusieurs modèles cibles conformément à leur méta-modèle à partir d'un ou de plusieurs modèles sources conformément à leur méta-modèle. En autre, les méta-modèles source et cible peuvent être les mêmes dans certaines situations [6].

Dans la figure (1.2), nous présentons les concepts de base de la transformation de modèles.

- **Transformation** : une transformation de modèles est une génération automatique d'un modèle cible à partir d'un modèle source selon la définition de la transformation.
- **Définition de transformation** : C'est un ensemble des règles de transformation qui décrivent comment un modèle source peut être transformée à un modèle cible.
- **Règle de transformation** : une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un langage source peuvent être transformées en une ou plusieurs constructions dans un langage cible.

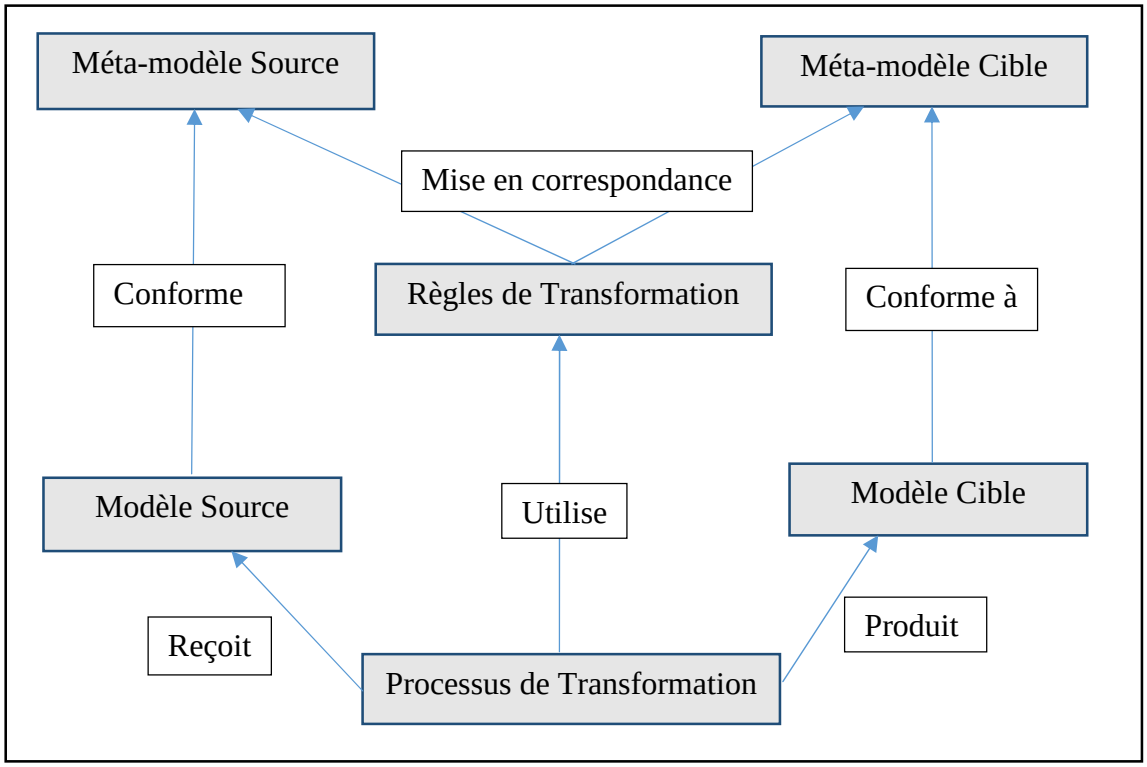

Figure 1.2 : Concepts de base de la transformation de modèles.

3.2. Pour quoi la transformation de modèles

La transformation de modèles a des objectifs divers comme la génération automatique du code, la translation de modèle, la migration, et l'ingénierie inverse que nous détaillerons dans ce qui suit [4].

3.2.1. Génération automatique de code

Dans la technologie de l'IDM la principale activité est la génération automatique de code et été utilisé dans la littérature de recherche à des buts différents. En générale l'objectif est de transformer un modèle vers un modèle spécifique représentant un code source pouvant être exécuté.

3.2.2. Translation de modèle

La translation de modèle est la transformation de modèle à un modèle équivalent mais dans une différente représentation. Il est souhaitable d'utiliser et d'échanger des modèles entre les différents outils de modélisation et même entre les différents langages de modélisation.

3.2.3. Migration de modèles

La migration de modèles définis comme une transformation d'un modèle écrit dans un langage en un autre modèle écrit dans un autre langage en gardant le même niveau d'abstraction des modèles.

3.2.4. Ingénierie inverse

L'ingénierie inverse est l'inverse de la génération de code, et permet de comprendre la structure du programme. Prenant le code source en entrée, elle permet de construire un modèle mental ou visuel à un niveau supérieur d'abstraction, afin de faciliter la compréhension du code et connaitre comment il est structuré.

3.3. Classification des approches de transformation

Dans la littérature, il existe plusieurs classifications des approches de transformation de modèle. Généralement, on distingue deux classe principales : les transformations de « modèle à code source » et les transformations de « modèle à modèle ». Pour chacune de ces deux catégories, on distingue plusieurs sous-catégories [9].

3.3.1. Transformations de type Modèle vers code

Dans cette catégorie, on retrouve les approches basées sur le parcours de modèles et celles basées sur l'utilisation de template.

- **Approche basées sur le parcours de modèles :** La transformation consiste à fournir un certain mécanisme de parcours de la représentation interne du modèle source et l'écriture du code dans un flux en sortie.
- **Approche basées sur les templates :** Ces approches sont les plus couramment utilisées dans les outils MDA actuels de génération de code. Un template consiste en un fragment de texte contenant des bouts de méta code permettant :
  - ✓ D'accéder aux modèles sources
  - ✓ D'effectuer une sélection de code
  - ✓ De réaliser des expansions itératives

La structure du template est généralement très proche du code à générer.

3.3.2. Transformations de type modèle vers modèle

Les transformations de type modèle vers modèle consistent à transformer un modèle source en un modèle cible, ces modèles peuvent être des instances de différents métamodèles. Elles offrent des transformations plus modulaires et faciles à maintenir. Dans les cas où on trouve un grand espace d'abstraction entre PIMs et PSMs, il est plus facile de générer des modèles intermédiaires qu'aller directement vers le PSM cible. Les modèles intermédiaires peuvent être utiles pour l'optimisation ou bien pour des fins de débogage. De plus, les transformations de type modèle vers modèle sont utiles pour le calcul des différentes vues du système et leurs synchronisations.

- **Approches de manipulation directe :** Ces approches offrent une représentation interne des modèles source et cible, et un ensemble d'APIs pour les manipuler. Elles sont généralement implémentées comme un cadre structurant orienté objet qui peut également fournir un ensemble minimal de concepts pour organiser les transformations (exemple : classe abstraite). Cependant, l'utilisateur doit implémenter et ordonnancer les règles de transformation. JMI (Java Metadata Interface) peut être cité comme exemple de ces approches.
- **Approches relationnelles** : Cette catégorie regroupe les approches déclaratives et utilise les relations mathématiques. L'idée de base est de spécifier les relations entre les éléments des modèles source et cible par des contraintes, ces derniers peuvent avoir une sémantique exécutable dans la programmation logique. Cependant, les relations ne sont pas exécutables mais peuvent être lues dans les deux sens.
- **Approches dirigées par la structure :** Les approches de cette catégorie distinguent deux phases : la première consiste à créer la structure hiérarchique du modèle cible, la

seconde définit les attributs et les références dans la cible. L'environnement définit l'enchaînement des règles, et l'utilisateur ne fournit que les règles de transformation.

- **Approches hybrides :** Les approches hybrides combinent les différentes techniques des catégories précédentes. Les langages de transformation de cette classe combinent les aspects déclaratif et impératif. Cette combinaison est représentée en général par des règles de mapping définissant les relations entre les éléments sources et cibles et des règles opérationnelles définissant les actions de la transformation.
- **Approches basées sur les transformations de graphes** : Ces approches sont déclaratives, elles se basent sur la théorie des transformations de graphe. Les règles de transformation sont définies pour des fragments de modèles, ces fragments peuvent être exprimés dans les syntaxes concrètes des modèles sources et cibles respectivement. Chaque règle est composée d'un graphe source (LHS : Left Hand Side) et d'un graphe cible (RHS : Right Hand Side). La transformation consiste à conserver les entités filtrées (LHS) et ajouter des nouvelles entités conformément à la règle (RHS). Les règles de transformation de graphes constituent ce que l'on appelle la grammaire de graphes.

4. Les approches de l'ingénierie dirigée par les modèles

L'IDM peut être considérée comme un domaine qui a émergé avec les technologies liées à l'instrumentation des modèles. Il existe différentes approches concrétisant différentes façons d'utiliser les modèles dans leur processus de développement des systèmes. L'approche la plus connue et peut-être la plus développée est l'approche MDA. Nous présentons cette approche dans la sous-section suivante, avant d'évoquer brièvement d'autres approches existantes [7].

4.1. L'approche MDA

L'approche MDA (Modèle Driven Architecture) est un standard proposé par l'OMG dans le domaine de conception des applications afin d'apporter une nouvelle façon de conception et permettre une réutilisation des modèles.

Le principe de l'approche MDA est de séparer les spécifications fonctionnelles d'un système des spécifications techniques de son implémentation sur une plateforme donnée. Autrement dit cette approche permet de réaliser le même modèle sur plusieurs plates-formes grâce à des projections. La mise en œuvre du MDA est entièrement basée sur lesmodèles et leurs transformations. [7].

4.1.1. Les types de modèle dans MDA

Dans un processus de développement orienté MDA, tout est considéré comme modèle. En partant de la phase de spécification des exigences jusqu'à la phase d'implémentation, les modèles sont utilisés pour représenter les artefacts logiciels manipulés durant le processus de construction de l'application [8]. Ainsi, MDA identifie quatre types de modèles : CIM, PIM, PDM et PSM.

- **Le CIM (Computation Independent Model) :** appelé aussi modèle de domaine ou modèle métier, modélise les exigences du système. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier.
- **Le PIM (Platform Independent Model) :** décrit le système sans montrer les détails de son utilisation sur une plate-forme particulière. Dans MDA, il est possible d'élaborer plusieurs modèles PIM indépendants de la plate-forme cible. Le PIM de basereprésente uniquement les capacités fonctionnelles métier et le comportement de l'application.
- **Le PDM (Platform Description Model) :** décrit la plate-forme sur laquelle le système va être exécuté. Actuellement, il est souvent défini informellement sous forme de manuels de logiciels et de matériels. Dans une démarche MDA, on se base sur les PDM pour générer les PSM à partir des PIM. Une telle approche est appelée cycle de développement enY.
- **Le PSM (Platform Specific Model) :** est le modèle produit par la transformation d'un

  PIM pour prendre en compte la plate-forme d'exécution cible choisie. Les PSM servent principalement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ces modèles contiennent les informations nécessaires à l'exploitation d'une plate-forme d'exécution. Ils sont donc essentiellement productifs mais pas forcément pérennes.

4.1.2. Quelques standards de MDA

L'approche MDA est liée à de nombreux standards [5], nous pouvons citer :

- **UML (Unified Modeling Language) :** Un langage visuel semi-formel pour la modélisation des systèmes. Il permet de schématiser l'architecture, les solutions et les vues avec des diagrammes augmentés de texte.

- **MOF (Meta-Object Facility) :** Un standard de méta-modélisation constitué d'un ensemble d'interfaces standards pour définir la syntaxe et la sémantique d'un langage de modélisation, créé principalement pour définir la notation UML.
- **XMI (XML Meta data Interchange) :** Un standard d'échange de métadonnées.
- **OCL (Object Constraint Language) :** Un langage qui, intégré à UML, lui permet de formaliser l'expression des contraintes.
- **SPEM (Software Process Engineering Metamodel) :** Un processus de méta-modélisation défini comme un profil UML.
- **CWM (Common Warehouse Metamodel) :** Une interface basée sur UML, MOF et XMI pour faciliter l'échange de métadonnées entre outils, plateformes et bibliothèques de métadonnées dans un environnement hétérogène.
- **MOFM2T (MOF Model-to-Text language) :** Une spécification utilisée pour exprimer des transformations de modèles en texte.
- **QVT [QVT] :** Un langage standard pour exprimer les transformations de modèles.
- **EDOC (Enterprise Distributed Object Computing) :** Une plateforme standard basée sur UML pour simplifier le développement des composants dans un environnement distribué.

4.1.3. L'architecture de MDA

L'OMG, dans le cadre de ses travaux concernant la méta-modélisation, a défini la notion de méta-méta-modèle ainsi que la standardisation d'une architecture générale décrivant les liens entre modèles, méta-modèles et méta-méta-modèles [5]. Cette architecture est hiérarchisée en quatre niveaux comme le montre la Figure (1.3).

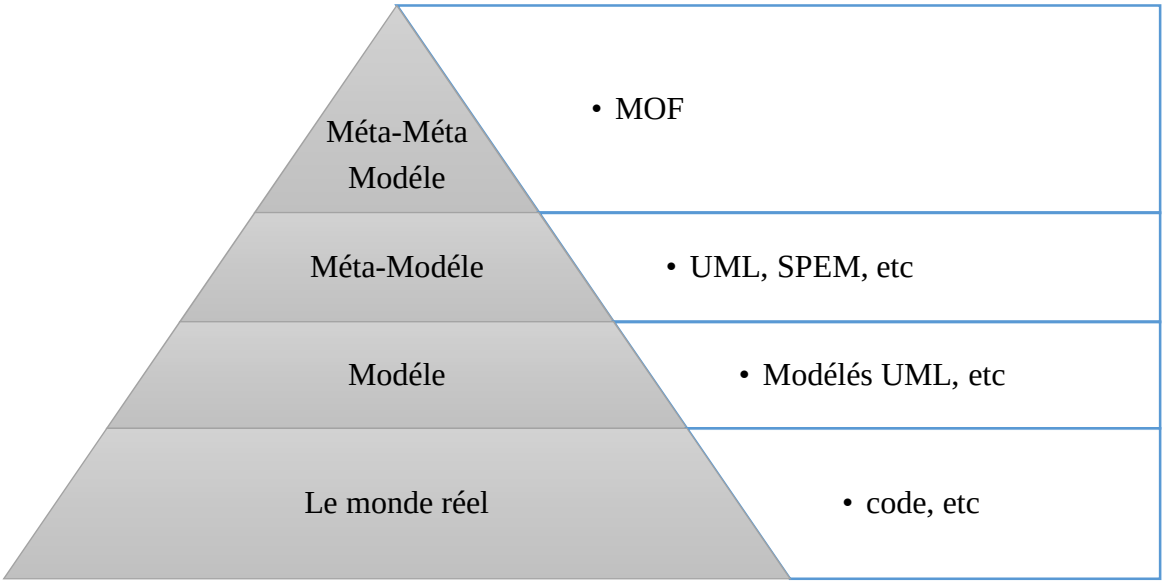

Figure 1.3 : Pyramide de modélisation de l'OMG.

Les quatre niveaux de la pyramide MDA représentent les différents niveaux d'abstraction nécessaires à la modélisation des systèmes. Chaque niveau d'abstraction entretient une relation d'instanciation avec le niveau supérieur.

- **Le niveau M_0 :**représente le monde réelune application qui s'exécute, par exemple. Il contient les informations du monde réel que l'on souhaite modéliser.
- **Le niveau M_1 :**est composé des modèlesLes informations de M_0 sont décrites par un modèle appartenant au niveau M_1. Par exemple, un modèle UML appartient au niveau M_1. Les PIM et les PSM présentés ci-dessus appartiennent à ce niveau. Les modèles du niveau M_1 de même famille sont exprimés dans un langage unique dont la définition est fournie explicitement au niveau M_2.
- **Le niveau M_2 :**représente les langages de définition de modèles Ces langages sont les métamodèles. Par exemple, le métamodèle UML permet de définir des modèles (niveau M1). Le métamodèle SPEM, par exemple, permet de définir des modèles de description de procédés. Les profils UML permettant d'étendre le méta-modèle UML sont aussi considérés comme appartenant au niveau M_2.
- **Le niveau M_3 :**est composé du langage unique de définition des métamodèles, appelé MOF. C'est un métamétamodèle qui définit la structure de tous les métamodèles qui se trouvent au niveau M_2. Le MOF est réflexif, c'est-à-dire qu'il s'auto-décrit, ce qui permet de dire que le niveau M_3 est le dernier niveau de la hiérarchie. Le niveau M_3 correspond donc aux fonctionnalités universelles de modélisation logicielle, alors que le niveau M_2 correspond aux aspects spécifiques des différentes familles, chaque aspect étant pris en compte par un métamodèle spécifique.

4.2. Autres approches basées sur les modèles

Il est à noter que l'approche MDA, bien qu'elle soit la plus importante, elle n'est pas la seule à concentrer son activité sur l'utilisation des modèles. Nous pouvons trouver l'approche CASE (Case Aided Software Engineering), l'approche MIC (Model-Integrated Computing), les Software Factories, etc. qui concentrent également leurs activités sur les modèles [9].

