الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université 8 Mai 1945 – Guelma

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Master Académique

Thème :

Une approche algébrique pour la transformation des modèles EMF

Encadré Par :

Mr. Berrehouma Nabil

Présenté par :

Kanté Mahamadou,

Doumbia Boubacar

Juin 2016

Remerciements

Nous remercions infiniment Dieu le tout puissant pour la santé, la force et le courage qu'il nous a donnés tout au long de notre parcours.

Notre gratitude va à l'égard de tout le corps professoral pour toutes nos connaissances acquises durant notre formation, spécialement à ***Mr. BERREHOUMA Nabil*** *pour son bon encadrement, ses conseils, sa patience et son attention qui nous ont énormément aidés dans ce travail.*

Nos remerciements les plus vifs s'adressent aussi messieurs le président, à tous les membres de jury d'avoir accepté d'examiner et d'évaluer notre travail.

Enfin nous sommes reconnaissants à toute personne qui a contribué d'une façon ou d'une autre à la réalisation de ce présent mémoire.

Dédicace Kanté

À mon père Kanté Nianguiry

A ma mère Kébé Niagalé

A mes frères ,soeurs, nièce, neuveux , oncles et tantes

Enfin A tous ceux que j'aime, à tous ceux qui m'aiment, et surtout à tous ceux qui le méritent.

Dédicace Doumbia

Je dédie le fruit de mes années d'études à mes très chers parents (Mahamadou Doumbia et Oumou Coulibaly) principalement à mes adorables mères (Oumou et Ami Fomba), et Tante Rokia Doumbia, pour l'amour et le sacrifice qu'elle a fait pour que je puisse suivre mes études dans des meilleures conditions, sans l'encouragement de mes parents et mes amis ce travail n'aurait jamais vue le jour.

A mes cousins et cousines ;

A mes Oncles et Tantes ;

A toute ma famille ;

A tous mes amis de Boulkassoumbougou au Mali et d'Algérie ;

A toutes ceux ou celles qui m'ont aidé de loin ou de près.

Table des matières

Table des figures

Résumé

L'architecture dirigée par les modèles ou MDA (Model Driven Architecture) est une approche proposée par l'OMG en 2000, et qui consiste à favoriser l'utilisation massive des modèles et l'évitement de la manipulation directe du code. L'approche MDA s'articule sur deux concepts clés : les modèles et la transformation des modèles. Plusieurs outils sont disponibles pour la mise en oeuvre de ces concepts. EMF est l'un de ces outils.

EMF (Eclipse Modeling Framwork) permet de définir des nouvelles formalises et d'éditer des modèles conformes à ces formalismes, il est possible aussi d'appliquer des transformations des modèles d'un formalisme vers un autre.

Dans la littérature, il existe plusieurs approches de transformation des modèles. L'approche algébrique consiste à exprimer la transformation sous forme d'une grammaire (comme en théorie de langage) où les parties gauches et droites des règles de production sont des modèles.

Notre travail s'inscrit dans un cadre de recherche qui consiste à concevoir et implémenter un outil permettant de concrétiser l'approche MDA en s'appuyant sur EMF pour réaliser des modèles et d'appliquer des transformations exprimées sous forme algébriques sur ces modèles.

Mots clés : Model Driven Architecture, Modèle, Transformation de Modèle, Eclipse Modeling Framework, Approche Algébrique, Règles de Production.

Abstract

The Model Driven Architecture (MDA) is an approach proposed by the OMG in 2000, and it consists to promote massive use of models and avoidance of direct manipulation code.

The MDA approach is based on two key concepts : the models and models transformation. Several tools are available for the implementation of these concepts. EMF is one of those tools.

EMF (Eclipse Modeling Framwork) defines new formalism and it allows to edit models that conform to these formalisms, it is also possible to apply transformations models of a formalism to another.

In the literature, there are several approaches to models transformation. The algebraic approach is about to express the transformation form as a grammar (like in language of theory) where the left and right portions production rules are models.

Thus, Our work is part of a research framework of designing and implement a tool to implement the MDA approach based on EMF models to achieve and implement transformations expressed in algebraic form on these models.

Keywords : Model Driven Architecture, Model, Transformation of Model, Eclipse Modeling Framework, Algebraic Approach, Production Rules.

Introduction Générale

L'architecture dirigée par les modèles ou MDA (Model Driven Architecture) est une approche proposée par l'OGM en 2000, elle consiste à mettre en avant plus l'utilisation des modèles et l'évitement au maximum de la manipulation directe du code. L'approche MDA s'articule sur deux concepts clés : **les modèles** et la **transformation des modèles**.

Plusieurs outils sont disponibles pour la mise en oeuvre de ces concepts. EMF est l'un de ces outils. Plusieurs plateformes sont disponibles pour la mise en oeuvre des différents concepts du MDA. EMF est l'un d'entre eux. **EMF (Eclipse Modeling Framwork)** permet de définir des nouvelles formalises et d'éditer des modèles conformes à ces formalismes, il est donc possible aussi d'appliquer des transformations des modèles d'un formalisme vers un autre.

En théorie, il existe plusieurs approches de transformation des modèles. **L'approche algébrique** consiste à exprimer la transformation sous forme d'une grammaire (comme en théorie de langage) dans laquelle les parties gauches et droites des règles de production sont des modèles. En ce sens, notre étude consiste à concevoir et à implémenter un outil permettant de concretiser l'approche MDA, utilisant EMF pour réaliser des modèles et d'appliquer des transformations exprimées sous forme algébriques sur ces modèles. Ce présent mémoire est divisé en quatres chapitres :

1 introduit la théorie algébrique de transformation des modèles, à travers les grammaires de graphe, nous y discutérons des différents types en occurence les NLC et les NLC-like.

2 étude sur l'approche MDA (Model Driven Architecture), l'architecture la modélisation (modèle et méta-modèle), la transformation des modèles et sur Eclipse Modeling framework, sa structure, son fonctionnement.

3 présente des généralités sur l'environnement de développement de logiciel Eclipse (integrated development environment,IDE) pour le langage de programmation java , ces origines, l'architecture d'Eclipse, et les plugins et leurs développement.

4 ce chapitre est consacré à la phase de conception et d'implémentation de notre outils de transformation des modèles exprimées sous forme algébrique

Enfin, nous clôturons ce mémoire par une conclusion générale et quelques perspectives.

CHAPITRE 1

Une approche algébrique pour la transformation des graphes

Plan du chapitre

1.1 Introduction

Nous présenterons de manière formelle l'approche algébrique de la théorie des transformations de graphes. Notamment, plusieurs types d'approche existent en théorie, le focus sera fait sur l'un d'entre eux : la grammaire de graphe edNCE.
La suite de ce chapitre présentera ainsi, d'abord quelques généralités sur certaines notions à savoir : graphes, grammaire de graphe et langage de graphe, puis au fur et à mésure avec des exemples nous tentérons d'expliquer formellement au mieux les grammaires de graphes allant de la plus simple au edNCE.

1.1.1 Définitions de base

Définition 1. `Graphe`

— Un graphe est un schéma constitué de sommets, dont certains sont reliés par des arêtes.

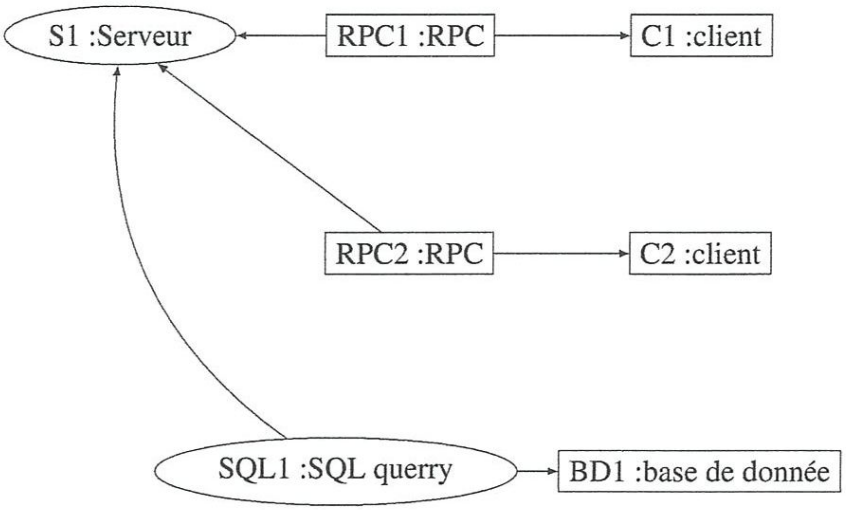

FIGURE 1.1 – Répresentation d'une application Client/Serveur sous forme de graphe

— Un graphe orienté est un graphe dont les arêtes sont orientées(fléchées).On distingue alors le sommet origine de l'arête et son extrémité.
— Deux sommet reliés par au moins une arête sont dits adjacents ou encore voisins.
— Une arête partant et arrivant au même sommet est appelée boucle. [Édu06]

Un graphe permet de répresenter des structures de données [1], les différents états de systèmes concurents et distribués ou encore l'etat d'un ensemble d'objets regis par des rélations. La figure 1.1 répresente un exemple Client/Serveur sous forme de graphe.[Sei10]

Client potentiels :

— réseaux routiers ;
— grilles de calculs ;
— bases de données ;
— systèmes discret ;
— réseaux sociaux ;
— .....