5. Intégration des méthodes formelles à l'IDM

Dans l'ingénierie dirigée par les modèles les méthodes formelles, sont une sorte particulière de technique pour le développement, la vérification et la validation de systèmes et de logicielle.

5.1. Pour quoi les méthodes formelles

Les méthodes formelles permettent de raisonner sur des programmes informatiques afin de démontrer leur conformité, en se basant sur des raisonnements logique mathématique. En pratique, la méthode formelle est basée sur une notation formelle permettant d'atteindre des exigences de qualité élevées du système à modéliser [11].

5.2. Définition

Une méthode formelles permet à offrir un cadre mathématique permettant de d'écrire d'une manière précise et strict les programmes que nous voulons construire. Ce cadre formelle vise d'éliminer les ambiguïtés existant au niveau du cahier des charges et du langage naturel.

5.3. Classification des méthodes formelles

Selon J.M.Wing, il existe différentes classification des méthodes formelles basée sur des critères spécifiques (voir figure 1.4) [16] [22].

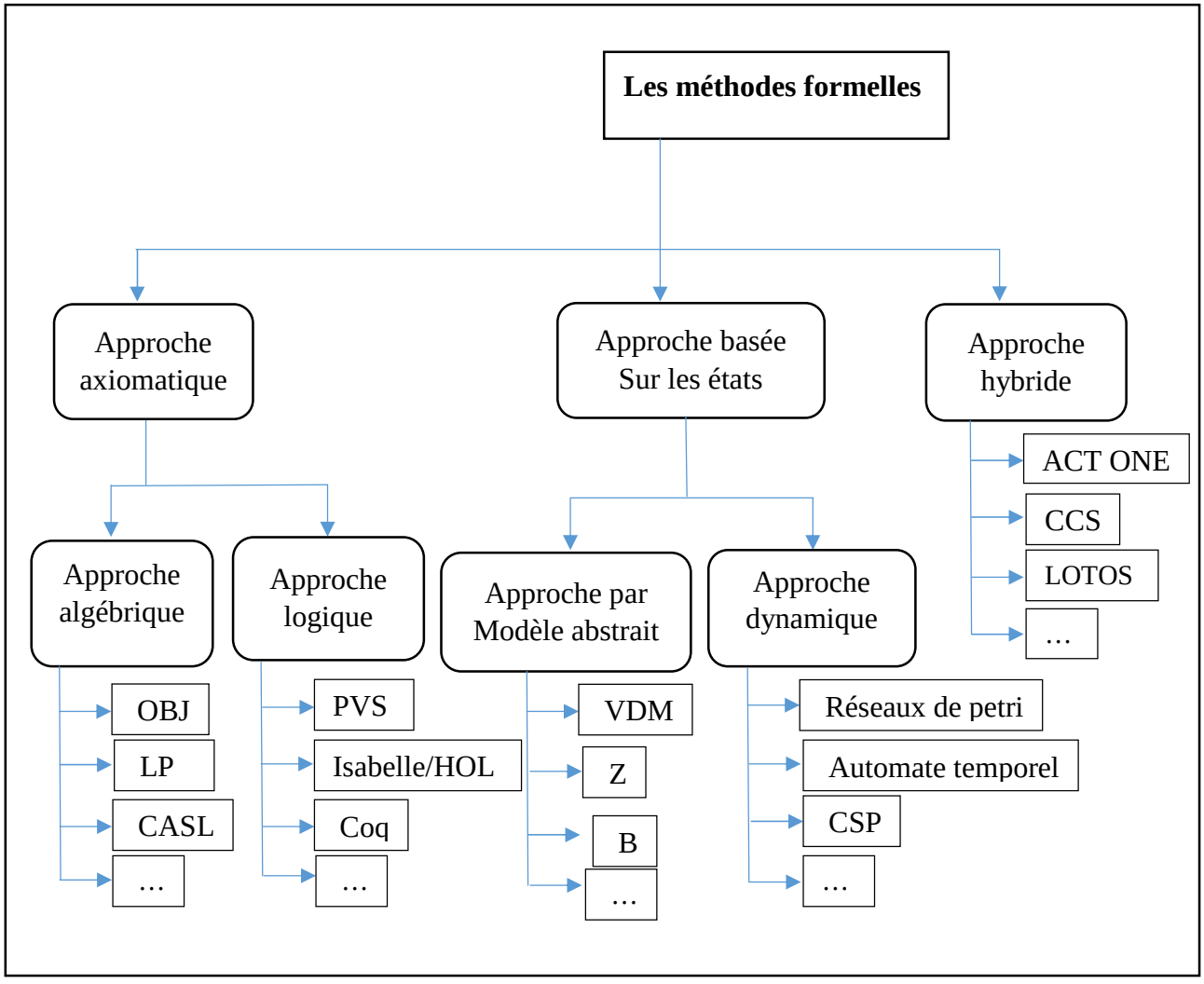

Figure 1.4 : classification des méthodes formelles.

5.3.1. L'approche axiomatique

Cette approche est basée sur la construction d'une axiomatisation qui permet d'obtenir les propriétés comportementales. Axiomatisation présentée par approches algébriques ou logiques.

- **L'approche algébrique :** permet de définir des types abstraits de données en spécifiant pour chaque opération les types de valeurs de ses paramètres et du résultat, en précisant une expression construite à l'aide des variables appelées termes, et en décrivant les propriétés des opérations sous forme d'équivalences entre termes (les axiomes). L'application des axiomes à des termes permet d'obtenir d'autres expressions d'équivalence.
- **L'approche logique :** permet d'exprimer des systèmes transformationnels en utilisant la logique temporelle pour exprimer des propriétés dynamiques de sûreté et vivacité des systèmes réactifs. Dans cette approche on s'intéresse à la démonstration de programmes en appliquant les théories de la démonstration automatique ou semi-automatique de théorèmes.

5.3.2. L'approche basée sur les états

Cette approche construit un modèle en termes de structures mathématiques tout en gardant les propriétés du système. L'approche basée sur les états appliquant des approches dynamiques et des approches ensemblistes.

- **Les approches ensemblistes :** permettent de fournir une syntaxe et une sémantique du modèle abstrait en se basant sur la théorie des ensembles, logique du premier ordre, ou la théorie des types. Les approches ensemblistes différent des approches algébriques par l'utilisation des types abstraits prédéfinis pour modéliser l'état du système à construire. Chaque opération est spécifiée indépendamment en décrivant son effet sur l'état du système.
- **Les approches dynamiques :**se basent sur la notion de processus pour spécifier les systèmes de transitions en utilisant les automates, les réseaux de Petri et les algèbres de processus comme CSP.

5.3.3. L'approche hybride

Combine l'axiomatisation avec le modèle de données ou bien l'inverse.

6. Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur l'ingénierie dirigée par les modèles où l'accent a été mis sur la présentation des concepts de base de l'IDM en général. Et plus spécifiquement la transformation dans le cadre de l'approche IDM. Nous avons aussi décrit le principe de l'approche MDA. Par la suite nous avons cité quelque information sur l'intégration des méthodes formelles. Le chapitre suivant va introduire la notion de l'approche orienté objet et celle de l'orienté aspect.

Chapitre 02

L'approche orientée objet et L'approche orientée aspect

Au Sommaire de ce chapitre

1. Introduction

L'orientée-objet induit une nouvelle culture du développement logiciel, qui s'intéresse aux relations entre le processus de développement et les données. L'orienté objet est un paradigme qui a montré son importance dans la résolution des problèmes complexes, et trouvé des limites pour lesquelles ce paradigme n'a donné aucune solution. Pour Cela les développeurs et les programmeurs ont pensé à définir un nouveau paradigme appelé paradigme orienté aspect permet la résolution des problèmes très complexes, de grandes tailles, et de donner des solutions meilleures.

Dans ce chapitre nous présentons quelque concept de base de l'approche orienté objet. Ensuite, nous décrivons une représentation générale du langage de modélisation objet unifié UML et du langage de spécification OCL. Enfin, nous présentons quelques notions de l'approche orientée aspect.

2. L'approche orienté objet

2.1. Définition

L'orienté objet est un paradigme de conception de systèmes logiciels. Cette approche considère le logiciel comme un ensemble d'objets interagissant via le mécanisme d'envoi de messages. Chaque objet, dispose d'un ensemble d'attributs décrivant son état et d'un ensemble de méthodes caractérisant son comportement [35].

2.2. Concept de base

- **Objet :** Un objet est une instance d'une classe il représente l'état d'une classe a un instant précis [13]. Un objet peut être défini par 3 caractéristiques fondamentales.

  - ✓ **Etat :** c'est la situation instantanée, regroupe les valeurs de tous les attributs d'un objet sachant qu'un attribut est une information qui qualifie l'objet qui le contient.
  - ✓ **L'identité** : est un concept elle ne se représente pas de manière spécifique en modélisation. Chaque objet possède une identité de manière implicite. L'identité permet de le distinguer des autres objets. Chaque objet est nommé, ce nom unique de l'objet est son seul et unique identifiant. On construit aussi une identité grâce à un identifiant découlant naturellement du problème.
  - ✓ **Le comportement :** définit la manière dont l'objet agit ou réagit aux divers messages qui lui parviennent de son environnement. Le comportement

regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet. Chaque atome de comportement est appelé opération.

Une opération est une fonction ou une transformation qui peut être appliquée aux objets.

- **Classe :** Une classe est la description d'un ensemble d'objets qui partagent les mêmes attributs, les mêmes opérations, les mêmes relations et la même sémantique. [13].

  Une classe est caractérisée par trois aspects essentiels :

  - ✓ **L'instanciation :** est la relation qui relie une classe à ses objets : c'est le mécanisme de création d'un nouvel objet, les classes agissant un peu comme des "moules à objets". Un objet est donc une instance de sa classe.
  - ✓ **Les attributs :** Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet.
  - ✓ **Les opérations :** Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet réalise. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures. De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

- **Les Lien et les associations :**
  - ✓ **Lien :** Un lien représente une relation entre des objets. Un lien est une connexion conceptuelle ou physique particulière entre des objets (instances de classes).
  - ✓ **Association :** Une association est une relation entre plusieurs classes, caractérisée par un verbe, décrivant conceptuellement les liens entre les objets de ces classes. L'association représente un ensemble potentiel de liens, comme la classe représente un ensemble potentiel d'objets. peut-être l'association considérée comme une connexion bidirectionnelle entre les classes.

- **Généralisation et Héritage :** La généralisation et l'héritage sont des abstractions puissantes qui permettent de factoriser des points communs entre des classes tout en préservant leurs différences.
  - ✓ **La généralisation** : est une association entre une classe et une ou plusieurs versions affinées de cette classe. En d'autres termes, la généralisation définit une relation de classification entre une classe plus générale ou super-classe, et une ou des classe(s) plus spécifique(s) ou sous-classes.
  - ✓ **L'héritage :** est un mécanisme de transmission des caractéristiques d'une classe vers une sous-classe. Une classe peut être spécialisée en d'autres classes,

afin d'ajouter des caractéristiques spécifiques ou d'en adapter certaines [27]. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.

- **L'agrégation :** est une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe. Une relation d'agrégation permet donc de définir des objets composés d'autres objets. L'agrégation permet donc d'assembler des objets de base, afin de construire des objets plus complexes [26].
- **La composition :** est une agrégation forte. Lorsque l'objet composé est supprimé, l'objet composite l'est aussi. Par exemple, la relation entre une personne et ses coordonnées est une composition car si on supprime une personne ses coordonnées vont disparaître.
- **L'encapsulation :** consiste à masquer les détails d'implémentation d'un objet, en définissant une interface via un ensemble de méthodes publiques [28]. L'interface est la vue externe d'un objet, elle définit les services accessibles aux utilisateurs de l'objet. L'encapsulation facilite l'évolution d'une application car on peut modifier l'implémentation d'un objet sans modifier son interface, et donc la façon dont l'objet est utilisé, est conservée.

  L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

2.3. Langage UML

2.3.1. Définition

UML (Unified Modeling Language) se définit comme un langage de modélisation graphique et textuel destiné à comprendre et décrire des besoins, spécifier et documenter des systèmes, Esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vue [12].

2.3.2. Les diagrammes d'UML

UML dans sa version 2.0 propose treize diagrammes qui peuvent être utilisés dans la description d'un système. Ces diagrammes sont regroupés dans deux grands ensembles [12].

- **Les diagrammes structurels ou statiques d'un système :**
  - ✓ **Les diagrammes d'objet :** Il montre des objets et des liens entre ces objets (les objets sont des instances de classes dans un état particulier).Il montre des objets et des liens entre ces objets (les objets sont des instances de classes dans un état particulier).

- ✓ **Les diagrammes de composants :** Il montre les composants du système d'un point de vue physique, tels qu'ils sont mis en œuvre (fichiers, bibliothèques, bases de données...). Il montre la mise en œuvre physique des modèles de la vue logique avec l'environnement de développement.
- ✓ **Les diagrammes de déploiements :** Ce type de diagramme UML montre la disposition physique des matériels qui composent le système, et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de nœuds, connectés par un support de communication.
- ✓ **Les diagrammes de paquetages :** Un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML, il sert à représenter les dépendances entre paquetages.
- ✓ **Les diagrammes de structures composites :** Le diagramme de structure composite permet de décrire sous forme de boîte blanche les relations entre les composants d'une seule classe.
- ✓ **Les diagrammes de classes :** Le but d'un diagramme de classes est d'exprimer de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes. Une classe a des attributs, des opérations et des relations avec d'autres classes.

➢ **Les diagrammes comportementaux ou dynamiques d'un système**

- ✓ **Les diagrammes de séquence :** Il représente séquentiellement le déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs. Le diagramme de séquence peut servir à illustrer un cas d'utilisation.
- ✓ **Les diagrammes d'états transitions :** Permet de décrire sous forme de machine à états finis le comportement du système ou de ses composants. Il est composé d'un ensemble d'états, reliés par des arcs orientés qui décrivent les transitions.
- ✓ **Les Diagrammes de temps :** Le diagramme de temps permet de décrire les variations d'une donnée au cours du temps.
- ✓ **Les diagrammes de cas d'utilisation :** Le diagramme des cas d'utilisation, permet 'identifier les possibilités d'interaction entre le système et les acteurs. Il permet de clarifier, filtrer et organiser les besoins.
- ✓ **Les diagrammes vus d'ensemble des interactions :** Les diagrammes vus d'ensemble des interactions mettent l'accent sur la vue d'ensemble du flux de contrôle des interactions. Il s'agit d'une variante du diagramme d'activités où

les nœuds sont les interactions ou d'événements d'interaction. Ils décrivent les interactions où les messages filières sont cachés.

- ✓ **Les diagrammes d'activités :** Un diagramme d'activité est une variante des diagrammes d'états-transitions. Il permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.
- ✓ **Les diagrammes de communication (collaborations)** Permettent de décrire sous forme de machine à états finis le comportement du système ou de ses composants. Il est composé d'un ensemble d'états, reliés par des arcs orientés qui décrivent les transitions.
    - **Exemple pour les diagrammes de communication [6]**

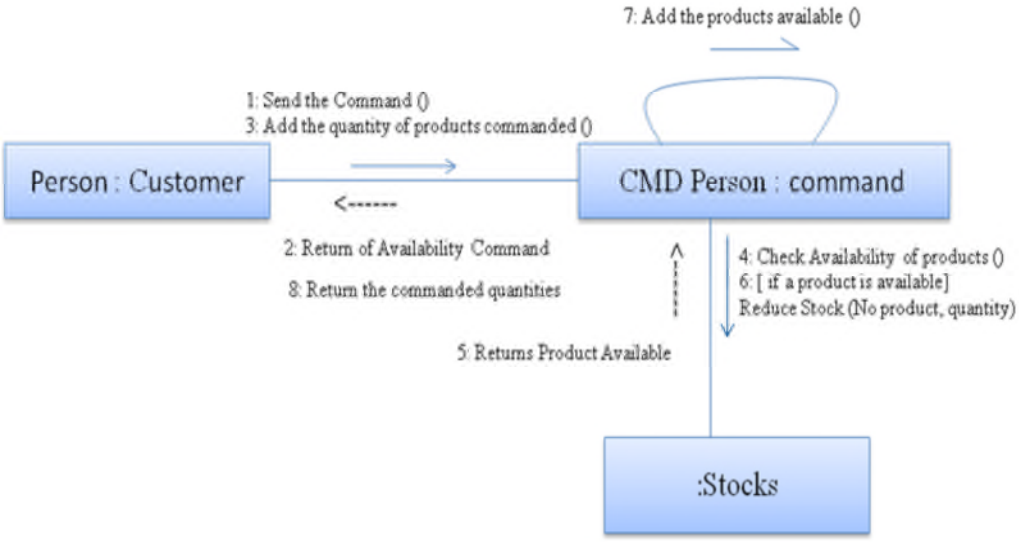

Figure 2.1 : un exemple simple d'un diagramme de communication.

- **Diagramme de communication liens :** Un lien est une connexion entre deux objets, qui indique qu'une forme de navigation et de visibilité entre eux est possible, d'autre part un lien permet d'acheminer des messages dans un sens ou dans l'autre. Plusieurs messages et ce dans les deux sens, peuvent circuler sur le même lien [6].
- **Diagramme de communication message :** Chaque message entre objets est représenté par une expression, une flèche indiquant sa direction, et un numéro indiquant sa place dans la séquence [6].

2.4. Langage OCL

2.4.1. Définition

OCL (Object Contraint Langage) est le langage de spécification des contraintes d'UML. OCL possède une syntaxe concrète qui permet d'exprimer des expressions et des contraintes. Les expressions OCL sont employées dans les digrammes UML [23].

2.4.2. Pour quoi utiliser langage OCL

Le langage OCL est utilisé avec des objectifs différents [31] :

- 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 prés et post conditions sur des opérations et des méthodes.
- Comme langage de navigation dans un diagramme.
- Pour spécifier des contraintes sur des opérations.

2.4.3. Les contraints OCL

Il y a deux types des contraintes : les invariants et les prés / post-conditions [24].

- **Un invariant :** c'est une contrainte qui définit sur une classe. Une expression OCL peut être un élément d'un invariant qui est une contrainte stéréotypée avec «invariant». Quand l'invariant est associé à un classificateur, celui-ci est référencé en tant que type. L'expression est alors un invariant du type et doit être vrai pour les instances ce type à n'importe quel moment.
- **Un pré / post-condition : c'**est une contrainte qui définit un comportement d'une méthode doit satisfaire. Une expression OCL peut être un élément d'une pré-condition ou d'une post-condition correspondant aux stéréotypes de contraintes associés à une opération ou à une méthode. Alors, l'instance de contexte self est une instance du type qui possède l'opération ou la méthode comme caractéristique. En OCL, la déclaration de contexte utilise le mot clé contexte, suivi du type et de la déclaration d'opération. Les étiquettes pré et post déclarent les contraintes comme étant respectivement une contrainte de pré condition ou une contrainte de post condition.
  - ✓ **La pré-condition :** d´écrits la condition qui doit être respectée avant l'exécution de la méthode.
  - ✓ **La post-condition :** d´écrit l'effet produit par l'exécution de la méthode.

2.4.4. Les types de langage OCL

En OCL, le nombre de types élémentaires est prédéfini et disponible à tout moment pour le modélisateur. Ces types de valeurs prédéfinis sont indépendants de tout modèle objet et font partie de la définition d'OCL. La valeur la plus élémentaire en OCL est la valeur d'un des types de base [24].

- **Les types de base :** Integre, Real, Booléen et String.
- **Les autres types :**
  - ✓ **Type Tuple :** enregistrement.
  - ✓ **OCL Message Type :** utilisé pour accéder aux messages d'une opération ou d'un signal, offre un rapport sur la possibilité d'envoyer/recevoir une opération/un signal.

2.5. Les avantage de l'approche orientée objet :

- L'approche orientée objet est largement adoptée, tout simplement parce qu'elle a montré son efficacité lors de la construction de systèmes dans une diversité de domaines métier et qu'elle englobe les dimensions et les degrés de complexité.
- La stabilité de la modélisation par rapport aux entités du monde réel.
- La simplicité du modèle qui fait appel à cinq concepts fondateurs (les objets, les messages, les classes, la généralisation et le polymorphisme).
- La construction itérative facilitée par le couplage faible entre composants et la possibilité de réutiliser des éléments d'un développement à un autre.

2.6. Les inconvénients de l'approche orientée objet :

- L'objet ne peut pas faire la suppression.
- Il n'existe pas un enchainement et un éparpillement du modèle.
- Le modèle de résolution est difficile à lire et à comprendre, en plus il peut provoquer des erreurs.
- La dispersion (duplication), les fonctionnalités transversales se transverses dans le modèle d'application.
- La réutilisation des modèles est complexe.

3. L'approche orienté aspect

3.1. Définition

L'approche orienté aspect est un paradigme permet la résolution des problèmes très complexes, de grande tailles, et de donner des solutions meilleures. Cette approche considère des concepts, des langages et des outils sont introduits avec la modélisation orientée aspect.

3.2. Concepts et terminologie de la Modélisation Orientée Aspect

Toute nouvelle approche de modélisation nécessite l'introduction de nouveaux concepts, de nouveaux langages et de nouveaux outils. De nouveaux concepts sont introduits avec la MOA afin de permettre aux développeurs de spécifier et de modéliser les préoccupations transverses [6]. Ces concepts sont :

- **Préoccupations :** une préoccupation est définie comme un intérêt relatif au développement d'un système, ayant un rôle critique ou important à un moment donné du développement. Une préoccupation est alors soit une préoccupation non transversale qui peut être liée à sa fonctionnalité appelée préoccupation de base, soit une préoccupation transversale, qui peut être liée à des exigences non fonctionnelles appelées préoccupation d'aspect ou simplement aspect.
- ✓ **Préoccupation de Base :** Une base est une unité de modularisation qui peut être représentée dans un module de manière isolée en respectant une décomposition dominante. Elle représente donc une préoccupation non transversale (fonctionnelle) qui peut être capturée de manière efficace avec des approches traditionnelles telles que l'approche orientée Objet.
- ✓ **Préoccupation d'aspect :** Un aspect est une unité de modularisation qui est entremêlée avec les autres préoccupations, qu'elles soient de base ou d'aspect. Elle représente donc une préoccupation transversale (non fonctionnelle) qui peut être capturée de manière efficace avec l'approche orientée Aspect.
- **Point de jonction (joinpoint en anglais (PJ)) :** Endroit dans le modèle où les conseils (advice) devraient être insérés.
- **Point de Coupe (pointcut en anglais (PC)) :** Un ensemble de points de jonction, souvent une expression régulière et signifie le choix des points de jonction pour l'application de conseils.
- **Conseil (advice en anglais) :** Une partie du modèle qui contient la totalité ou une partie de l'aspect inséré avant ou après ou autour d'un point de jonction. Il implante une préoccupation transverse.

- **Aspect (Aspect en anglais) :** Module définissant des Conseils et leurs points d'activation.
- **Tisseur (weaver en anglais) :** Est un outil spécial permettant d'intégrer ou de composer les aspects au modèle de base pour obtenir un modèle final (base+aspect).

3.3. Les techniques et les technologies Orientées Aspect

3.3.1. Ingénierie des exigences orientée aspect

Les techniques d'ingénierie des exigences qui reconnaissent explicitement l'importance d'identifier clairement et traiter des préoccupations transversales d'une manière précoces sont appelées les approches d'ingénierie des exigences orientées aspect. Les préoccupations transverses peuvent être des exigences non fonctionnelles aussi bien que des exigences fonctionnelles [29].

3.3.2. Les approches d'architecture orientée aspect

Un langage de conception orienté aspect consiste en quelque sorte à préciser certains aspects comme un moyen de spécifier la manière dont les aspects doivent être composés et un ensemble de sémantique de composition bien définis pour décrire les détails de la façon dont les aspects doivent être intégrés dans le paradigme orienté objet [29].

3.3.3. Les approches de conception orientées aspect

La conception orientée aspect porte sur la représentation explicite des préoccupations transversales à l'aide de langages de conception adéquats. Un langage de conception orienté aspect consiste en quelque sorte à préciser certains aspects comme un moyen de spécifier la manière dont les aspects doivent être composés et un ensemble de sémantique de composition bien définis pour décrire les détails de la façon dont les aspects doivent être intégrés dans le paradigme orienté objet [29][32].

3.3.4. La programmation Orientée Aspect

L'orientée aspect se manifeste au niveau de la programmation que les langages de programmation orientés aspect. La plupart de ces langages orientés aspect sont des langages (orientés objet) existants étendus avec des fonctionnalités orientées aspect pour représenter les aspects, exprimer les points de coupure, les point de jonction.

3.3.5. Les approches de Vérification et de test des programmes orientés aspect

L'approche orientée aspect a posé de nouveaux défis dans les techniques de vérification et validation des logiciels afin de s'assurer que la fonctionnalité désirée est satisfaite par le système. Des aspects peuvent potentiellement endommager la fiabilité d'un système pour

lequel ils sont tissés, et peuvent rendre invalides des propriétés essentielles du système qui étaient correctes avant le tissage d'aspect. Pour assurer la validité du logiciel par aspects, il y a beaucoup de recherche sur l'utilisation de méthodes formelles et techniques de tests spécialement adaptées aux aspects [29].

3.3.6. L'intergiciel orienté aspect

L'intergiciel n'est pas une étape dans le cycle de vie, il est un domaine important et vaste pour les idées orientées aspect. Beaucoup de développeurs de logiciels ont adopté des approches d'intergiciel pour aider à la construction de systèmes distribués à grande échelle. L'intergiciel facilite le développement de systèmes logiciels distribués en supportant l'hétérogénéité, cachant les détails de distribution et fournissant un ensemble de services spécifiques pour un domaine commun [32].

3.4. Fondements de la Modélisation Orientée aspect

3.4.1. Définition de la Modélisation Orientée Aspect

Pour résoudre les problèmes de la modélisation orientée objet, il serait intéressant de modulariser l'implantation des modèles transverses (technique, non fonctionnel) indépendamment les uns des autres. On parle alors de séparation de ces modèles (fonctionnels, non fonctionnels) et de les combiner ultérieurement pour produire le modèle final. La Modélisation Orientée Aspect (MOA) est une approche (parmi d'autres) permettant d'atteindre ce but. Autrement dit le processus de tissage (la composition) d'aspects se divise en deux phases. Tout d'abord, une phase de détection permettant d'identifier des parties particulières d'un modèle de base, puis une phase de composition permettant de construire le modèle prenant en compte l'aspect [6]. En général, le cycle de développement de la MOA se fait en trois étapes :

- **La décomposition aspectuelle :** Elle consiste à décomposer les besoins afin d'identifier et séparer les modèles transverses encapsulés dans des modules aspects (techniques, non fonctionnels) et métiers (de base, fonctionnels) qui peuvent être modularisés dans des modules de base tel que classe.
- **Implantation des modèles :** Elle consiste à implanter chaque modèle séparément. Les modèles métiers sont implantés par les techniques conventionnelles de la modélisation orientée objet alors que les modèles transverses sont implantés par les techniques de la modélisation orientée aspect.
- **Recomposition aspectuelle :** Elle consiste à construire le système final en intégrant ou recoupant les modèles métiers avec les modèles transverses. Cette phase est

appelée tissage (weaving en anglais). Un tisseur (weaver) utilise des règles spécifiées par le concepteur de l'application afin de recouper correctement les modèles entre eux. Le tissage d'aspects est une opération qui accepte en entrée les modules de base et les modules d'aspect. Leur but est l'application et l'attachement des aspects dans les modules de base selon les points de jonction correspondant à la spécification des points de coupure d'aspect.

Donc un aspect = les points de coupure + Conseil.

Les points de coupure= ∑ les points de Jonction.

On peut distinguer deux types de tissage : le tissage statique (avant l'exécution) et le tissage dynamique (durant l'exécution)

Dans la figure (2.2), nous présentons le processus de la modélisation orientée aspect tels que : S1, S2 et S3 sont des états, vérification et sécurité sont des aspects.

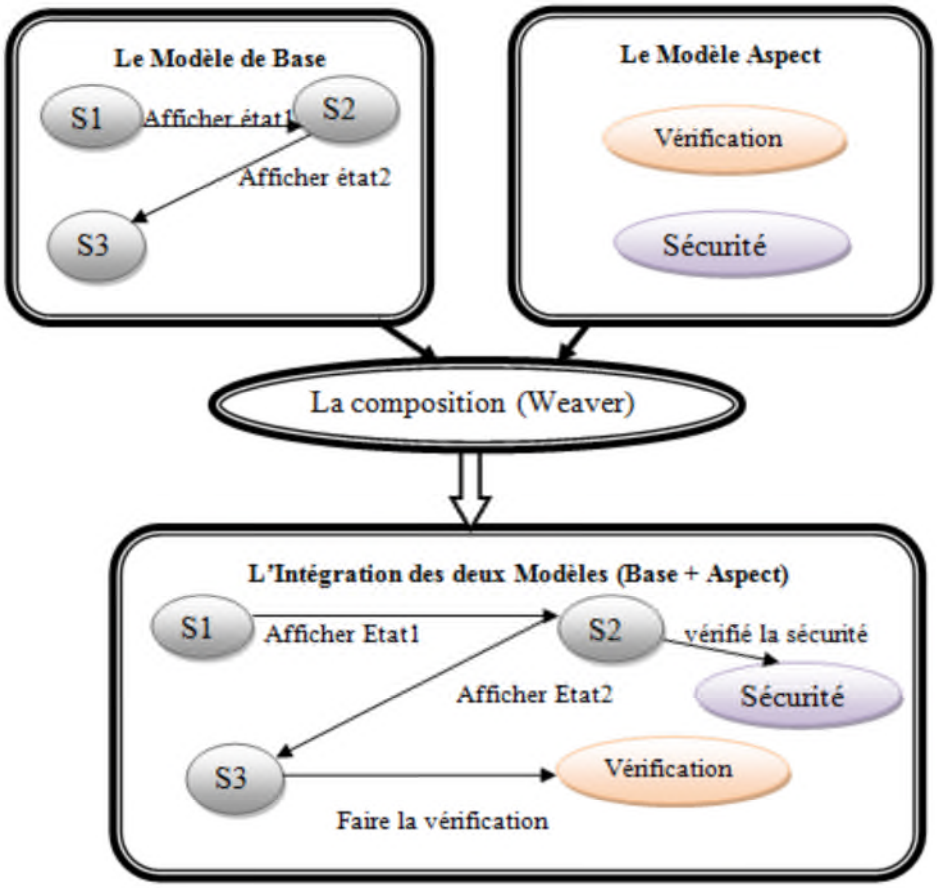

Figure 2.2 : le processus de la modélisation orientée aspect.

3.5. Les inconvénients de l'approche orientée aspect

- La gestion des transactions, par la coupe transversale, est difficile à factoriser dehors dans un aspect distinct.
- Les différents aspects peuvent effectivement nuire même aux points de jonction dans le tissage. Ainsi, la MOA peut violer le principe d'encapsulation, bien que d'une manière assez systématique et bien contrôlée.
- La MOA est surtout adaptée pour les projets de développement de logiciels à grande échelle.

4. Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur l'approche orienté objet. Ensuite, nous avons présenté le langage de modélisation unifié UML 2.0 avec ces diagrammes, et le langage de spécification OCl. Par la suite, nous avons aussi décrit le principe de l'approche orientée aspect. Le chapitre suivant va introduire les vues des transformations ainsi que les techniques et les propriétés de la vérification formelles. Puis nous décrivons le modèle de transformation et la présentation de l'approche USE. Enfin représente les concepts de base de l'approche USE.

Chapitre 03

Vérification des transformations de modèles

Au sommaire de ce chapitre

1. Introduction

En Ingénierie dirigée par les modèles (IDM), le développent de logiciel est basé sur la définition de modèles qui décrivent différentes vus du système à construire ainsi que les transformations des modèles qui offrent un processus de développement (semi)automatique. La vérification de modèles et transformation de modèles est cruciale afin d'améliorer la qualité du produit final.

Dans ce chapitre, nous décrivons les deux vues des transformations existent dans la littérature. Par la suite, nous présentons les propriétés importantes pour la vérification de ces transformations ainsi que les techniques utilisées. Enfin, nous présentons les concepts de base de l'approche de vérification USE.

2. Vues des transformations

En ingénierie dirigée par les modelés une transformation possédante deux natures, elle peut être vu comme une transformation de modèles, comme elle peut être vu comme un modèle de transformation [14] :

2.1. Vue opérationnelle

Dans cette perspective, on parle de transformation de modèles. Dans la littérature, de nombreux travaux traitent les transformations de modèles d'une manière opérationnelle (exécutable). Mais, Cette description exécutable est nécessaire d'un point de vus implémentation [15].

2.2. Vue conceptuelle

D'un point de vus modélisation ou conception, une transformation peuvent être vu comme un modèle descriptif [14]. SelonBézivin J: "*Model transformation can be abstracted to a transformation model"*.Dans cette perspective, on parle de modèles de transformations.

3. Transformation de modèles

Moussa Amrani et al, ont proposé une approche tridimensionnelle pour la vérification formelle de transformations de modèles [15] : la transformation étudiée, les propriétés intéressantes ou adressées, et les techniques de vérification formelle utilisée pour établir les propriétés.

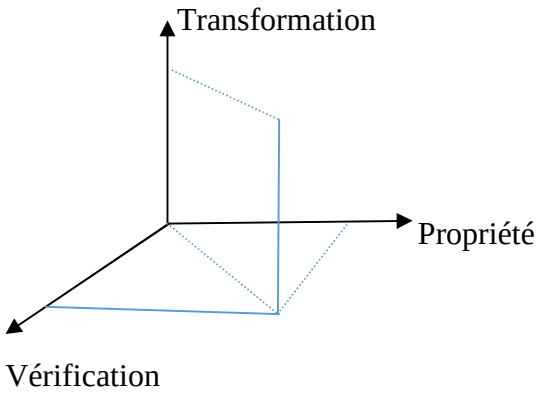

Figure 3.1 : L'approche tridimensionnelle.