Définition 2. `Grammaire de graphe`

Une grammaire de graphe fournit un mécanisme dans lequel les transformations locales sur un graphe peuvent être modélisés d'une façon mathématiquement précise.

Cette grammaire est principalement un ensemble fini de productions ; une production est, en général, un triplet (M, D, E) où M et D (Mother pour M, Daughter pour D) sont des graphes et E est un ensemble de mécanisme(règles) de connexion.

Une règle de production peut être appliquée à un ("host") graphe H chaque fois qu'il y a

1. Ensemble organisé d'informations ayant quelque chose en commun et qu'on a groupées pour leur traitement, e.g : nom,prénom,âge,proffesion

une occurrence de M dans H, en le remplaçant par (une copie isomorphe [2]) D, et enfin en utilisant le mécanisme de connexion E pour attacher D au reste H^- de H.
De nouveaux arcs sont utilisés comme des liens qui relient D à H^- i.e. les liens dont un des extremités appartient à D, et les autres à H^-.
Lorsque M est retiré de H, tous les arcs entre les noeuds de M et les noeuds de H qui n'appartiennent pas à M sont retirés aussi et logiquement les nouveaux liens seront définis en fonctions des anciens noeuds. [Roz82]

Définition 3. `Grammaire de graphe (formellement)` une grammaire de graphe est un uplet de la forme $G = (\sum, \Delta, P, C, S)$ où :

- $\sum - \Delta$ et Δ (avec $\Delta \subseteq \sum$) sont des alphabets de noeuds non-terminal et terminal respectivement.
- P est un ensemble fini de règles production.
- C est une relation de connexion, i.e., un couple (x,y) d'élement de $\sum$ où x est un noeud de M et y un noeud de D .
- S est le graphe initial.[Roz82]

Définition 4. `Alphabet`
Un alphabet est un ensemble fini de symboles.

- $A = 0, 1$
- $\sum = a, b, c$
- $\Delta = if, then, else, a, b$

Définition 5. `Langage`
Un langage sur un alphabet X est une partie de X^*. C'est donc un ensemble de mots.

$$L \subset X^* o L \in P(X^*)$$

Soit $X = a, b$:

- θ (l'ensemble vide) est un langage
- ε est un langage
- a, ba, bba est un langage
- .....

2. Deux objets sont dits isomorphes s'il existe un isomorphisme ; un isomorphisme entre deux ensembles structurés est une application bijective qui préserve la structure et dont la réciproque préserve aussi la structure de l'un vers l'autre

FIGURE 1.2 – Une production d'une grammaire NLC

1.2 Remplacement de noeud et la méthodologie NLC

La façon habituelle de reécrire un graphe H en un graphe H' est de remplacer un sous-graphe M de H par un graphe D et de lier D au reste de H.
Le Mécanisme de connexion lie D au reste H^- en construisant des arcs, i.e., en établissant des liens entre certains noeuds de D et certains noeuds de H^-.

Dans le cas restreint de remplacement de noeud, le graphe M consiste seulement en un **seul noeud**, une "unité locale" du graphe hôte.
Un exemple typique de mécanisme de remplacement de noeud est le mécanisme NLC(Node label Controlled mechanism). Les productions sont du type noeud-remplacement et les instructions de connexion connectent le graphe D au voisinage du noeud mère.
Une production NLC est de la forme $X \rightarrow D$ où X est une étiquette de noeud (non terminal), et D est un graphe non orienté avec des étiquettes de noeud(terminal ou non-terminal).
Une telle production peut être appliquée à tout noeud m dans le graphe hôte qui a le label X(i.e., il n'y a aucune condition d'application) ; le résultat sérait le remplacement du noeud m dans le graphe M par D.[Roz82]

Exemple 1. Considérons la production $X \rightarrow D$ (voir Fig.1.2), avec

1. les règles de connexion (c,a), (b,b) et (b,X), où X est un label de noeud non-terminal et a,b et c sont des étiquettes de noeud terminaux.
2. Soit le graphe hôte (voirFig.1.3(a)).
3. soit m le noeud de H qui porte l'étiquette X.

L'application de $X \rightarrow D$ sur le noeud m est montrée en deux étapes dans les figures 1.3(b) et 1.4.
Le premier montre le résultat du remplacement de m par D ; m avec ses 3 arêtes sont supprimés et et D est ajouté au reste H^- de H.
la figure 1.4 quand à elle represente le résultat final H', i.e., le résultat de l'integration D à H^- (remarquez les nouveaux liens en gras sur la figure) ; Ainsi H est transformé en H' par l'application de $X \rightarrow D$

Exemple 2. Considérons une grammaire de graphe NLC $G = (\sum, \Delta, P, C, S)$ avec

1. le langage L(G) étant l'ensemble des élements de $(abc)^+$ avec une liaison supplémentaire avec tout les noeuds avec l'étiquette b.

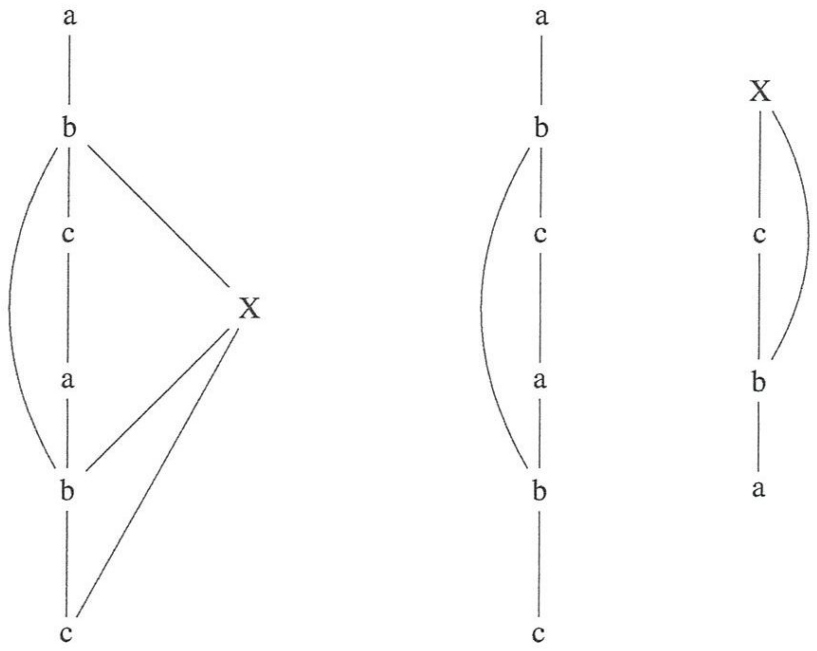

FIGURE 1.3 – (a) Graphe host. (b)Remplacement du noeud de M par D

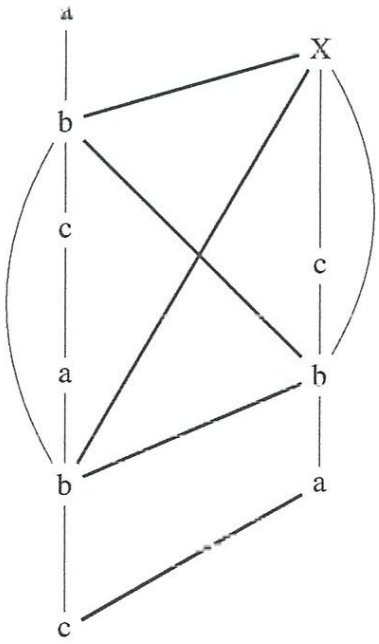

FIGURE 1.4 – Intégration de D

2. G est defini comme suit : $\sum = X,a,b,c$,$\delta = a,b,c$, $S = X$, et l'ensemble des regles de production sont celle de la Fig. 1.2 et $X \rightarrow abc$, et $C = (c,a),(b,b),(b,X)$.

Les graphes intermédiaires, ou de formes propositionnelles, générées par cette grammaire sont des chaînes de caractère dans $(abc)^*X$ avec des arêtes additionnelles entre tout les noeuds avec le label b ou X. $\ominus$ La grammaire NLC est une des tentatives de définition d'une classe de grammaire de graphe hors-contexte, i.e., une grammaire qui sérait similaire aux grammaires algébrique habituelles pour les chaines de caractères.
Elle l'est dans le sens ou ils sont complètement locaux et il n'y a aucune restriction sur l'application d'une règle.
Mais, en général la grammaire NLC n'a pas toutes les propriétés d'une grammaire hors-contexte au sens de l'indepandance de l'ordre d'application des règles par rapport au résultat.