3.1. Propriétés

3.1.1. Propriétés dépendantes du langage (Language-RelatedProperties)

D'une perspective opérationnelle, la spécification d'une transformation est conforme à un langage de transformation, dont il pose des propriétés soit au moment de l'exécution, soit au moment de conception (la relation entre la spécification de la transformation et son langage) voir Figure (3.1) [15].

- **Au moment de l'exécution :**
  - ✓ **Terminaison :** qui garante l'existence d'un modèle(s) cible. C.à.d., l'exécution de la transformation se termine pour toute spécification d'une transformation.
  - ✓ **Déterminisme (confluence) :**qui assure l'unicité du modèle cible correspondu un modèle source pour une spécification d'une transformation.
- **Au moment de conception :**
  - ✓ **Typing :** assure la bonne formation de la spécification d'une transformation.

3.1.2. Propriétés dépendantes de la transformation (Transformation-relatedProperty)

- **Propriétés sur les modèles et leurs meta-modèles :**
  - ✓ **Conformance& Model Typing :** la conformité assure que le modèle est valide par rapport à son Meta-modèle. Actuellement, la conformité est vérifiée automatiquement par le Framework de modélisation [9].
  - ✓ **N-Ary Transformations Properties :** dans le cas où une transformation opère sur plusieurs modèles au même temps (composition de modèles, émerge de modèles, …).

- **Propriété relatives aux modèles d'entrées et sorties :**
  - ✓ **Propriétés syntaxiques (SyntacticProperties) :** il s'agit d'un type de propriétés permettant de relier les éléments des metamodèles source et cible, on essaye d'assurer que certains éléments ou structures de n'importe quel modèle source seront transformé en d'autre éléments ou structures relatives au modèle cible. Cela est reconnue comme la préservation des invariants de la transformation ou la correspondance structurale.
  - ✓ **Propriétés sémantiques (SemanticProperties) :**au-delà de ces relations structurales entre les modèles sources et cibles, il y a des propriétés sémantiques qui doivent être préservés. Cela est reconnu comme la correction sémantique ou la consistance dynamique. Généralement, ce type de propriétés est très difficile à définir et à les prouver.

3.2. Techniques

Plusieurs techniques de vérification formelles ou semi formelles ont été utilisées pour la transformation de modèles dans la littérature, les plus distinguées sont :

3.2.1. Test

C'est une technique classique de vérification et de validation par l'exécution d'une partie ou de la totalité du code implémenté. Le but du test est de détecter les erreurs du système le plus tôt possible. L'inconvénient majeur de cette technique est l'absence de raisonnement mathématique sur les propriétés de spécification, car cette technique repose sur le langage naturel ou sur des formalismes graphiques avec une sémantique un peu précise [4].

3.2.2. Vérification de modèles (Model checking)

Le Model checking est une technique de vérification automatique des systèmes dynamiques. Il s'agit de construire un modèle du système à vérifier avec un formalisme donnée, généralement représenté par un automate à états fini. Ce modèle est vérifié s'il satisfait un ensemble de propriétés (une spécification) souvent formulées en logique temporelle. En autres termes et de point de vue logique, le système est décrit par un modèle sémantique, et les propriétés sont décrites par des formules logiques [13].

3.2.3. Preuve de théorèmes (Theorem proving)

La preuve de théorèmes est une technique où le système et les propriétés recherchées sont exprimés comme des formules dans une logique mathématique. Cette logique est décrite par un système formel qui définit un ensemble d'axiomes et de règles de déduction. La preuve de théorèmes est le processus de recherche de la preuve d'une propriété à partir des axiomes du système. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été éventuellement dérivés. Au contraire du Model checking, le

Theorem proving peut s'utiliser avec des espaces d'états infinis à l'aide de techniques comme l'induction structurelle. Son principal inconvénient est que le processus de vérification est normalement lent, sujet à l'erreur, demande beaucoup de travail et des utilisateurs très spécialisés avec beaucoup d'expertise [2].

4. Modèle de transformation

Martin Gogola et al, ont proposé une approche pour la vérification des propriétés de modèles de transformation. Cette approche propose une analyse de certaines propriétés par l'utilisation de l'outil validateur de modèles.

4.1. Propriétés

La figure (3.2) présente les différentes propriétés qui peuvent être analysées et vérifiées par le validateur de modèles [17].

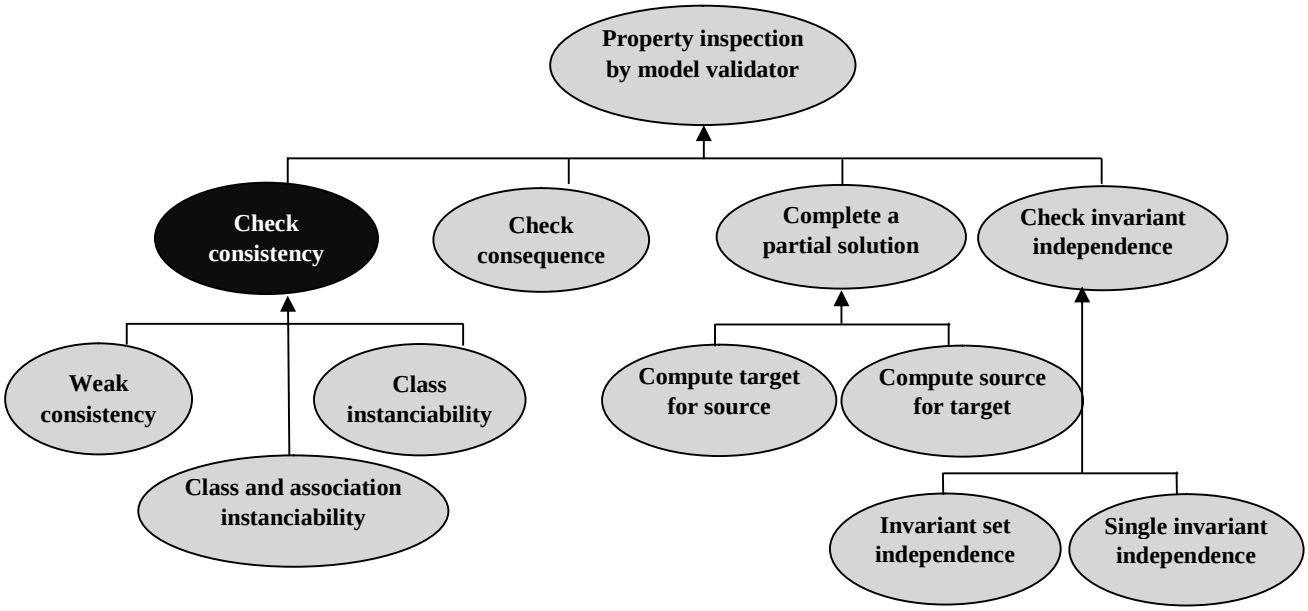

Figure 3.2 : Propriétés vérifiables par le validateur de modèles.

4.1.1. Consistency

La consistance est une propriété qui peut être vérifiée par l'analyse des modèles d'objets, c.à.d. l'instanciation de modèle de transformation. Elle permet de voir si les contraintes sont contradictoires ou non [17].

- **Weak consistency :** signifie qu'au moins un diagramme d'objet valide peut être trouvé, même pour une seule class instanciées avec une populationnon vide [17].
- **Class instanciability:** signifié que toutes les classes sont instanciées avec une population non vide [17].

- **Class and association instanciability:** signifié que toutes les classes et toutes les associations sont instanciées avec une population non vide[17]**.**

4.1.2. Conséquences

La conséquence du modèle peut être inspectée, ainsi que les contraintes indiquées sont remis au modèle validateur et un contre est recherché dans l'espace de recherche finie déterminée par la configuration, la conséquence supposée est considéré comme valide [18].

4.1.3. Partial solution complétion

Une solution partielle du modèle peut être donnée au validateur modèle qui tente de compléter la solution partielle. En termes de notre exemple, un modèle de transformation, ce qui peut être utilisé pour calculer explicitement une instance source, une instance cible et vice ver ses propriétés de transformation comme injectivité peuvent être contrôlés de cette façon [18].

4.1.4. Invariant Independence

Un ensemble des invariants peut être testé pour l'indépendance, c.à.d. elle permet de vérifier si un invariant est non impliqué par autre invariants. En d'autre terme, l'invariant en question est indépendant a tout autre [18].

4.2. Techniques

Plusieurs outils ou techniques pour la vérification de modèles de transformation existent dans la littérature, nous pouvons distinguer : Dresden OCL compiler, Kent Modeling Framework (KMF), UML based Spécification Environnement (USE). Dans notre travail, Nous allons utiliser l'outil USE : UML based Spécification Environnement.

5. Approche USE

5.1. Présentation

USE est un outil de spécification des systèmes. Il permettant de définir des diagrammes de classes UML, des diagrammes d'instance conformes à ces diagrammes de classe et de vérifier des contraintes OCL sur ces diagrammes d'instances [21].

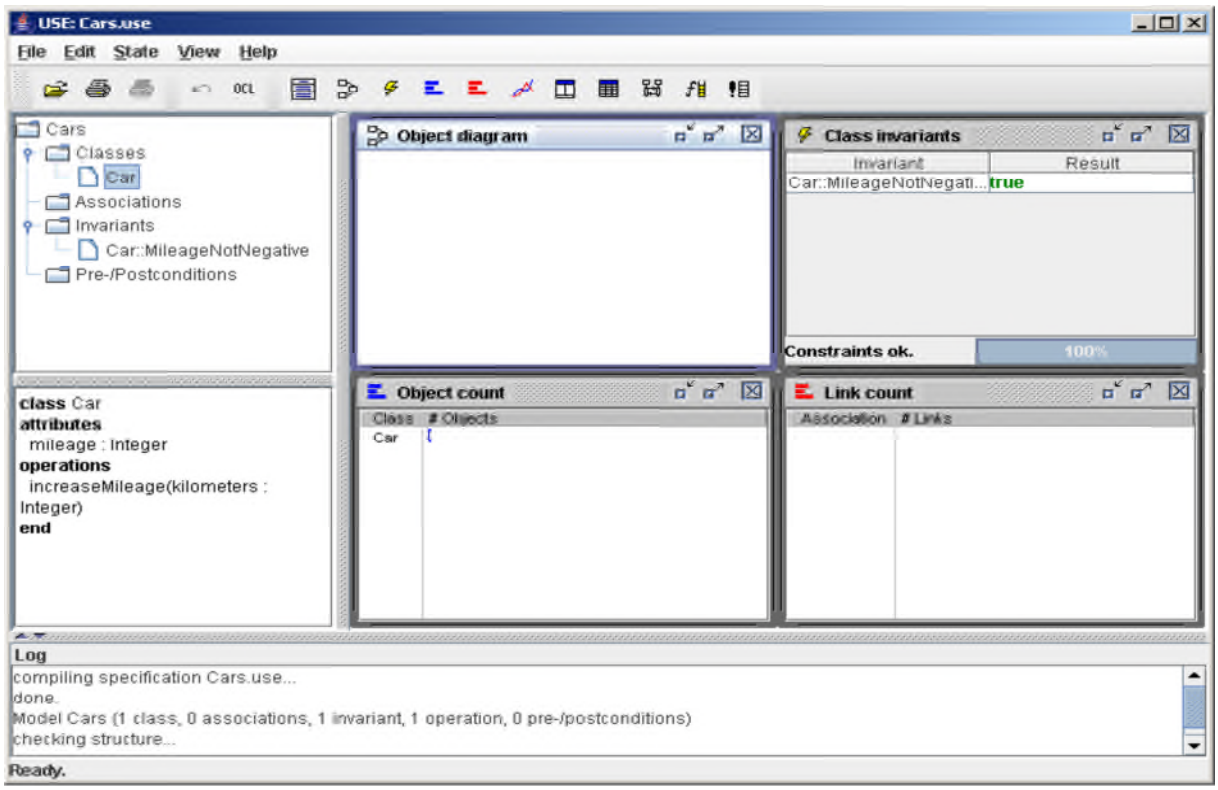

Figure 3.3 : Exemple représente la spécification d'un système dans l'outil USE.

5.2. Concept de bases

5.2.1. Diagramme de classe

Le modèle UML à utiliser les données dans une forme textuelle, sont présentés dans le diagramme de classe. Un diagramme de classe est identifié par : une classe avec des attributs et des opérations, et une association avec des noms et des multiplicités [21].

5.2.2. Classe invariant

Les invariants de classe montre leur évaluation dans l'état actuel du système. Un invariant peut être évaluée à vrai, faux, ou peut-être pas applicable on n'a pas à se soucier d'une véritable invariant. Un faux invariant indique un état de système non valide [21].

5.3.3. Diagramme d'objet

L'état d'un système complet peut être capturé dans un diagramme d'objets. Le diagramme d'objet présentant des objets avec des valeurs d'attributs et des liens d'objet [21].

5.3.4. Diagramme de séquence

Le diagramme de séquence utilisé pour l'exécution des opérations. Les paramètres de fonctionnement et les opérations de séquence est utiliser par les commandes USE [21].

5.3.5. Class extent

La classe extent a été exécutée indique les identités de l'objet et les valeurs des attributs prennent actuellement. La mesure de la classe détermine un schéma objet UML caractérisant l'état actuel du système. L'étendue de la classe peut éventuellement être représenté avec invariants évalués séparément pour chaque objet [21].

5.3.6. OCL expression evaluation

Utilisation de cette expression permet d'interroger l'état actuel du système en évaluant OCL [21].

6. Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur les vue de transformation. Et plus spécifiquement les propriétés et les techniques pour la vérification des transformations de modèles. Nous avons aussi décrit les propriétés et les techniques de modèle de transformation. Par la suite nous avons cité la présentation de l'approche USE. Le chapitre suivant va introduire l'approche étudiée ainsi que le modèle de transformation proposé et enfin la vérification de ce modèle de transformation.

Chapitre 04

Vérification d'une approche de transformation de modèles

Au sommaire de ce chapitre

1. Introduction

La présentation de notre modèle consiste à assurer le passage des diagrammes de communication orienté objet vers des diagrammes de communication orientée aspect. L'objectif de ce chapitre consiste à proposer un modèle représentant ce passage et d'appliquer la technique de vérification USE a ce modèle de transformation.

Dans ce chapitre, nous présentons une vue générale sur l'approche étudiée. Ensuite, nous décrivons le modèle de transformation proposé ainsi que ses détails. Par la suite nous vérifions certaines propriétés. Enfin, nous décrivons un exemple d'étude de cas.

2. L'approche étudiée

Pour transformer les diagrammes de communication orientée objet à des diagrammes de communication orientée aspect, nous proposons deux méta-modèles. La première pour le schéma de communication et le second pour le modèle aspect. Ces méta-modèles sont représentés par le formalisme de diagramme de classes UML. Nous proposons également un ensemble de règles est notre grammaire graphique [34].

2.1. Le Meta-modèle de diagramme de communication

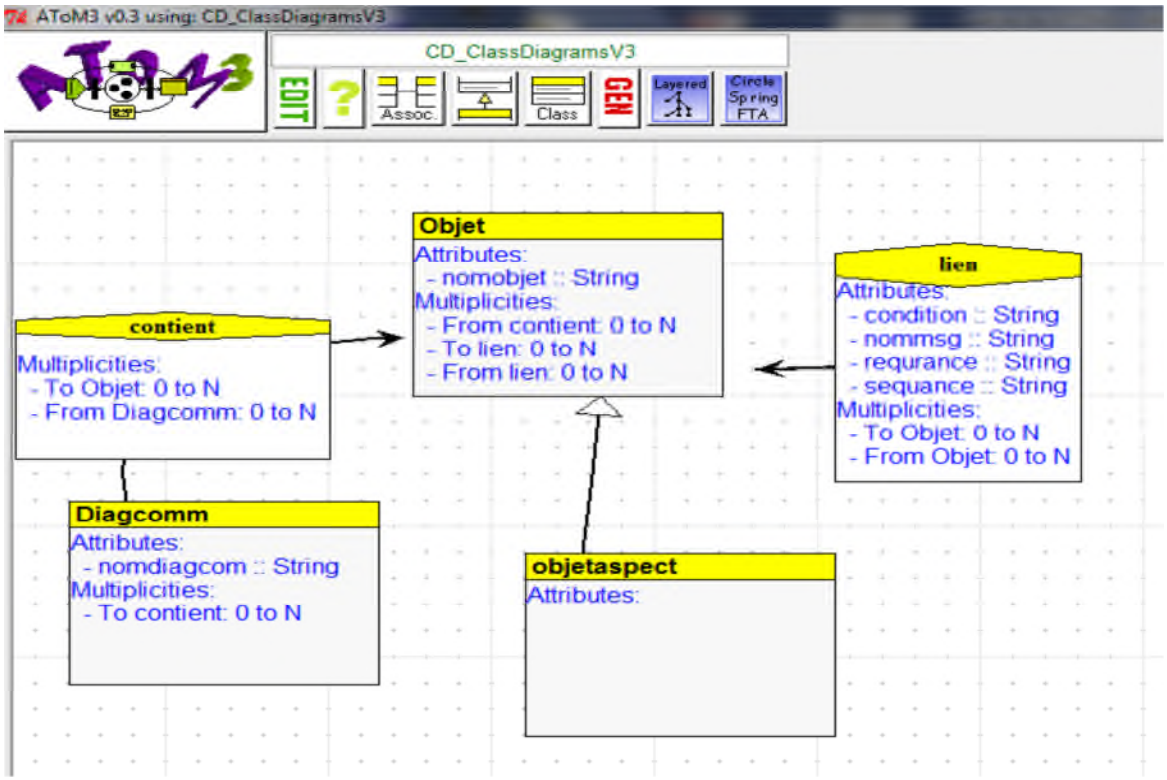

Figure 4.1 : Le Meta-modèle de diagramme de communication.

- **Diagcomm :** Cette classe est utilisée pour représenter le schéma de communication.
- **Objet :** Cette classe représente les objets. Chaque objet a un nom et peut communiquer via des connecteurs.
- **Objetaspect :** Cette classe représente les objets supplémentaires (aspects) de le faire. Il hérite de tous ses attributs, multiplicités, des associations de la classe d'objet.
- **Contient :** Est une association de composition. Cela connecte le schéma avec ses objets.
- **Lien :** Est une simple association. Qui permet la communication entre les deux objets. En d'autres termes, il représente les messages envoyés ou reçus par un objet.

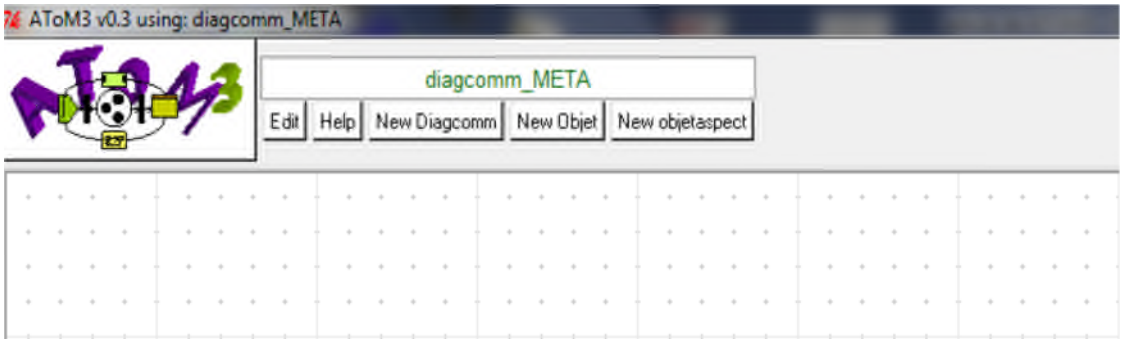

Figure 4.2 : Un outil pour manipuler des schémas de communication.

2.2. Le Meta-modèle de modèle aspect

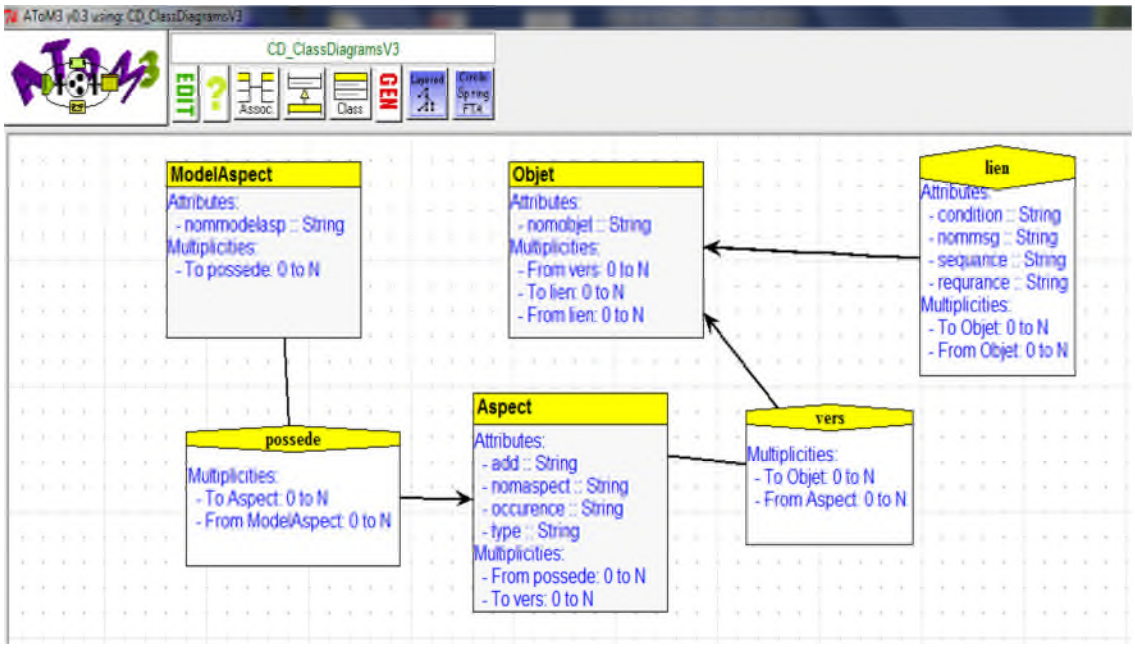

Figure 4.3 : Le méta-modèle d'Aspect Modèle.

- **modelAspect :** Cette classe comprend les aspects avec leurs points d'activation que les objets ou les liens entre deux objets.
- **Aspectj** : Cette classe représente aspects conformément à la syntaxe donnée. Chaque aspect a un nom, le point d'activation et les conseils pour l'insérer.
- **Objet :** Cette classe représente les objets. Chaque objet a un nom et peut communiquer via des connecteurs.
- **Possède :** Est une association de composition, qui relie le model Aspect avec ses aspects.
- **Vers :** Est une association simple, relie les aspects avec les objets sur lesquels un aspect doit être inséré.
- **Lien :** Est une simple association, qui permet la communication entre les deux objets. En d'autres termes, il représente les messages envoyés ou reçus par un objet.

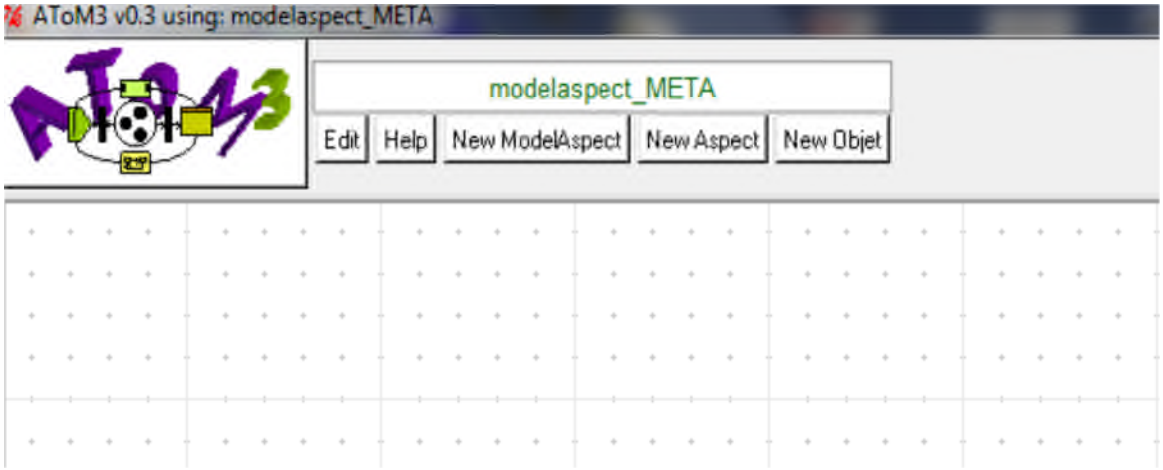

Figure 4.4 : un outil pour manipuler le modèle d'aspect.

2.3. La grammaire proposée

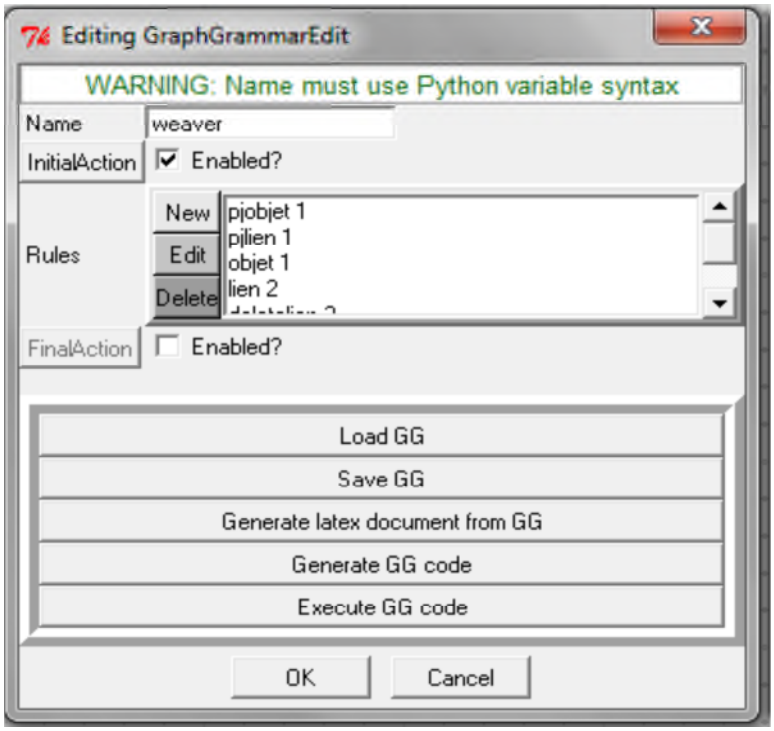

Figure 4.5 : La grammaire graphique.

2.3.1. Les règles appliquées dans la grammaire

- **Règle 1,2 (priorité 1,2) :** Ces règles sont appliquées pour localiser un aspect non préalablement traités (Visité ==0), et créer un objet lié à un autre objet ou entre deux objets communicants déjà dans cet ordre. Selon les conditions de la manière suivant :
  - ✓ Aspect.PJ== le nom de l'objet ou le lien entre deux objets. Pour ajouter aspects de type objet.
  - ✓ Aspect.AD== objet ajouté (aspect de type objet).
  - ✓ Aspect.Type== «créer» pour créer les aspects de type objet ajouté.

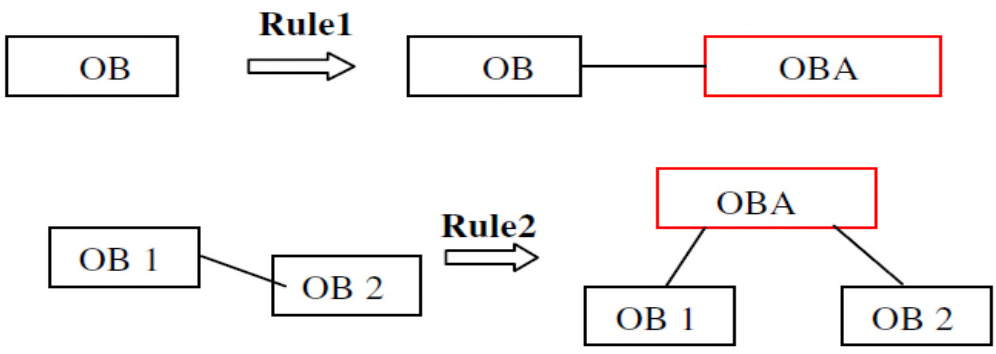

Figure 4.6 : Un exemple simple de l'application des règles 1et 2.

- **Règle (propriété 3) :** Cette règle est appliquée pour créer et ajouter un lien relié deux objets qui ne sont pas communiquer.
  - ✓ Aspect.AD== "lien"
  - ✓ Aspect.Type== «créer» ou «contexte» et pour marquer l'aspect comme visité.
  - ✓ Aspect.PJ== le nom de l'objet1(OB1) et l'objet 2(OB2) qui ne communique pas.

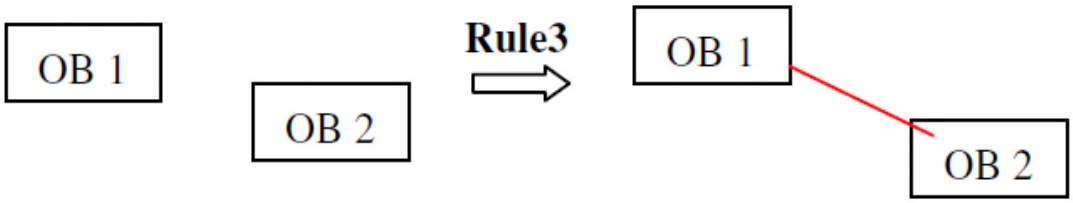

Figure 4.7 : Un exemple simple d'application la règle 3.

- **Règle 4 (propriété 4) :** Cette règle est appliquée pour traiter un aspect pas visité, et ajouter un objet qui communique les deux objets.
  - ✓ Aspect.AD = = «objet»
  - ✓ Aspect.Type== «créer» et pour marquer l'aspect comme visité.
  - ✓ Aspect.PJ== le nom de l'objet1 (OB1) et l'objet2 (OB2) qui ne communiquer.

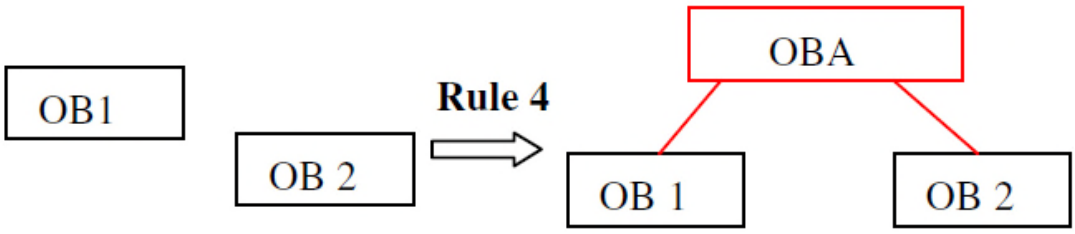

Figure 4.8 : Un exemple simple d'application la règle 4.

- **Remarque :**

✓ OB1, OB2: Les objets.

✓ OBA : Est l'aspect de type objet qui doit être inséré dans BM.

✓ Aspect.PJ : points de l'aspect de jonction.

✓ Aspect.AD : les conseils d'un aspect.

✓ Aspect.Type: l'action de l'aspect est la création, la suppression ou le lien entre deux objets.

3. Le modèle de transformation proposé

Dans notre travail, nous avons utilisé l'approche de Martin Gogola et al [17]. Pour cela, on a commencé par la modélisation de la transformation entre les diagrammes de communication orienté objet et les diagrammes de communication orienté aspect. Nous avons étudiée cette transformation sous forme d'un modèle de transformation représenté par le diagramme de classe UML de la figure (4.9) enrichie par des contraintes OCL de la figure (4.10).

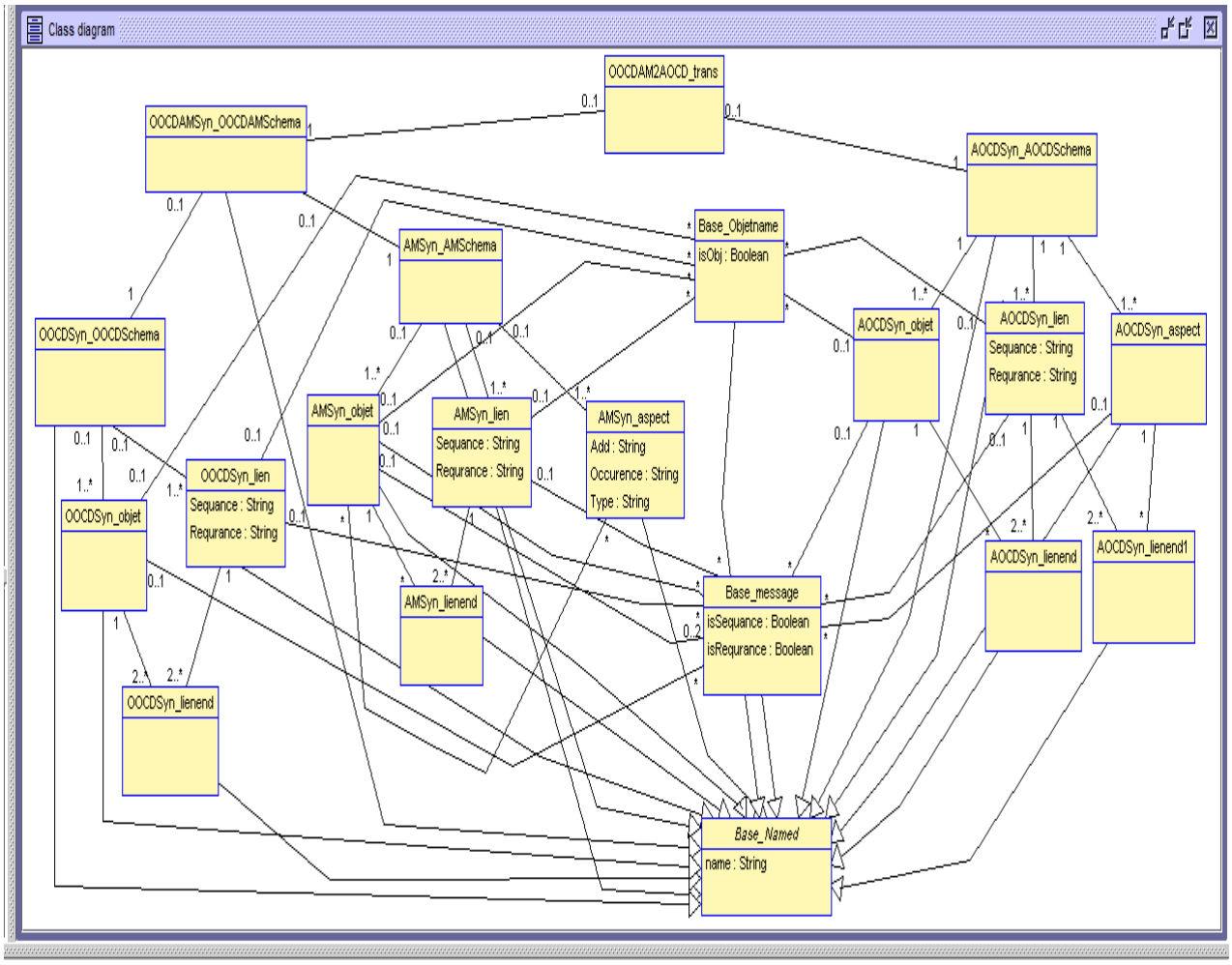

Figure 4.9 : Diagramme de class pour le modèle de transformation.