1.3 La grammaire edNCE

C'est souvent plus commode d'être capable de faire référence aux noeuds dans le graphe D directement dans les règles de connexion plutôt qu'à travers leurs étiquettes.
D'où, dorénavant chaque instruction de connexion sera maintenant de la forme (μ,x), où x est un noeud dans le graphe D (i.e, dans un des côtés droits des règles de productions de la grammaire), et, comme avant, μ est une étiquette de noeud qui est un voisin d'un noeud du graphe M ;
cela signifie que le noeud x devrait etre connecte a chaque noeud portant l'etiquette μ.
Cela nous donne une façon commode de faire une distinction entre les noeuds individuels dans D.
Les grammaires NLC-Like [3] avec ce type de mécanisme de connexion sont appelées les grammaires de graphe NCE (Neighbourhood controlled Embedding), i.e, grammaires de graphe avec un mécanisme de contrôle d'adjacence.

Cet acronyme accentue la localité du processus de liaison que les grammaires NCE héritent des grammaires NLC. Les grammaires NCE restent des grammaires NLC tant qu'il est question de remplacement ; c'est du NLC avec respect de voisinage du noeud mère car les noeuds du graphe D sont accessible directement indépendanment de leurs étiquettes.
Formellement, une grammaire NCE est un système de la forme $(\sum,\Delta,P,C,S)$ tel que $\sum,\Delta$ et S sont comme avant dans le NLC, et P est un ensemble finis de règles où chaque production est maintenant de la forme $X \rightarrow (D,C)$ tel que $X \rightarrow D$ est une règle NLC et C, une

3. Nom donné au grammaires qui derrive du NLC(extentions/généralisation/...)

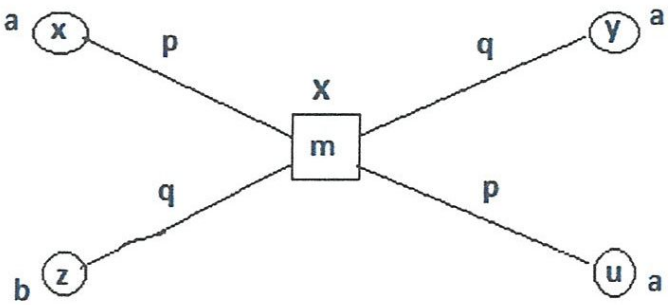

FIGURE 1.5 – Adjacence du node de M

rélation de connexion pour le graphe D, i.e., $C \subseteq \sum \times V_D$ (où V_D est l'ensemble des noeuds du graphe fille D).[Roz82]

Exemple 3. Considérons le language L de l'exemple 2 mais avec $b = a$, donc les noeuds du graphe ont comme label soit a, soit c uniquement.Une grammaire NCE (similaire à celle de l'exemple 2) qui génere L aura les regles suivantes :
$X \to (D_1, C_1)$ et $X \to (D_2, C_2)$,
où $X \to D_1$ est montré dans la figure 1.2 (mais avec b=a), $C_1 = (c, x_1), (a, x_2), (a, x_4)$, $D_2 = aac$, et $C_2 = (c, x_1), (a, x_2)$,
où les noeuds de D_1 et D_2 sont numerotés de gauche à droite. Les figures 1.3 et 1.4 (avec b=a) montrent une illustration de cette grammaire.⊖

En plus des étiquettes de noeud, les liens peuvent aussi avoir des labels.

Exemple 4. Pour le graphe hôte de la figure 1.5, lors du remplacement du noeud m, le mécanisme de connexion peu faire la différence entre ses voisins x et y (tous avec le même label a), car x est p-voisin de m (i.e., est connecté à m par une arrête avec le label p) alors que le noeud y est q-voisin de m). Mais il est aussi à noter qu'il est impossible de différencier les noeuds x et u puisqu'ils ont la même étiquette p. ⊖

Il serait simple d'étendre le NLC aux graphes orientés, la relation de connexion C est alors un triplet (μ, δ, d), où $d \subset in, out$ pour pouvoir différencier un arc entrant ou sortant d'un noeud du graphe M.
Ainsi on utilise un d (pour directed) miniscule pour indiquer qu'une grammaire de graphe NLC-like génère un graphe orienté.
De même, on utilise un e minuscule pour indiquer qu'elle génère un graphe avec des liens étiquettés en plus bien sur des étiquettes de noeud.

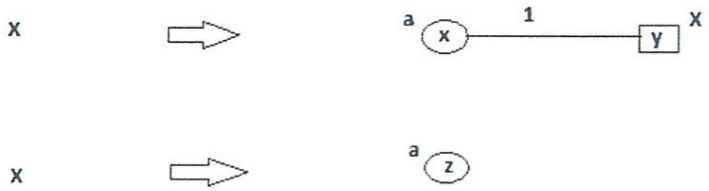

FIGURE 1.6 – Une production d'une grammaire eNCE (sans relation de connexion)

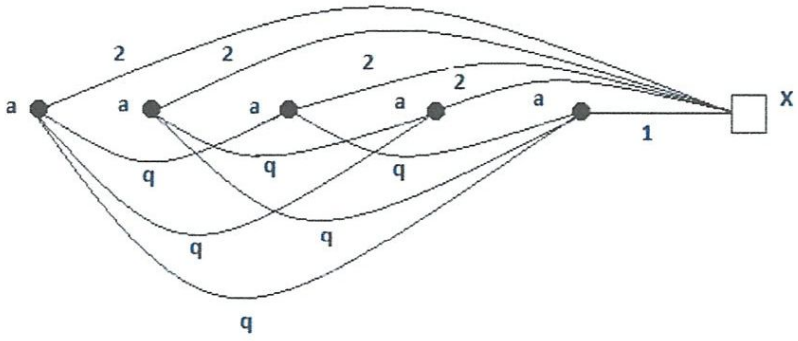

FIGURE 1.7 – Forme propositionnelle d'une grammaire eNCE

Alors on peut avoir alors les grammaires dNLC, eNCE et edNLC. L'exemple suivant montre l'usage de re-étiquage d'arête dynamique voir Fig.1.6

Exemple 5. Soit G une grammaire eNCE qui génere des liens complets de tout chaînes dans a^+ i.e., tout graphe avec des noeuds $x_1,\ldots,x_n(n \geq 1)$ d'étiquette a et des liens x_i,x_j avec $i \leq j-2$;
tous les arêtes ont l'étiquette q.
Sachant qu'une règle d'une grammaire eNCE est de la forme $(\mu, p/q, x)$ où μ est une étiquette de noeud, p et q sont des étiquettes d'arête, et x un noeud du graphe fille D.
$G = (\sum, \Delta, \Gamma, \Omega, P, S)$ est defini comme suit :

1. $\sum = X, a$
2. $\Delta = a$
3. l'alphabet des labels d'arête Γ est 1,2,q avec l'étiquette final de lien $\Omega = q \subseteq \Gamma$
4. S=X
5. P a deux règles de productions : $X \rightarrow (D_1, C_1$ et $X \rightarrow (D_2, C_2$
6. $X \rightarrow D_1$ et $X \rightarrow D_2$ sont comme dans la figure 1.6 , $C_1 = (a, 2/q, x), (a, 2/2, y), (a, 1/2, y)$ et $C_2 = (a, 2/q, z)$. Voir Fig 1.7

Les étiquettes de liaison 1 et 2 divise les voisins des noeuds non-terminaux en deux types : le premier à sa gauche, et les autres noeuds.L'application de la règle $X \rightarrow (D_1, C_1$ conduit à un changement de type 1-voisin de X : après cela devient du type 2 ; Ce changement dynamic de type est causé par le mécanisme de connexion (a,1/2,y). ferrées [DEGM05], i.e., une grammaire C-eNCE. $\ominus$

1.4 conclusion

Dans ce chapitre, nous avons présenté un aperçu sur la théorie algébrique pour la transformation des graphes, fondément du travail en question, et nous avons pu découvrir l'étendue de ce dernier du NLC jusqu'au edNCE. Dans le chapitre suivant nous présenterons un IDE [4] outils de modelisation qui nous permettra de concrétiser le concept théorique qu'on a vue dans ce chapitre.

4. Integrated Development Environment, environnement de développement intégré réunissant tous les outils nécessaires à la creation d'applications, aussi complexes qu'elles soient.

CHAPITRE 2

Eclipse Modeling Framework

Plan du chapitre

2.1 Introduction

Eclipse Modeling Framework (EMF) est une plateforme qui exploite les facilités d'Eclipse [1] pour fournir un outillage puissant de manipulation des modèles et de génération automatiques des environnements de modélisation appropriés à des domaines spécifiques. Il s'agit de la concrétisation de l'approche MDA [2] dans le contexte d'éclipse. Dans ce chapitre, nous allons nous intéressé sur la transformation des modèles et la génération de code pour la construction des outils et d'autres applications basées sur une structure de modèle de donnée, ainsi que l'utilisation de quelques modèles standards (XML [3], XMI [4]).

2.2 Modélisation et transformation des modèles

2.2.1 Modèles et Modélisation

Dans notre projet on utilise l'approche MDA pour la modélisation ainsi que la transformation de nos modèle. Dans ce qui suit nous discutérons les concepts de l'approche MDA.

Modèles Un modèle est une répresentation abstraite de la réalité. L'approche MDA vise à mettre en valeur les qualités intrinsèques des modèles à savoir la pérennité, la productivité et la prise en compte des plateformes d'exécution. Afin de permettre la génération automatique de la totalité du code des applications et d'obtenir un gain significatif de productivité. Les modèles spécifient différents niveaux d'abstraction, facilitant la gestion de la complexité inhérente aux applications.