Notre modèle est composé de quatre parties comme suit : une partie Base contentent les concepts et les éléments utilisée dans les deux modèles source et cible. Une partie pour le schéma des diagrammes de communication orientée objet et le modèle aspect associée (OOCDAMSyn_OOCDAMSchema), ce dernier contient deux sous modèles : le premier représente le modèle de base (OOCDSyn_OOCDSchema), pour les diagrammes de communication orienté objet avec les concepts d'objet, de lien, et de lienend, le deuxième pour le modèle aspect (AMSyn_AMSchema), avec les concepts d'aspect, d'objet, de lien, et

de lienend. Une partie pour les diagrammes de communication orienté aspect (AOCDSyn_AOCDSchema), avec les concepts d'objet, d'aspect, de lien, et de lienend. Enfin, une partie de la transformation (OOCDAM2AOCD_trans).

Le modèle de transformation proposé est contient vint classes qui sont reliées par des associations.

Classe OOCDAM2AOCD_Trans: cette classe est utilisée pour représenter la transformation, et pour relier les deux schémas source et cible.

Classe OOCDAMSyn_OOCDAMSchema: cette classe est utilisée pour représenter le modèle source, et pour relier les deux schémas de modèle de base et de modèle d'aspect.

Classe OOCDSyn_OOCDSchema: représente les concepts de l'orientée objet.

Classe AMSyn_AMSchema : représente les concepts de l'orienté aspect.

Classe AOCDSyn_AOCDSchema : cette class utilisée pour représenter le modèle cible.

Classe Base_message : représente la relation (les messages) entre le modèle source et cible.

Classe Base_Objetname : représente la relation (les objets) entre le modèle source et cible.

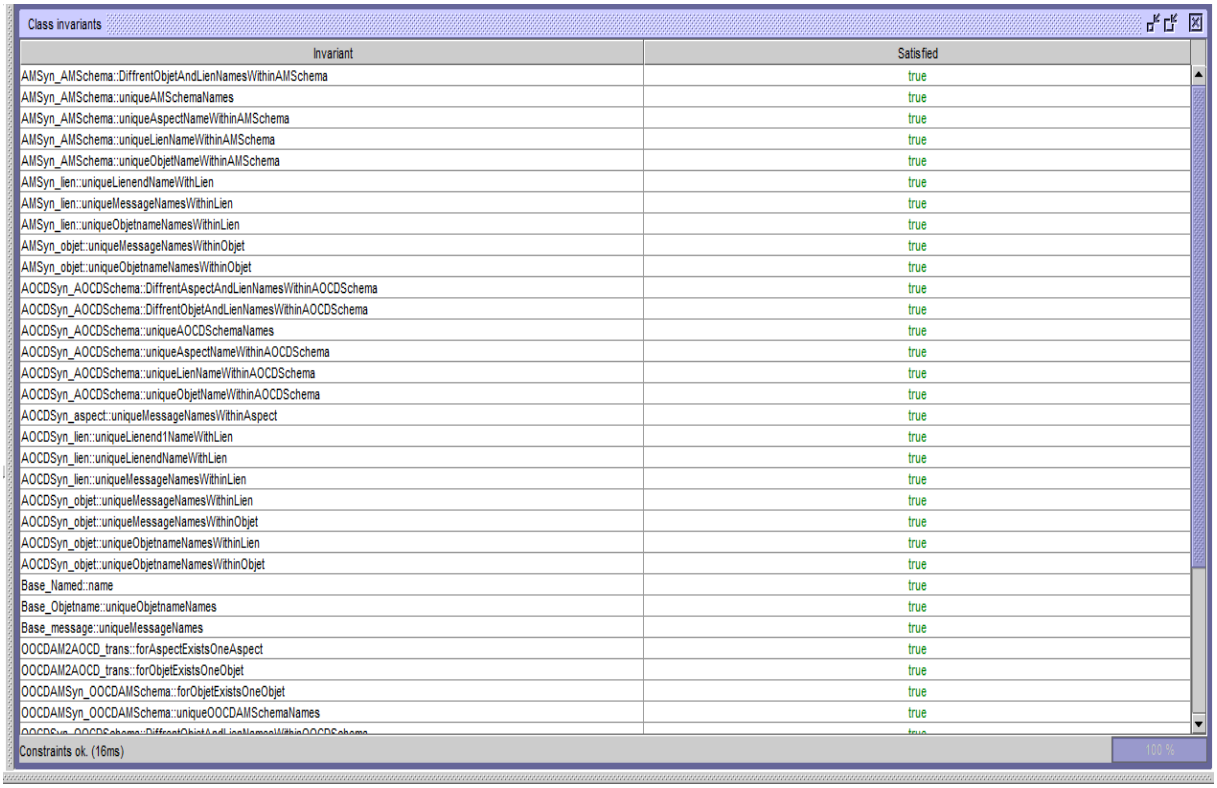

Figure 4.10 : Invariantes de classes pour le modèle de transformation.

Cette figure représente les contraintes créer pour la vérification de modèle de transformation, on a défini quelque contraintes comme suit :

--Naming restriction : Diffrent message have Diffrent Name

context self:Base_message inv uniqueMessageNames:

Base_message.allInstances -> forAll(self2|self.name=self2.name implies self=self2)

--Naming restriction : Diffrent Objetname have Diffrent Name

context self:Base_Objetname inv uniqueObjetnameNames:

Base_Objetname.allInstances -> forAll(self2|self.name=self2.name implies self=self2)

-- Diffrent OOCDAMSchema have diffrent names

context self:OOCDAMSyn_OOCDAMSchema inv uniqueOOCDAMSchemaNames:

OOCDAMSyn_OOCDAMSchema.allInstances -> forAll(self2|self.name=self2.name implies self=self2)

-- Diffrent AOCDSchema have diffrent names

context self:AOCDSyn_AOCDSchema inv uniqueAOCDSchemaNames:

```
    AOCDSyn_AOCDSchema.allInstances -> forAll(self2|self.name=self2.name implies
self=self2)

-- within one OOCDSchema,diffrent Objet have diffrent names

  context self:OOCDSyn_OOCDSchema inv uniqueObjetNameWithinOOCDSchema:

   self.Objet -> forAll(o1,o2|o1.name=o2.name implies o1=o2)

-- For every Objet in the OOCDSchema there is a AMSchema

context self:OOCDAMSyn_OOCDAMSchema inv forObjetExistsOneObjet:

  self.OOCDSchema.Objet->forAll(o |

  self.AMSchema.Objet->one(o |

  o.name=o.name ))

-- For every Objet in the OOCDAMSchema there is a AOCDSchema having the

-- same name and Objetname with the same properties

context self:OOCDAM2AOCD_trans inv forObjetExistsOneObjet:

self.OOCDAMSchema.AMSchema.Objet->forAll(o |

self.AOCDSchema.Objet->one(obj |

o.name=obj.name and

  o.Objetname->forAll(oa|

  obj.Objetname->one(obja|

  oa.name=obja.name and oa.isObj=obja.isObj))))
```

3.1. La vérification de modèle proposé

3.1.1. Vérification de la consistance (Check consistency)

- **Weak consistency:**

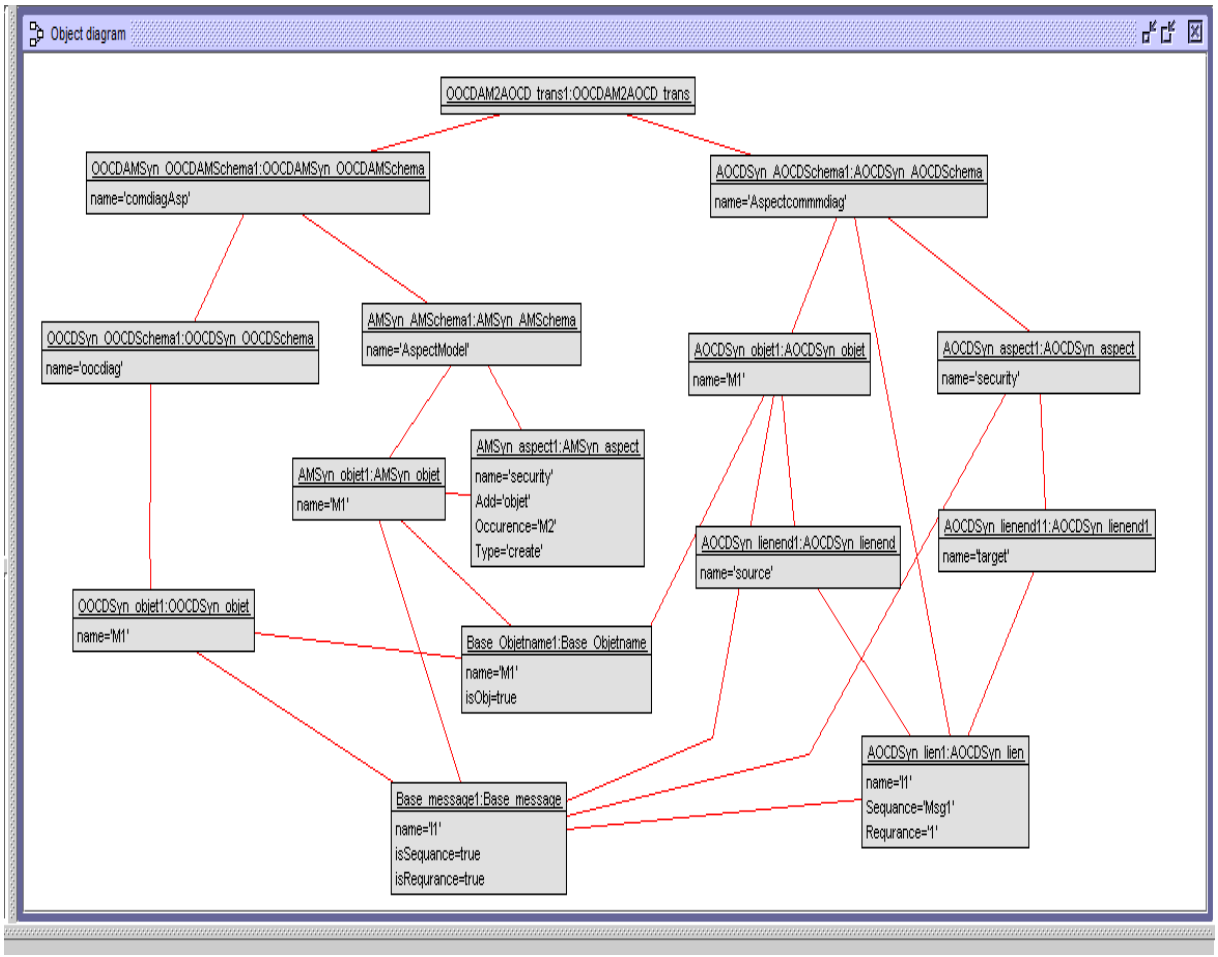

Figure 4.11 : Diagramme d'objet pour prouver (weak consistency).

Cette figure représente la vérification de la propriété (weak consistency) signifie que il existe un diagramme d'objet valide même pour une seule class instanciées dans notre modèle de transformation.

- **Class instanciability**

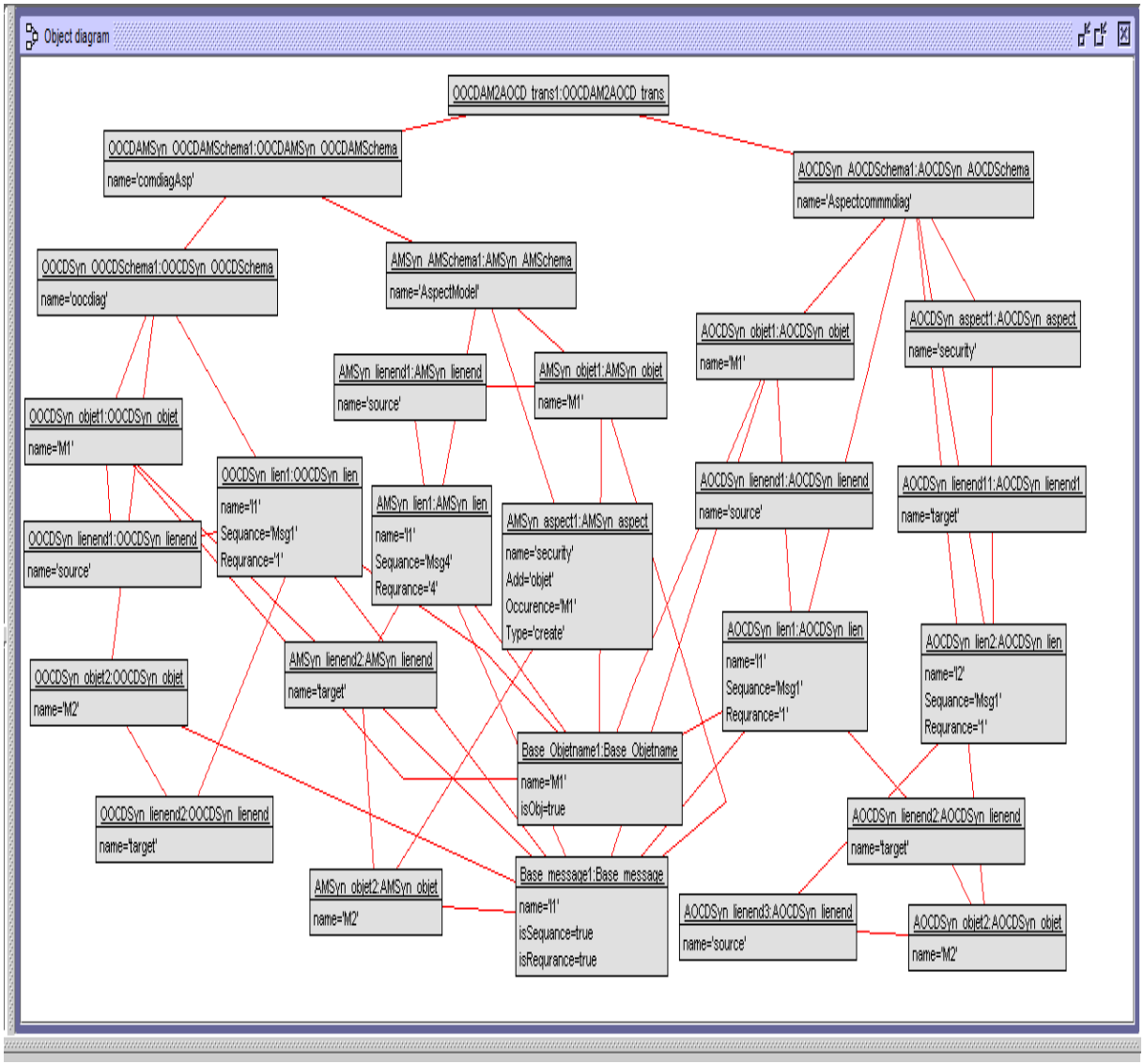

Figure 4.12 : Diagramme d'objet pour prouver (class instanciability).

Cette figure représente la vérification de la propriété (class instanciability) signifie que il existe un diagramme d'objet valide pour toutes les classes sont instanciées dans notre modèle de transformation.