Méta-modèles un méta-modèle définit la structure que doit avoir tout modèle conforme à ce méta-modèle. Autrement dit, tout modèle doit respecter les règles de structuration de son méta-modèle. Par exemple (voir Fig. 2.1), le méta-modèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc. D'une manière générale, les méta-modèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de

1. Voir le chapitre 3
2. Model Driven Architecture
3. Extensible Markup Language
4. Metadata Interchange

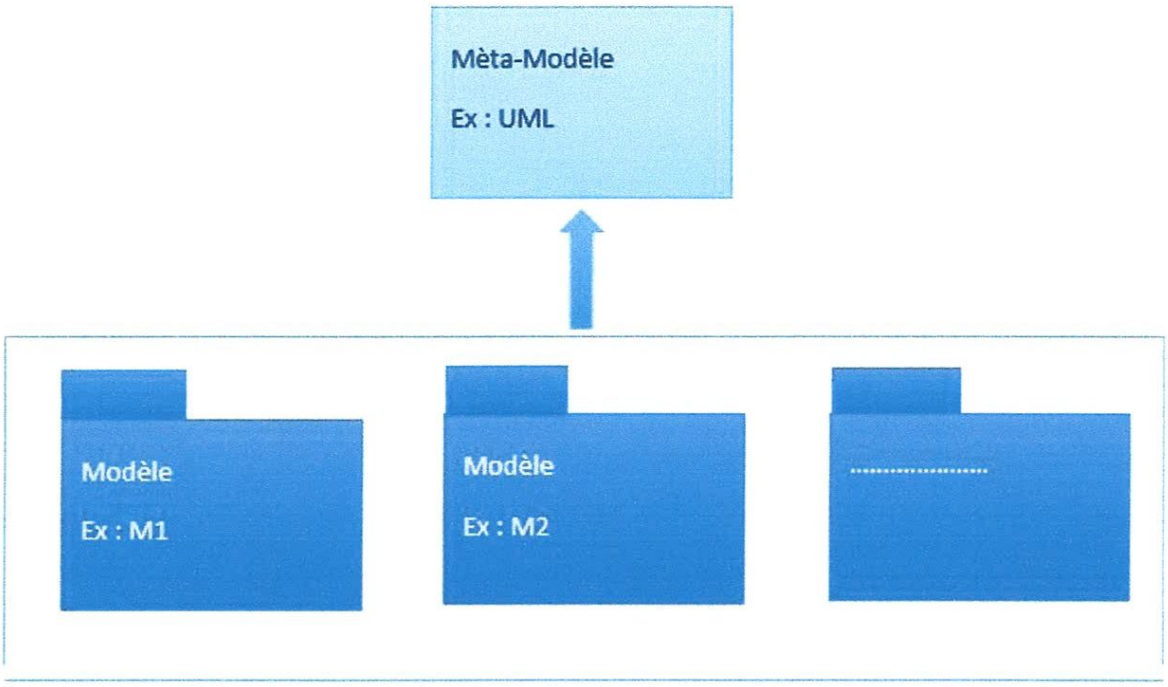

FIGURE 2.1 – Relation entre Modèle et Méta-modèle

leurs connexions et de leurs règles de cohérence. MOF[5] les représente sous forme de diagrammes de classes.[eK11]

2.2.1.1 Architecture MDA

L'architecture dirigée par les modèles ou MDA est une démarche qui s'articule sur la manipulation et la transformation de modèle. (voir Fig.2.2).

D'une manière générale, il s'agit de modéliser le domaine que l'on veut traiter de manière indépendante de l'implémentation (niveau matériel ou logiciel). Ceci permet une grande réutilisation des modèles. L'approche MDA se concentre sur les modèles, fournissant un niveau supérieur d'abstraction pendant le développement et permettant la séparation entre modèles indépendants de la plateforme(les modèles PIM, Platform indépendant model et les modèles PSM,Platform Specific Model).

En MDA, le code source n'est plus retenu comme l'élément central d'un système, mais plutôt la construction de modèle. Cette approche gravite autour de trois concepts de base :

1. les modèles,
2. les méta-modèles

5. Méta Object facility

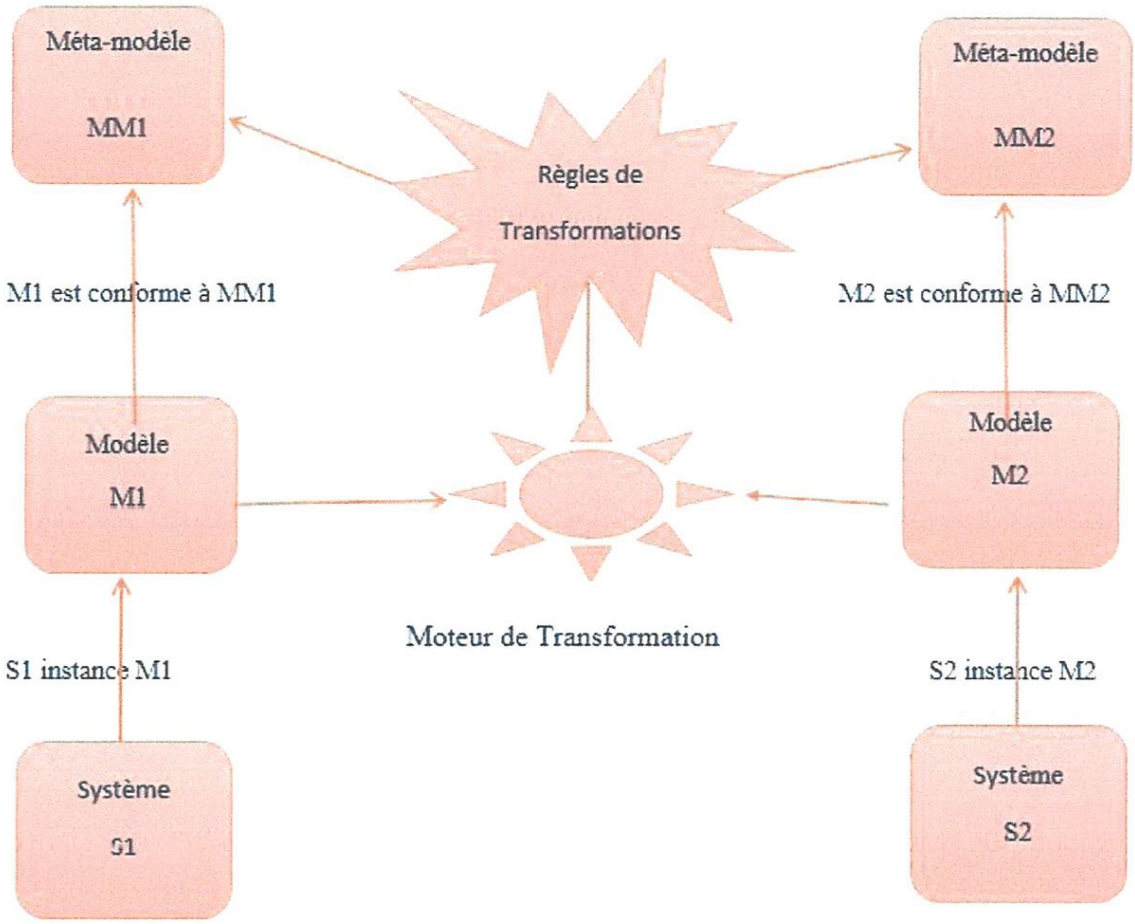

FIGURE 2.2 – Concepts de base de l'approche MDA

3. les transformations de modèles

L'élaboration d'un nouveau système informatique commence par la définition d'un ou de plusieurs modèles d'exigences : CIM [6]. Elle se poursuit par l'élaboration des modèles d'analyse et de conception abstraite de l'application : PIM. Les modèles PIM ne contiennent aucune information sur les plates-formes d'exécution.[SF14]

PIM PIM ou modèle d'analyse et de conception abstraite dans l'approche MDA, cette phase utilise elle aussi un modèle. Cette étape consiste à structurer l'application en modules et sous modules, et dans laquelle est appliqué les modèles de conception, c'est-à-dire celle qui est réalisable sans connaitre aucune des techniques d'implémentation.[cK11] Le PIM de base représente uniquement les capacités fonctionnelles métiers et le comportement du système, sans "dégradations" par des considérations technologiques. La clarté de ce modèle doit permettre à des experts du domaine de le comprendre bien mieux qu'un modèle d'implémentation. Ils peuvent ainsi vérifier plus

6. Computation Independent Model

facilement que le PIM est complet et correct. Les PIM souvent intègrent des aspects technologiques et architecturaux mais toujours sans détails spécifiques à une plateforme. Ces modèles peuvent, par exemple, contenir des informations sur la persistance, les transactions, la sécurité, etc. Ces concepts permettent de projeter plus précisément le modèle PIM vers un modèle spécifique : PSM.

PSM Platform Specific Model, Il est dépendant de la plateforme technique spécifiée par l'architecte. Le PSM sert essentiellement de base à la génération de code exécutable vers la ou les platesformes techniques. Il décrit comment le système utilisera cette ou ces plates-formes.Il existe plusieurs niveaux de PSM :

— Le premier, issu de la transformation d'un PIM, se représente par un schéma UML 2.0 spécifique à une plate -forme.

— Les autres PSM sont obtenus par transformations successives jusqu'à l'obtention du code dans un langage spécifique (Java, C++, etc.) Un PSM d'implémentation contiendra par exemple des informations comme le code du programme, les types pour l'implémentation, les programmes liés, les descripteurs de déploiement.[SF14]

2.2.2 Transformation des modèles

Afin d'obtenir des résultats utiles au développement, nous avons besoin des transformations de modèles(voir Fig.2.3) pour rendre les modèles productifs. Pour ces raisons, l'approche MDA permet notamment de transformer un modèle vers un autre modèle. En parlant de transformation, on distingue quatre types de transformations de modèles qui sont :

1. PIM vers PIM : Sans augmenter aucune information liée à la plateforme, on effectue cette transformation pour l'enrichissement, la filtration ou la spécialisation des informations des modèles. Par exemple, le passage du modèle d'analyse à celui de conception. Cependant, ces transformations ne sont pas toujours automatisables. Le fait de passer d'un PIM à un autre PIM est appelé raffinement. Ce processus consiste à introduire des détails supplémentaires dans le modèle.
   NB : Il est aussi utilisé pour le passage de PSM à PSM.
2. PIM vers PSM : Tout d'abord on doit s'assurer que le PIM est suffisamment raffiné pour pouvoir être spécialisé vers une plate-forme donnée, après on applique les transformations qui nous permettent d'avoir par construction une bonne partie des modèles PSM à partir des modèles PIM. Ces transformations sont les plus utiles de MDA car elles donnent une garantie de la pérennité des modèles aussi bien que leur productivité et leur lien avec les plates-formes d'exécution. Dans la transformation,

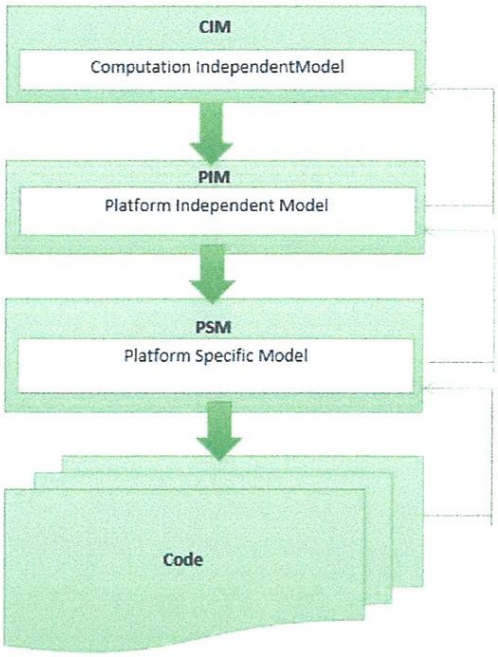

FIGURE 2.3 – Transformations de MDA

il y a des règles qui doivent être généralisées et capitalisées pour obtenir dans le futur une automatisation importante.

3. PSM vers PSM : Une transformation PIM vers PSM n'est pas toujours suffisante pour permettre la génération de code d'où la nécessité de passer de PSM à PSM en utilisant des formalismes intermédiaires. Par exemple, pour générer un code C++, à partir d'un formalisme en UML 2.0, un passage d'UML 2.0 vers SDL[7] puis de SDL vers C++ pourrait être utilisé. La transformation PSM à PSM (raffinement) s'effectue lors des phases de déploiement, d'optimisation ou de reconfiguration.
4. PSM vers PIM : Cette transformation est utilisée pour revenir à un modèle indépendant de plate-forme (PIM) à partir d'un modèle spécifique de plate-forme (PSM) ou éventuellement du code. C'est une opération de rétro-ingénierie (reverse engineering) qui est assez complexe à réaliser et difficilement automatisable. Ces transformations sont néanmoins nécessaires pour permettre l'intégration d'applications existantes dans le processus MDA.[SF14]

2.3 Eclipse Modeling Framework

Définition 1. EMF

Pour répondre à la question , " Qu'est-ce que EMF ? ", Nous empruntons la description

7. Specification and Description Language

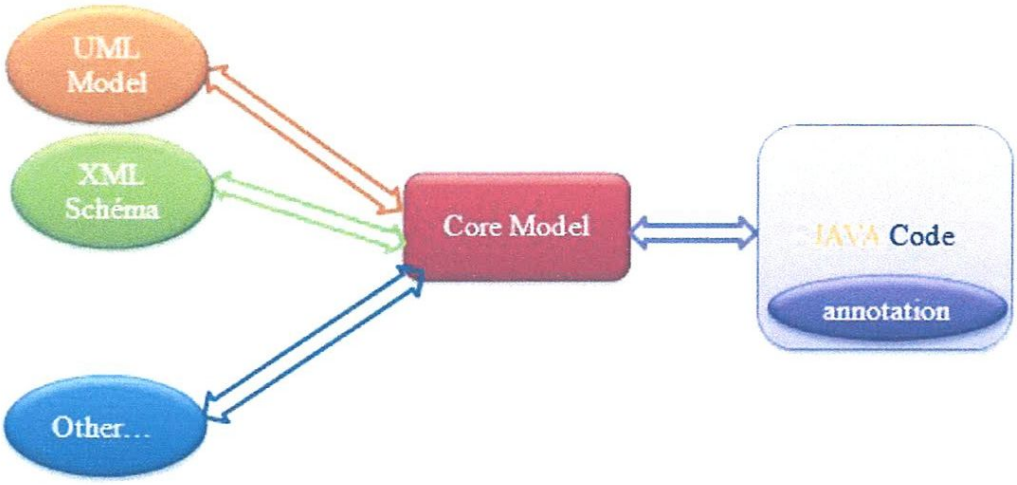

FIGURE 2.4 – Organisation d'EMF

du site d'EMF : "Le projet EMF est une plateforme de modélisation et de génération de code pour les outils et autres applications basées sur un modèle de données structurées. A partir d'une spécification du modèle décrit dans XMI , EMF fournit des outils et support d'exécution pour produire un ensemble de classes Java pour le modèle, avec un ensemble de classes d'adaptation qui permettent la visualisation et l'édition basée sur la commande du modèle , et un éditeur de base".[htt16] En d'autres termes, EMF rend possible de définir un méta-modèle et de générer les interfaces dédiées à ce méta-modèle afin de pouvoir manipuler les instances du méta-modèle dans Eclipse.[ecl16]

2.3.1 Objectifs

EMF a pour objectif, de proposer un outillage qui permet d'obtenir automatiquement le code Java en passant par le modèle. Pour cela le Framework s'articule autour d'un modèle (le Core Model, (voir fig 2.4). EMF propose plusieurs services :

- la transformation des modèles d'entrées, présentés sous diverse formes, en Core Model.
- la gestion de la persistance du Core Model
- la transformation du Core Model en code Java.[SF14]

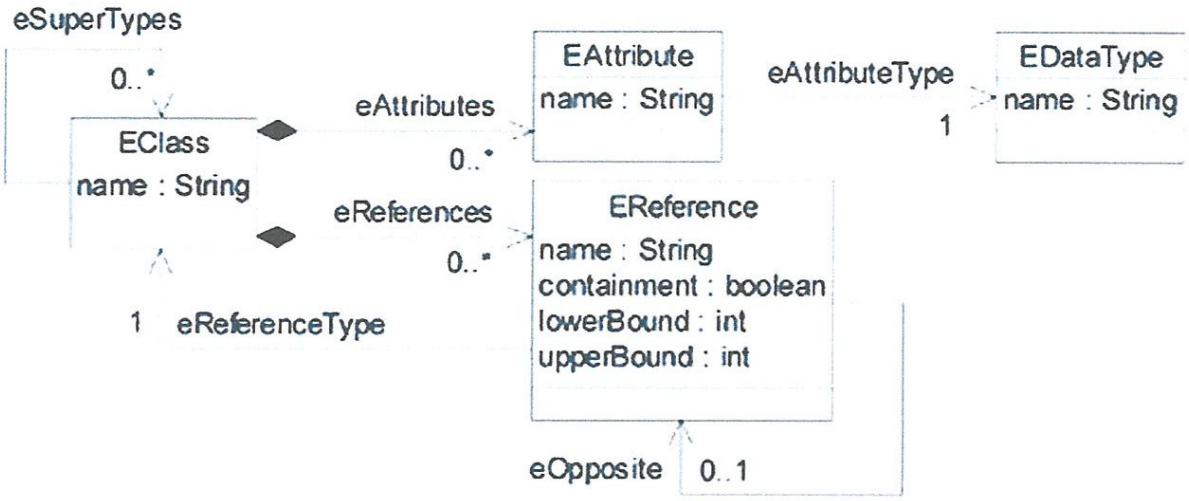

FIGURE 2.5 – Noyau d'Ecore

2.4 Architectures EMF

2.4.1 Ecore

Le modèle écore est considéré comme la définition des concepts de modélisation pour construire des diagrammes de classes (méta-modèle). IL se caracterise par :

2.4.1.1 Noyau Ecore (The Ecore Kernel)

Le plus souvent on utilise le noyau écore (voir fig 2.5) pour la description du processus par lequel le système démarre, lequel dont le chargement d'une petite portion du système est généralement impliqué pour endurer le chargement et initialisation de son reste. Ce modèle defini quatre types d'objets c'est-à-dire quatre classes qui sont :

EClass permet de modéliser des classes elles-mêmes. Les classes sont identifiées par nom et peuvent avoir plusieurs attributs et références. Pour supporter l'héritage, une classe peut faire référence à plusieurs classes (classes différentes) comme ses super-types.