- **Class and association instanciability**

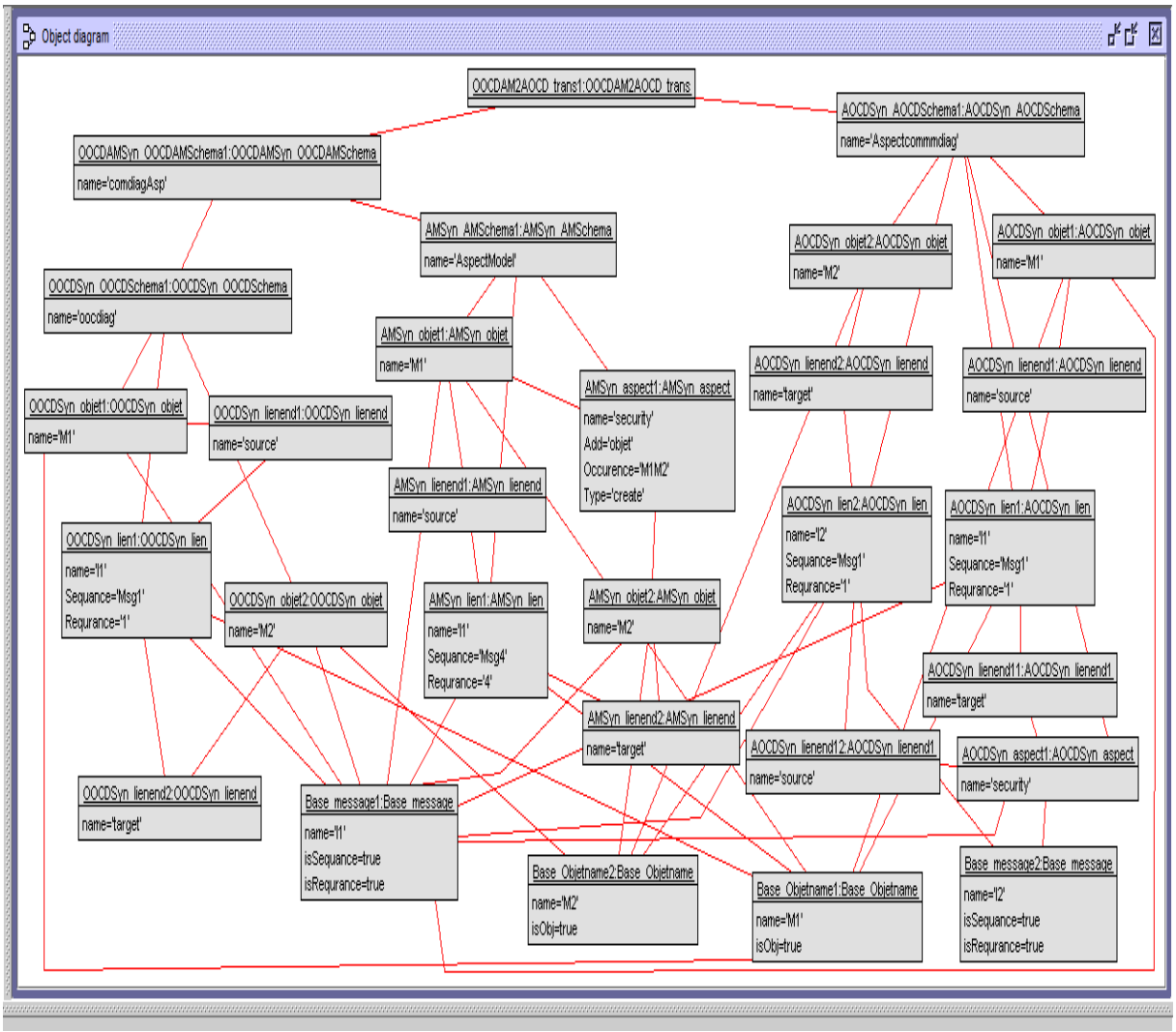

Figure 4.13 : Diagramme d'objet pour prouver (class and association instanciability).

Cette figure représente la vérification de la propriété (class and association instanciability) signifie que il existe un diagramme d'objet valide pour toutes les classes et toutes les associations sont instanciées dans notre modèle de transformation.

4. Exemple d'étude de cas

L'exemple d'étude de cas sur le processus de participation à une conférence. Il représente un aspect qui est la vérification d'information et de l'article. Cet aspect permet de vérifier les informations d'un chercheur et de vérifier si le format de l'article est conforme ou non au format de la conférence après la soumission [6].

4.1. L'exemple de transformation de modèles

4.1.1. Le modèle source

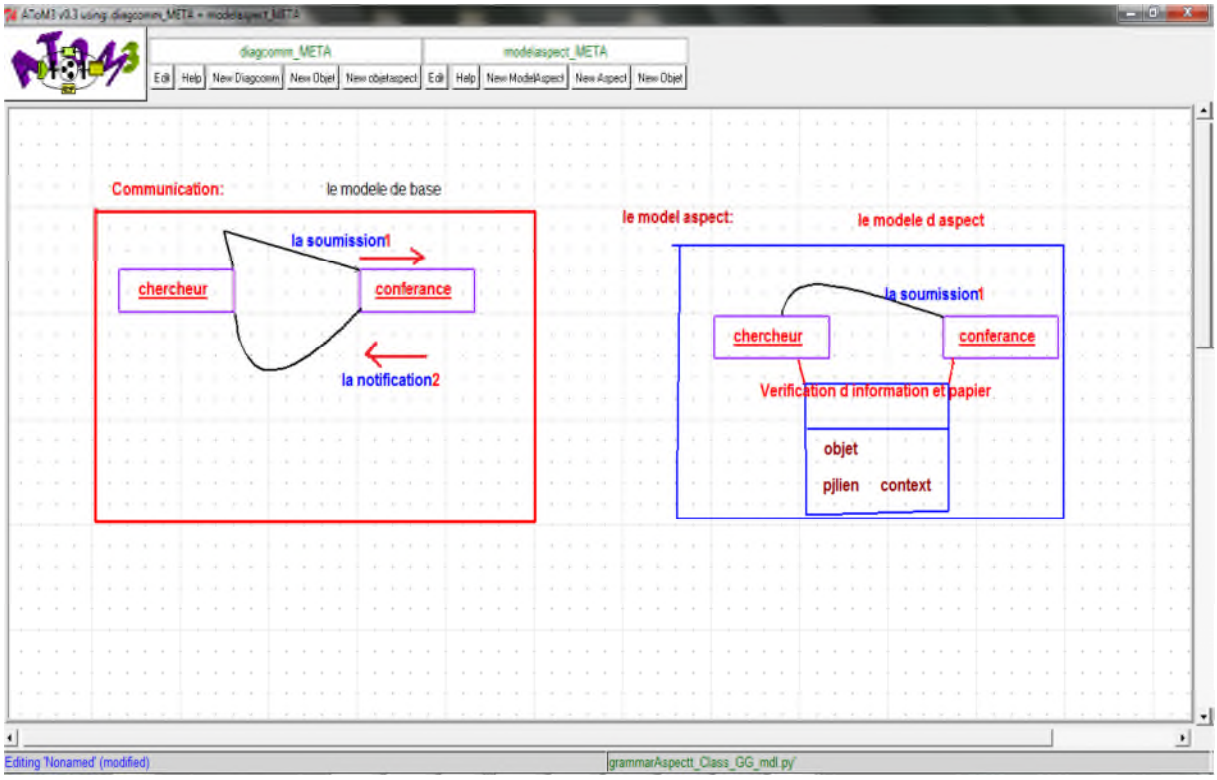

Figure 4.14 : Le modèle source.

4.1.2. Le modèle cible

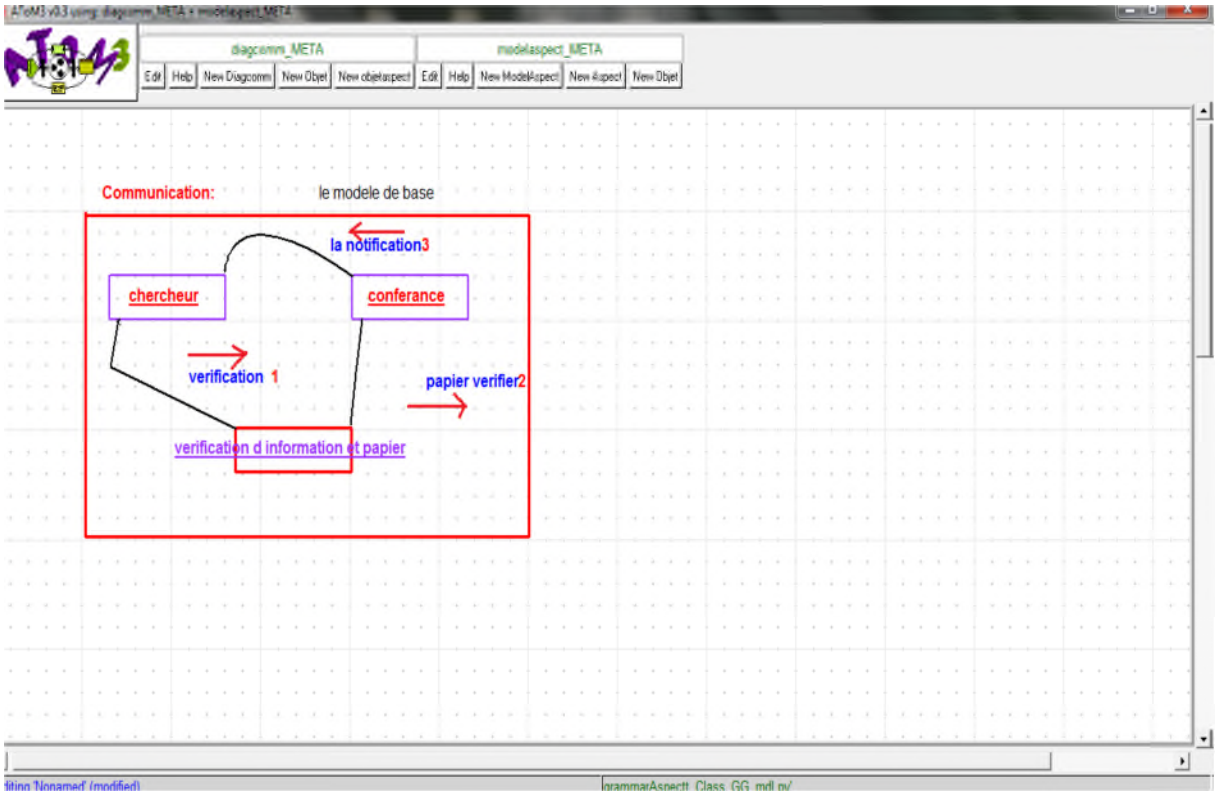

Figure 4.15 : Le modèle cible obtenu.

4.2. Le modèle objet généré pour l'exemple

Nous avons utilisé l'outil USE sur les modèles d'entrée et de sortie de l'étude de cas précédente (figure 4.14 et 4.15 respectivement) sous une forme textuelle et on a obtenu le diagramme d'objets des figures 4.16, 4.17, 4.18. Ce diagramme d'objets est considéré comme instance de la transformation. Notons que les figures représentent le même schéma divisé en trois parties.

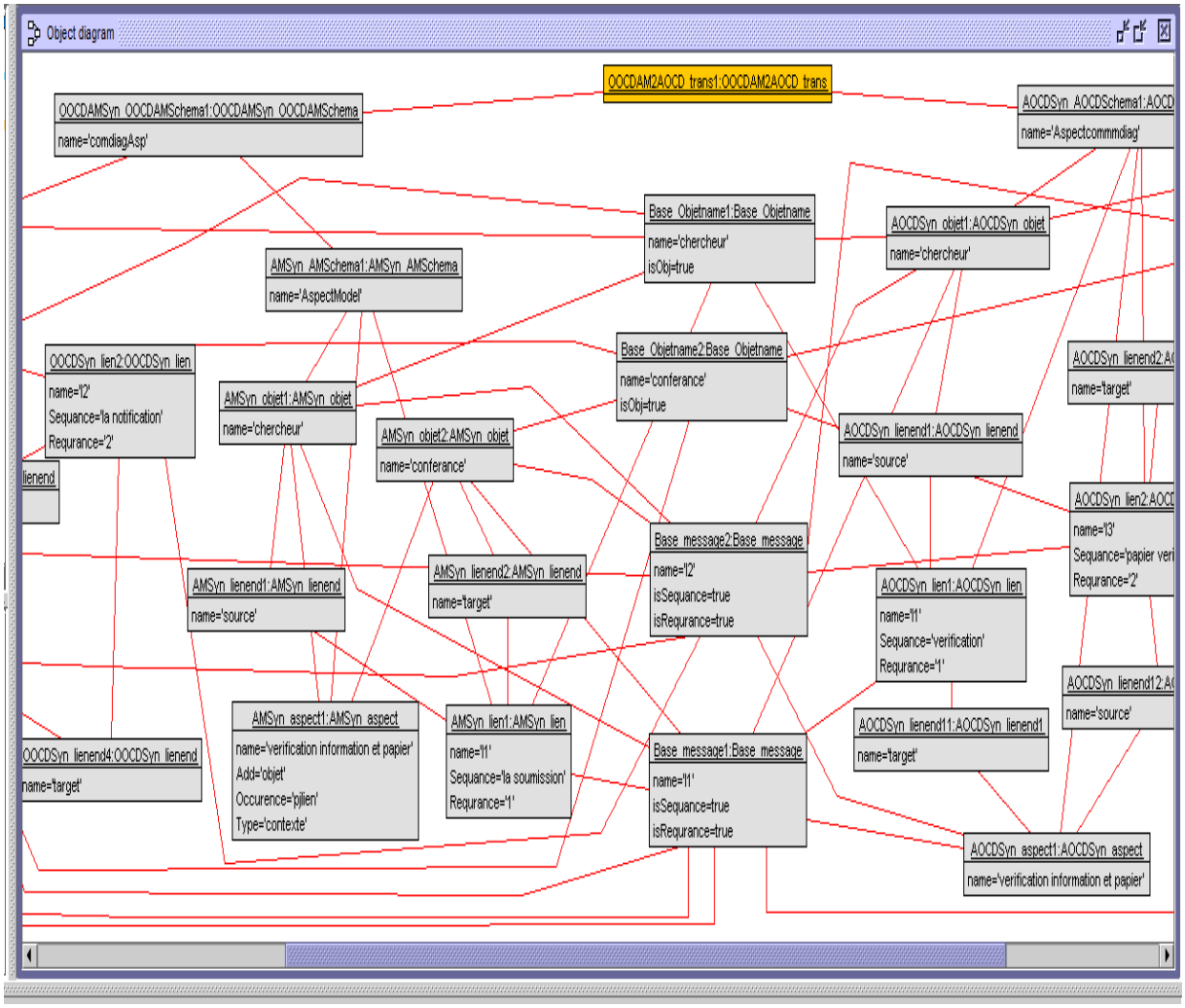

Figure 4.16 : Modèle objet généré (la partie OOCDAM2AOCD_trans)

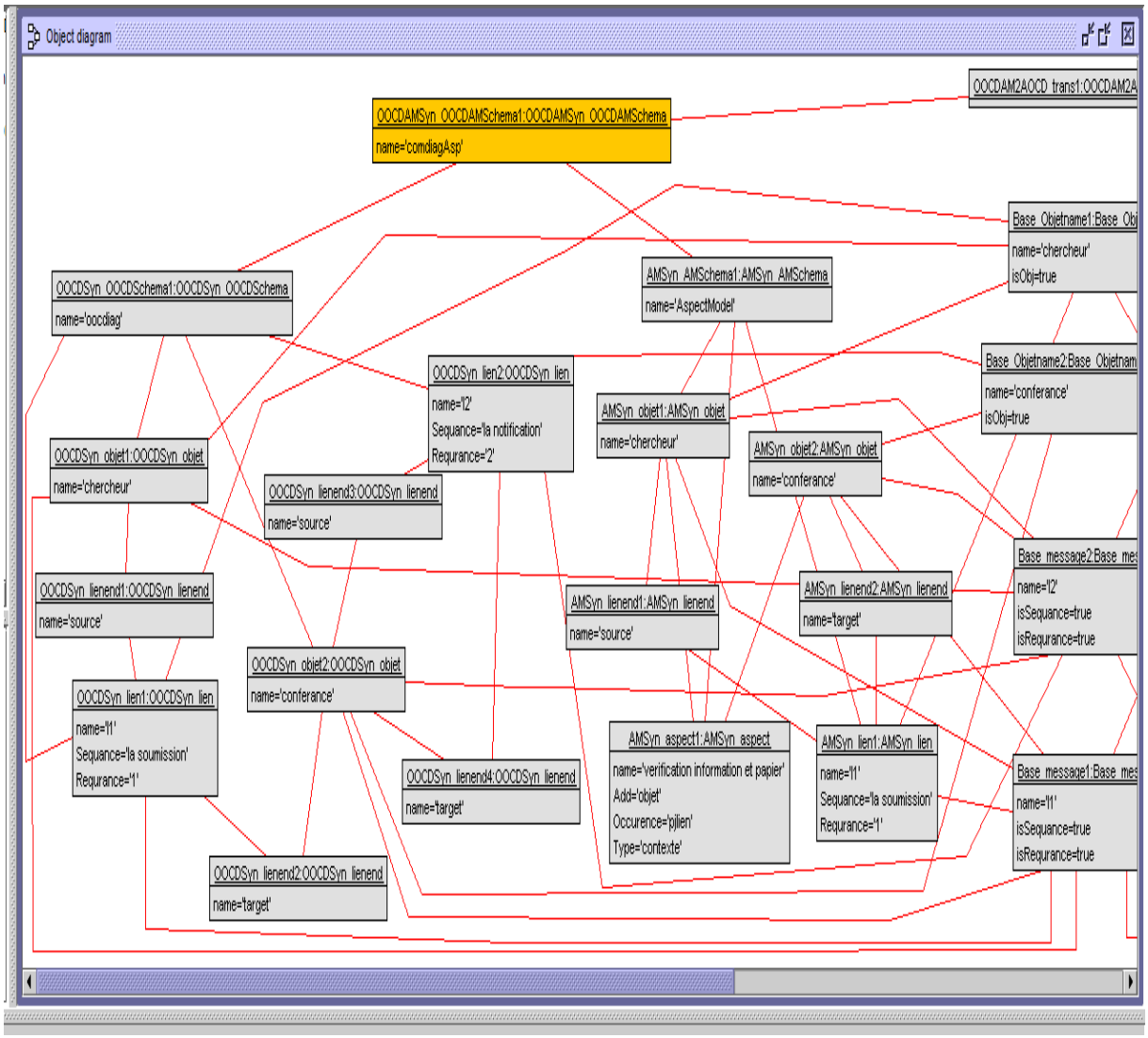

Figure 4.17 : Modèle objet générer (la partie OOCDAM_Syn).