EAttribute modélise des attributs ainsi que les composants des données d'un objet qui sont identifiés par nom et contiennent un type.

EDataType utilisé pour représenter des types simples dont les détails ne sont pas modélisés comme classes. Ils sont complètement associés avec un primitif ou type de l'objet défini en Java. L'identification des types de donnée est faite par nom et on utilise ces types de donnée comme les types d'attributs.

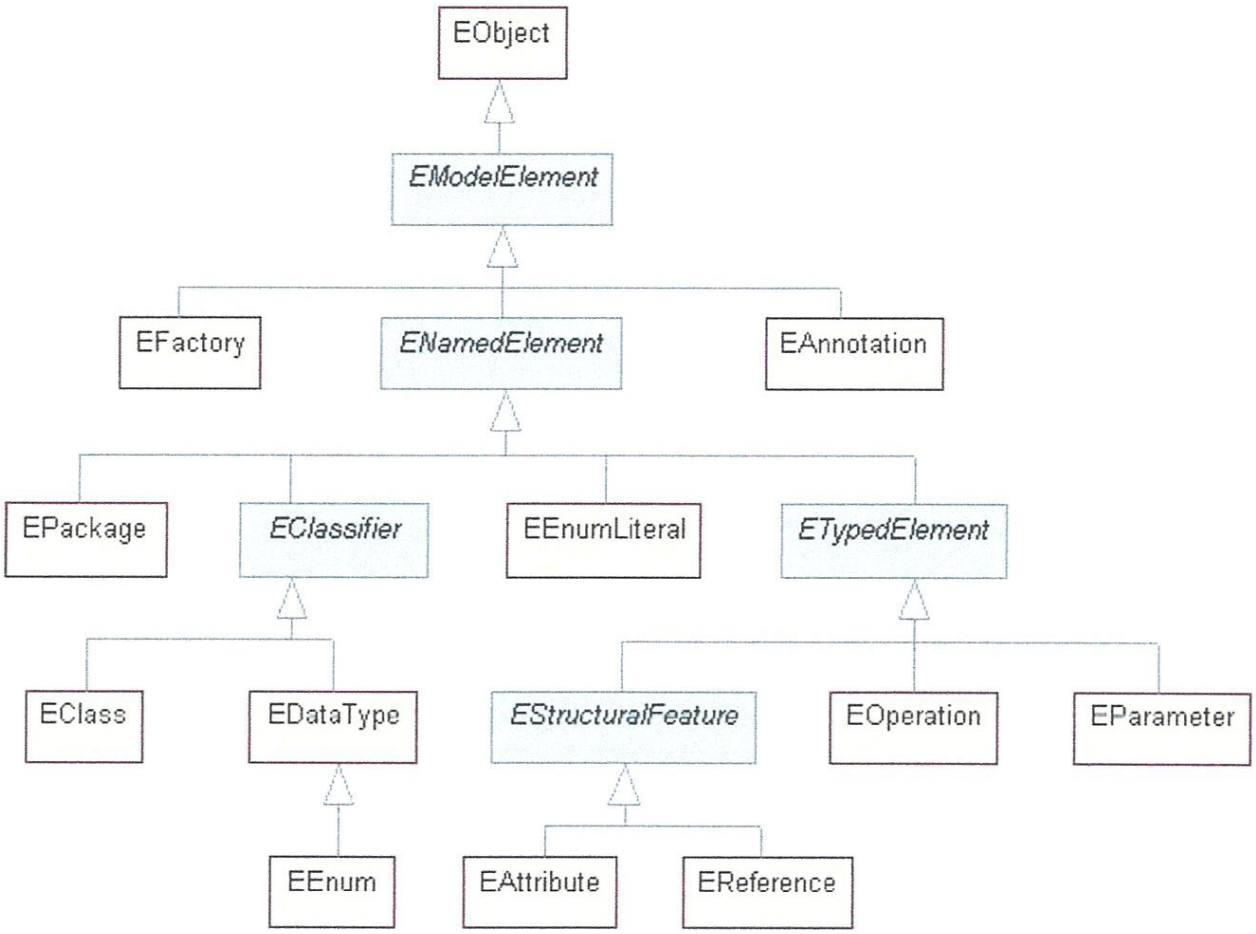

FIGURE 2.6 – Méta-modèle d'Ecore

ERefernce utilisé pour modéliser des associations entre classes ; il modélise l'extrémité d'une association. Comme attributs, les références sont identifiées par nom et ont un type. Cependant, ce type doit être les EClass à l'autre extrémité de l'association. On peut avoir une autre référence correspondante dans le cas où l'association est naviguable dans la direction opposée. Une référence spécifie des liens inférieurs et supérieurs sur sa multiplicité. Finalement, une référence peut être utilisée par représentation d'un type d'association plus fort appelé freinage ; la référence spécifie de mettre en vigueur la sémantique du freinage.[DS08]

On tient à rajouter qu'EMF s'appuie sur le méta-modèle Ecore (voir fig 2.6) qui respecte les principes définis par le MOF[8] qui est un standard OMG[9].

8. extended Meta Object Facility
9. Object Management Group

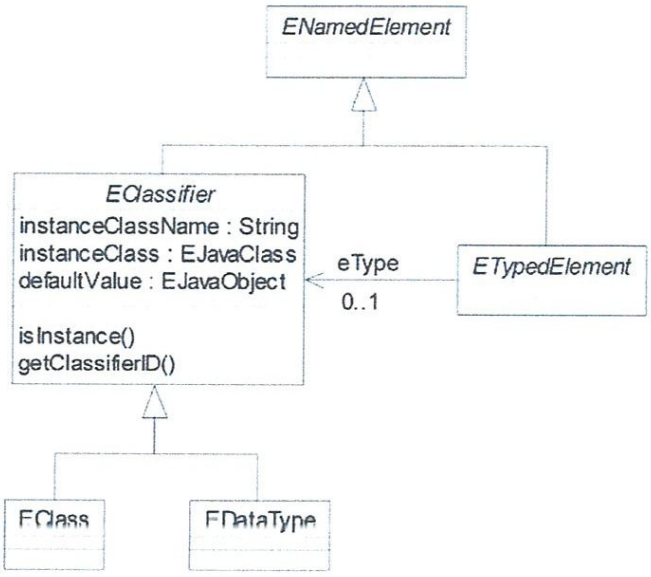

FIGURE 2.7 – Classifier d'Ecore

2.4.1.2 EClassifier

EClassifier hérite un attribut du nom d'ENamedElement, super classe de EClass et de EDataType comme illustré dans la figure 2.7, EClassifier à plusieurs sous-types (attributs et opérations) dont nous discuterons maintenant.[Mer08]

2.4.1.3 Packages et Factory

Factory Cette interface comprend une méthode create pour chacune des classes du modèle d'entrée. Cela va permettre de créer des instances (des objets) des classes de l'application. Le modèle de programmation EMF incite fortement à utiliser ces méthodes pour créer les objets lors de l'utilisation de l'application, en lieu et place de l'opérateur new.

Packages Cette classe apporte des facilités pour accéder aux métadonnées Ecore du modèle. Il contient des accésseurs aux EClasses, EAttribute,EReferences implémentées dans le modèle.[Dar08]

2.4.1.4 Annotations

Les annotations constituent un mécanisme par lequel une information supplémentaire peut être attachée à tout objet dans un modèle Ecore. La figure 2.8 modélise les Annotations.

En regardant la figure 2.8, on voit qu'avec la sous-classe EModelElement, il y a une inclusion d'une référence de l'eAnnotation qui peut contenir zéro ou plus d'EAnnotations

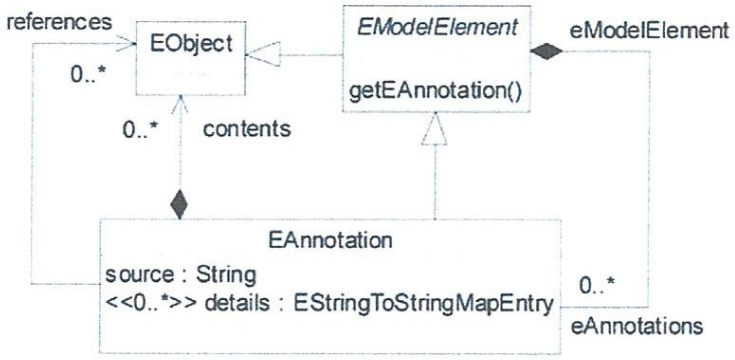

FIGURE 2.8 – Annotation d'Ecore

pour toutes les classes définies dans Ecore. Vu que cette association est bidirectionnelle (à double sens) alors par la référence opposée, un EAnnotation peut accéder à son EModelElement. EAnnotation définit typiquement un attribut source qui est utilisé pour entreposer un URI [10] qui représente le type de l'annotation. Quant à EModelElement, elle définit aussi une opération nommée getEAnnotation, et obtenir de façon commode à partir de cet attribut l'un de son EAnnotations.[Mer08]

2.4.2 Code source java correspondant aux modèles

EMF permet de générer entierement et automatiquement le code java correspondant à un modèle donné à travers trois packages :

- $< nom - meta - modele >$: contient des interfaces pour chaque entité du méta-modèle.
- $< nom - meta - modele.impl >$: mise en oeuvre concrète des interfaces définies dans $< nom - meta - modele >$.
- $< nom - meta - modele.util >$: ce package génère deux classes $< nom - meta - modele >$AdapeterFactory et $< nom - meta - modele >$Switch ces deux classes facilitent la manipulation du code générer.

2.4.3 Options des éditeurs sémi-graphiques générés

On a deux parties (Edit et Editor) génériques avec les éditeurs semi-graphiques qui contiennent chacune des options :

10. Uniform Resource Identifier

2.4.3.1 .Edit

Edit est un plugin Eclipse qui inclut des classes génériques réutilisables, permettant de générer des éditeurs pour des modèles EMF. Il fournit :

- Des classes fournisseurs de contenu avec une prise en charge des sources de propriété et d'autres classes de simplification qui permettent d'afficher des modèles EMF via des viewers [11] JFace de bureau standard et des feuilles de propriété.
- Un ensemble de commandes, incluant des classes d'implémentation de commandes génériques permettant de générer des éditeurs qui prennent en charge de manière totalement automatique des actions comme l'action annuler et refaire etc.
- Un générateur de code capable de générer tout ce qui est nécessaire à la construction d'un plug-in éditeur complet pour notre modèle EMF. Il produit un éditeur parfaitement structuré en conformité avec le style recommandé pour les éditeurs de modèles Eclipse EMF. Si on le souhaite, on peut par la suite personnaliser le code, sans perte de connexion au modèle. [edi16]

2.4.3.2 .Editor

Editor permet de générer l'interface graphique du modèle en s'appuyant sur Edit qui est considéré comme intermédiaire entre le modèle et Editor.

2.5 Contraintes d'intégrité des modèles

Un des problèmes que nous n'avons pas encore abordé c'est comment définir un état correct d'un modèle EMF. A ce propos, la plate-forme EMF offre la possibilité de définir des contraintes sur les modèles. Une contrainte est une propriété qu'un modèle doit respecter tout au long de sa vie. Ils sont définis au niveau du méta-modèle et on a deux façons de les exprimer.

2.5.1 Expression des contraintes avec java

Une annotation ajoute des informations que nous ne pouvons pas exprimer dans le meta-modèle, et justement en EMF, une classe ou un type de donnée peut être annoté, représentant une ou plusieurs contraintes.
Par exemple supposons l'existence d'une contrainte évitant d'avoir des règles de grammaire récursive (voir chap 1), alors l'annotation suivante devra être ajouter à l'interface de la classe Rule.

11. Elements d'IHM qui permettent d'afficher un contenu dans l'environnement éclipse

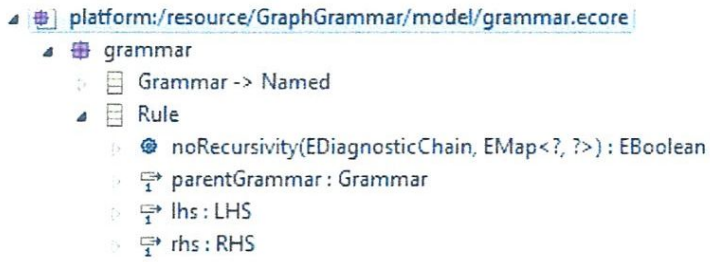

FIGURE 2.9 – Méthode NoReursivity dans le Modèle Ecore(grammar)

```
/**
 * @model annotation="http://www.eclipse.org/emf/2002/Ecore
constraints='NoRecursivity' "
 */
public interface Rule
{
  ...
}
```

Pour un accès simple à partir du code manupilant un objet avec contrainte, EMF propose de répresenter les contraintes comme des méthode de classe. Evidenment, cette méthode a des propriétés particulières :

— il retourne un EBoolean qui indique si la condition est satisfaite ou pas.

— il a aussi deux paramètres qui sont de type EDiagnosticChain et EMap.

Il est aussi possible de définir directement les contraintes dans le modèle Ecore(voir Fig.2.9) si on le souhaite. Le code genéré par la suite doit être modifier selon les bésoins du modèle, EMF offre toute une panoplie de possibilité d'amélioration du rendue par exemple comment et quoi afficher lors d'un échec de validation d'un modèle, on peut aussi jouer sur le degré d'une contrainte (simple, fatal, sévère ...) En révenant à notre exemple de non récursivité, même si la contrainte n'est pas vraiment neccessaire ci-dessous à titre d'exemple le code correspondant avec description.

```
public boolean noRecursivity(DiagnosticChain dc, Map<?, ?> e) {
    // TODO: implement this method
    //  > specify the condition that violates the invariant
    // -> verify the details of the diagnostic, including severity and
       message
    // Ensure that you remove @generated or mark it @generated NOT
    boolean valid=true; //boolean à retourner
    String lhsname=this.lhs.getNode().getName(); // obténir le LHS de la
       règle
```

```
    Iterator<Node> i = this.getRhs().getSubGraph().getNodes().iterator();
    while(i.hasNext()) //parcours de la partie droite
    {
        String rhsname=i.next().getName();
       if(lhsname.equals(rhsname)){valid=false;break;} //test de
           correspondance
    }
    if (!valid) {
       if (dc != null) {
          dc.add
             (new BasicDiagnostic
                (Diagnostic.ERROR,
                 GrammarValidator.DIAGNOSTIC_SOURCE,
                 GrammarValidator.RULE__NO_RECURSIVITY,
                 EcorePlugin.INSTANCE.getString("_UI_GenericInvariant_diagnostic",
                     new Object[] { "noRecursivity", "" }),
                 new Object [] { this }));
       }
       return valid; //contrainte non satisfaite
    }
    return valid; // satisfaite, pas d'occurence du lhs dans le rhs
}
```

2.5.2 Expression des contraintes avec OCL

Tout d'abord OCL (Object Constraint Language) est un langage d'abstraction de niveau élevé parfaitement intégré à UML, qui permet de préciser les modèles UML tout en faisant en sorte qu'ils soient toujours indépendants des plates-formes de programmation et de spécifier des contraintes sur les modèles UML 2.0. Grâce à ces contraintes, on peut se permettre de spécifier le corps des opérations des classes. [SF14]
OCL contient des expressions qui ne génèrent aucun effet de bord [12]. En parlant d'expression, il est à savoir qu'une expression est rattachée à un contexte qui représente l'élément du modèle UML. La syntaxe pour la représentation d'une expression OCL est :

— *context monContexte*
— *<stéréotype> nomContrainte : Expression de la contrainte*

Avec l'expression de contrainte OCL, il est à connaître que :

12. Modification indirecte de la valeur d'une variable

— Toute contrainte OCL est liée à un contexte spécifique, l'élément auquel la contrainte est attachée.
— Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clé **self**.

2.6 Comparaison des modèles

Avec les modèles EMF, il existe deux plateforme de comparaison et d'interrogation de modèles EMF (EMF Query et EMF Compare) qui ont des rôles différents :

2.6.1 EMF Query

C'est un outils qui permet de définir des questions explicatives sur les modèles EMF, et les exécutés éfficacement sans codage manuel i.e., il permet d'interroger un modèle, le mirroir pourait être fait avec le langage de requete SQL [13]. Sa syntaxe est textuelle Xtext, EMF Query fournit le support pour l'évaluation rapide des questions du modèle actuellement développées. Pour rendre supportable l'intégration de ces questions dans les applications existantes d'EMF, il faut que les scénarios d'intégrations aient un code productible, tel que la validation rapide du modèle, définition de l'interface de l'utilisateur du modèle ou définition de haut niveau pour les modèles Ecore.

2.6.2 EMF Compare

EMF Compare permet de fournir une comparaison pour tout genre du Modèle EMF et de la rendre facile. Il est integré dans l'environnement EMF.

2.7 Persistance des modèles EMF

Le framework d'EMF qui assure la persistance(sérialisation XMI avec résolution des références) est centré sur quatre(4) interfaces fondamentaux :Ressource, RessourceSet, Ressource.Factory, and URIConverter.[Mer08]

2.7.1 URIConverter

Tout d'abord, un URI est une chaîne de caractère composée de trois parties fondamentales :

1. scheme : identifie le protocole utilisé pour accéder à une ressource, les plus communs sont "file" pour identifier directement la location des fichiers, et "jar" pour

13. Structured Query Language : langage de requête structurée

les archives contenant des fichiers jar. Dans eclipse c'est "platform" qui est utilisé pour l'identification de ressources spécifique dans eclipse comme dans les plu-gins externes en cours d'exécution.