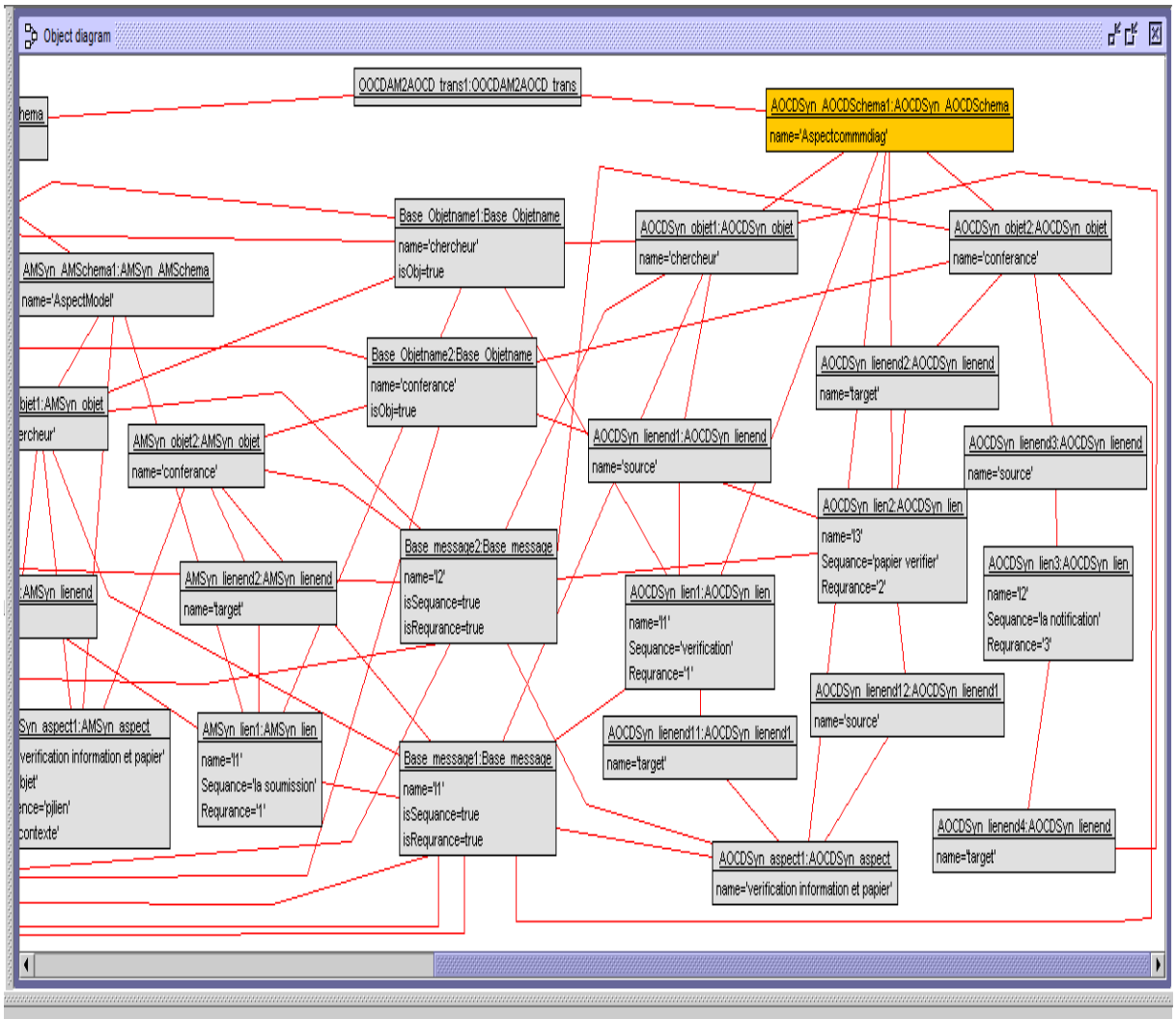

Figure 4.18 : Modèle objet générer (la partie AOCD_Syn)

Ce diagramme d'objet représente une instance de la transformation entre le diagramme de communication orienté objet vers le diagramme de communication orienté aspect en utilisant le validateur USE pour la vérification de la propriété de la consistance (weakconsistency, class instanciability, et class andassociation instanciability).

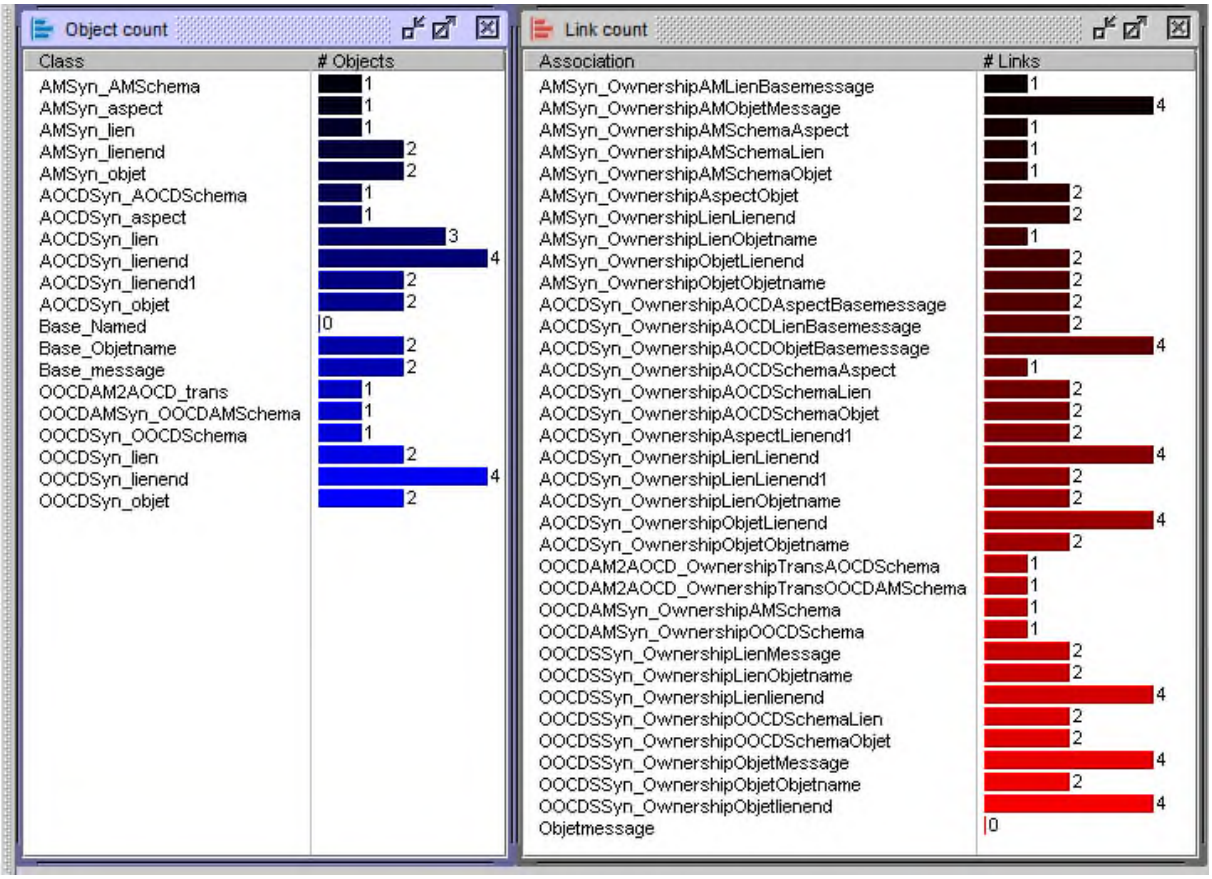

Figure 4.19: check class and association population.

Cette figure représente que toutes les classes et toutes les associations sont instanciées notre modèle de transformation.

5. Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur l'approche étudiée où l'accent a été mis sur les métas modèles des diagrammes de communication orienté objet et orienté aspect. Et plus spécifiquement les règles de transformation. Nous avons aussi décrit le modèle proposé ainsi que ses détails. Par la suite nous vérifions certaines propriétés. Enfin, nous décrivons un exemple d'étude de cas.

Conclusion

Générale

Conclusion

Le travail présenté dans ce mémoire s'inscrit dans le domaine de l'ingénierie dirigée par les modèles. Cette dernière facilite beaucoup le processus de développement de logicielle ainsi que décrive différentes vus du système à construire. Dans la démarche IDM, le processus de développement des systèmes peut être vu comme un ensemble de transformations de modèles partiellement ordonné. Chaque transformation agit sur un modèle en entrée et produit un modèle en sortie. Pour réaliser cette translation, il faut bien sûr sur établir des règles de transformation.

Le problème major de l'IDM est la vérification de transformation de modèles afin d'assurer la qualité du produit final. Le travail que nous avons présenté est situé dans le cadre de et la vérification et validation de transformations de modèles.Nous avons proposé d'abord un modèle de transformation modélisant la transformation entre les diagrammes de communication orienté objet et les diagrammes de communications orienté aspect. Par la suite, nous avons vérifié une propriété importante : la consistance de la transformation avec leurs trois options.De ce fait l'outil USE permet de définir des diagrammes de classes UML, des diagrammes d'instance conformes à ces diagrammes de classe et de vérifier des contraintes OCL sur ces instances.

Ce travail nous a conduits à explorer différents domaines :

- Découvrir le domaine de l'IDM avec ses différents formalismes et outils utiliser dans le processus de développement d'application.
- Découvrir les concepts de base des approches orienté objet et orienté aspect.
- L'utilisation de l'outil USE avec le langage OCL s'inscrit globalement dans l'activité de vérifier des contraintes sur des instances.

Dans de futurs travaux, nous envisageons de vérifier d'autres propriétés comme invariant Independence, conséquence, termination et confluence.

Bibliographie

[1] Amine El kouhane. *Specification d'un métamodéle pour l'adaptation des outils UML*. Thèse de Doctorat. Université Lille1 Sciences et technologies (2013).

[2] El-hillali Kerkouche. *Modélisation Multi-Paradigme : Une Approche Basée sur la Transformation de Graphes*. Thèse de Doctorat en Sciences en Informatique. Universite de mentouri constantine. (2011).

[3] Benoit Combemale. *Ingénierie Dirigée par les Modèles (IDM).* Etat de l'art. 2008. <hal-00371565> Submitted on 29 Mar (2009).

[4] BERRAMLA Karima. *Vérification et validation des transformations de modèles*. Mémoire de Magister en Informatique. Université d'Oran. (2014).

[5] Pascal André *Object Management Group Organisme et Principales normes*. 44322 Nantes Cedex 03.

[6] Mouna Aouag. *Des diagrammes UML 2.0 vers les diagrammes orientés aspect à l'aide de transformation de graphes.* Thèse de doctorat en Système Distribués. Université de Constantine 2. (2014).

[7] Zahaoui anissa amel. *Developpement d'une chaine d'outils du nouveau standard fondationnele UML (FUML).* Mémoire de Magister en Informatique Embarqué. Université Badji Mokhtar. (2014).

[8] Adil Anawar. *Formalisation par une approche IDM de la composition de modèles dans le profil VUML*. Thèse de doctorat en Informatique. Université de Toulouse. (2009).

[9] Daniel Calegari and Nora Szasz. *Verification of model transformation: a survey of the state-of-the-art. Electronic notes in theoretical computer science*. 292:5-25, (2013).

[10] Mouna Bouarioua. *Une approche basée transformation de graphes pour la génération de modèles de réseaux de Petri analysables à partir de diagrammes UML*. Thèse de doctorat en Système Distribués. Université de Constantine 2. (2013).

[11] Anne E. Haxthausen. *An introduction to formal méthodes for the development of safety-critical Applications.* Technical university Denmark, (2010).

[12] Bernard ESPINASSE *Unified Modelling language* Professeur à l'Université d'Aix-Marseille. (2009).

[13] Tibermacine Okba. *UML et Model Checking* Mémoire de Magister en Informatique. Université El Hadj Lakhdar – BATNA (2009).

[14] Bezivin, J., B¨uttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A. : Model Transformations ? Transformation Models! *In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. 9th Int. Conf. Model Driven Engineering Languages and Systems (Models'2006), Springer, Berlin, LNCS 4199 (2006) 440–453.*

[15] Amrani, M., Lucio, L., Selim, G.M.K., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A *Tridimensional Approach for Studying the Formal Verification of Model Transformations*. In Antoniol, G., Bertolino, A., Labiche, Y., eds.: Proc. Workshops ICST, IEEE (2012) 921–928.

[16] J. Christian Attiogbé, *Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette.* In HDR, Université de Nantes, Nantes Atlantique Université, (13 septembre 2007).

[17] Gogolla, M., Hamann, L., Hilken, F.: *Checking Transformation Model Properties with a UML and OCL Model Validator.* In: Amrani, M., Syriani, E., Wimmer, M. editors, *Proc. 3rd Int. Workshop on Verification of Model Transformation (VOLT'2014),* pages 16-25, http://ceur-ws.org/, 2014. CEUR Proceedings, Vol. 1325.

[18] Martin Gogolla, Frank Hilken. *UML and OCL Transformation Model Analysis:Checking Invariant Independence, Database Systems Group*. University of Bremen, Germany. (2015).

[19] Birgit Demuth. *The Dresden OCL Toolkit and Its Role In Information Systems Development. Dresden* University of Technology, Department of Computer Science. D-01062 Dresden, Germany.

[20] Birgit Demuth , Sten Loecher , Steffen Zschaler. *Structure of the Dresden OCL Toolkit.* Dresden University of Technology.

[21] Martin Gogolla, Fabian B¨uttner, Mark Richters *USE: A UML-based specification environment for validating UML and OCL*. University of Bremen, Bremen, Germany, EADS Space Transportation, Bremen, Germany, Received 2. (November 2005). received in revised form 6 November 2006; accepted 16 January 2007.

[22] J. Christian Attiogbé, *Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette.* In HDR, Université de Nantes, Nantes Atlantique Université, (13 septembre 2007).

[23] Sylvie ratté, Document thématique. *UML et OCL* École de technologie supérieure Département de Génie logiciel et des TI (6 février, 2005).

[24] Benoit Combemale, Xavier Crégut, Marc Pantel Presentation d'OCL 2 IRISA CNRS Laboratory, University of Rennes 1 IRIT CNRS Laboratory, University of Toulouse (15 novembre 2011).

[25] Abdelkrim Amirat *Une Approche Hybride pour la Séparation des Préoccupations avec Résolution de Conflits durant l'Ingénierie des Besoins*. Thèse de Doctorat en Informatique. Université Badji Mokhtar. (2007).

[26] Michael R Blaha, James Rumbaugh, *Modélisation et conception orientées objet avec UML 2*. (2005).

[27] Pierre-Alain Muller, Nathalie Gaertner, *Modélisation objet avec UML*. (2003).

[28] Benoit Charoux, Aomar Osmani, Yann Thiery-Mieg, UML 2. (2005).

[29] Boubendir, A. *Un cadre génirique pour la détection et la résoulution des interactions entre les aspects.* Thèse de doctorat, Université de Mentouri Constantine. (2011).

[30] Amina Gacem *Diagramme de Classe UML et Base de Données Relationnelle-objet.* Ecole des Hautes Etudes Commerciales HEC Alger. (2014).

[31] Claude Belleil *Le langage UML 2.0 OCL (Object constraint language)*. Université de Nantes.

[32] Brichau johan ET D'Hondt theo *Introduction to aspect-oriented software development* (2005).

[33] Jeannette M. Wing, *A Specifier's Introduction to Formal Methods*, *Computer*, vol. 23(9):8-23, 1990.

[34] Mouna Aouag, Wafa Chama, Allaoua Chaoui. *From UML Communication Diagrams to Aspect-Oriented Communication Diagrams Using Graph Transformation.* Université Mentouri Constantine.

[35] Jean-Marc, *Processus unifié de développement orienté objet de logiciels*. ESTIA/LIPSI.