2. scheme-specific : toutes les parties incluent entrent scheme et fragment, son format et interpretation depend entièrement de scheme.
3. fragment (facultatif) : sert à identifier une partie de la ressource specifiée par scheme et scheme-specific, il est separé du reste de l'URI par un #. Par exemple l'URI *file* : */E* : */bara/runtime – EclipseApplication/graph.xml#loc* va identifier quelques choses dans le fichier graph.xml qui sera localisé en utilisant le fragment loc.

l'URIConverter est utilisé par le framework pour normaliser les URIs, donc il sert à convertir un URI en un autre URI qui sera utilisé pour acceder à une ressource.
Par exemple si un EPackage avec l'URI http ://www.example.com/epo2.ecore est serialisé dans le workspace comme /models/epo2.ecore par conversion, ceci permettra au framework de lire automatiquement dans ce méta-modèle comme bon lui semble, lorsqu'il charge une instance de ce modèle.[DS08]

2.7.2 RessourceSet et Ressource.Factory

Contexte commun pour différentes ressources se référençant, Factory quand à lui Définis comment les ressources sont crées(plus autres utilitaires). Dans ce qui suit un fragment de code montrant l'utilisation de l'objet ResourceSet.

— ResourceSet rs – new ResourceSetImpl() ;
— URI uri = URI.createFileURI("/data/example.xmi") ;
— Resource resource = rs.createResource(uri) ;

2.7.3 Ressource

En effet, la persistance dans EMF repose sur Ressource, simplement une ressource est un conteneur pour un ou plusieurs objets qui doivent être persistés(save() et load()) ensemble, avec leurs contenus.
On accede au contenu via getCoontents().

2.7.3.1 Sauvegarde

La sauvegarde consiste à la création de l'URI, puis celle de la ressource et ensuite il ne reste plus qu'à mettre à jour le modèle.
URI uriGraph =URI.createFileURI("</data/Mygraph.graph">) ;
Resource r-graph = resourceSet.createResource(uriGraph) ;
r-graph.getContents().add(node) ;

```
try {
r-graph.save(null) ;
}
catch(IOException e) {
e.printStackTrace() ;
}
```

2.7.3.2 Chargement

En plusieurs étapes, elle consiste aussi à la création de l'URI, puis celle de la ressource et ensuite le chargement du modèle qui ne doit pas être vide.

```
URI uriGraph = URI.createFileURI("</data/Mygraph.graph">) ;
Resource r-graph = resourceSet.createResource(uriGraph) ;
try {
r-graph.load(null) ;
}
catch(IOException e) {
e.printStackTrace() ;
}
if (resource.isLoaded() and resource.getContents().size() > 0) {
system = (structure.System) resource.getContents().get(0) ;
} [Gai07]
```

2.8 Modéliation graphique

2.8.1 Syntaxe abstraites et concrètes

Les modèles spécifient différents niveaux d'abstraction, facilitant la gestion de la complexité inhérente aux applications.

2.8.1.1 Abstraites

La syntaxe abstraite permet d'obtenir une structuration pour les données d'entrées, en suggérant au passage la manière de stocker ces données en mémoire[sss16]

2.8.1.2 Concrètes

Une syntaxe concrète graphique permet de fournir un moyen de pouvoir visualiser et/ou éditer un modèle de façon agréable et efficace. Pour la réalisation de cela, nous utiliserons des outils développées avec l'outil GMF (Eclipse Graphical Modeling Framework). Ils

permettent de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse. GMF assure le développement d'éditeur graphique Ecore utilisé pour visualiser et éditer les modèles Ecore (les méta-modèles).[416]

2.8.2 La plateforme GEF

GEF pour Graphical Editing Framework, est un environnement pour le développement des éditeurs visuels graphiques avec abstraction totale du Modèle mais avec possibilité de le modifier via une infrastructure qui organise une série d'implémentations du pattern 'Command'. Cette infrastructure est aussi utilisée pour le fonctionnement interne de GEF.

2.8.3 La plateforme GMF

GMF pour Graphical Modeling Framework,est venu historiquement après GEF et EMF pour relier les deux plateformes. C'est un Framework de l'environnement de travail Eclipse. Il permet de représenter des modèles de manière graphique c'est-à-dire sous la forme d'un graphe composé de noeuds et d'arcs et fournit une infrastructure permettant l'exécution d'éditeurs graphiques basés sur les Framework EMF et GEF. [eK11]

2.8.4 La plateforme Sirius

C'est une technologie permettant de saisir et de visualiser tout type d'information de façon graphique et personnalisée.
Sirius permet aussi de modéliser des informations et de les représenter de façon très visuelle et il est le résultat d'une collaboration initiée, lorsque Thales a souhaité se doter d'un atelier de modélisation pour supporter le processus d'ingénierie de système du groupe.
Ce dernier a fait appel à la société Obeo pour développer une technologie de modélisation par point de vue, intégrant des contraintes fortes en termes d'ergonomie, de performance, de robustesse et de capacité de personnalisation. Thales et Obeo ont mis en place plusieurs objectifs pour la mise en place de Sirius :

- Renforcer l'écosystème Eclipse Modeling en mettant à disposition une technologie de modélisation mature et de premier plan.
- Inscrire Sirius dans une dynamique de pérennité sur plusieurs décennies, afin de supporter des usages stratégiques chez certaines grandes entreprises.
- Multiplier les collaborations avec des partenaires dans un environnement ouvert et facile d'accès.

La version 0.9 est la première version de Sirius faite en Novembre 2013, elle offre la possibilité de définir des types arbres, tables et diagrammes comme éditeurs, elle a été suivie

par la version 1.0 (stabilisée) en juin 2014. Sirius sera d'ailleurs par la suite la technologie que nous rétiendrons pour les manipulations visuelles sur nos modèles.[All13]

2.9 Conclusion

Il y'a à rétenir :

1. EMF est une plateforme de modeisation qui concretise l'approche MDA.
2. EMF permet de prendre en entrée plusieurs formats de modèles (UML, Java Annoté, Schéma XML, XMI) : il peut donc être compatibles avec d'autres outils utilisés en amont dans le cycle de vie du logiciel.
3. une fois que l'on dispose du modèle d'entrée, les transformations se passent très facilement : quelques clics suffisent.
4. le code généré sera aussi précis et complet que le sera le modèle d'entrée : EMF pourra donc s'adapter à l'évolution des recherches (et donc des outils) qui fabriquerons les modèles du futur.
5. la génération du code à partir du modèle n'empêche pas son raffinage manuel : une régénération après une modification du modèle n'écrase pas les compléments.

Dans le prochain chapitre nous parlerons de l'environnement Eclipse qui est le conteneur d'EMF.

CHAPITRE 3

Environnement Eclipse

Plan du chapitre

3.1 Introduction

L'objectif principal de ce chapitre est l'étude de la plateforme Eclipse. Eclipse est un noyau qui sert uniquement comme une base sur laquelle on peut implanter des composants

logiciels pour offrir une variété des services allant des simples éditeurs de texte jusqu'aux environnements de développement compliqués.
Dans le jargon d'Eclipse, les composants logiciels s'appellent des plugins, Cette appellation reflète la faculté de ces derniers d'exploiter les fonctionnalités des autres plugins et leurs capacités à exposer leurs services aussi aux autres plugins. Nous pouvons constater très rapidement que la plateforme éclipse sans aucun plugin ne sert à rien pour l'utilisateur final.

3.2 Un peu d'histoire

La première version Open source d'Eclipse a été lancé en novembre 2001 par IBM [1], Borland, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft et Webgain. Eclipse est essentiellement une réécriture du logiciel VA4Java [2] en langage Java. Dès l'origine du projet, ils voulaient offrir une solution multiplateforme, pouvant être exécutée sur les différents systèmes d'exploitation de ses clients. De même le projet s'est voulu extensible par le biais de plugins. Actuellement le projet Eclipse est géré par un consortium incluant un représentant de chaque firme participant dans le projet. Le consortium détermine les objectifs du projet et veille à élaborer une communauté Eclipse qui incarne l'esprit de l'open-source. Le projet Eclipse est chapeauté par « theProject Management Committee PMC » et divisée en trois (03) sous projets :

1. La plateforme,
2. The Java développement KIT (JDT) l'outil de développement java,
3. The plug-in developpment Kit (PDT) l'outil de développement des plugins.

Chaque sous projet est à son tour divisé en sous sous projet, à titre d'exemple le sous projet PDT contient deux sous projet UI et Core. La contribution dans le projet Eclipse n'est pas limité aux membres du consortium, mais il est possible à tout individu ou entreprise à contribuer dans le projet.[wik16]

3.3 Architecture d'Eclipse

En plus du noyau « platform runtime », la plateforme Eclipse contient les composant suivants the WorkBench, WorkSpace, Help et Team (voir Fig.3.1) très connu des utilisa-

1. International Business Machines
2. Visual Age for java : est un logiciel développé par l'organisme OTI qui est spécialisé dans le développement des outils pour la programmation orientée objet, OTI qui a été racheté par IBM en 1996, était derrière le fameux produit visual age d'IBM reconnu à l'époque comme standard dans la programmation orienté objet en Smalltalk.

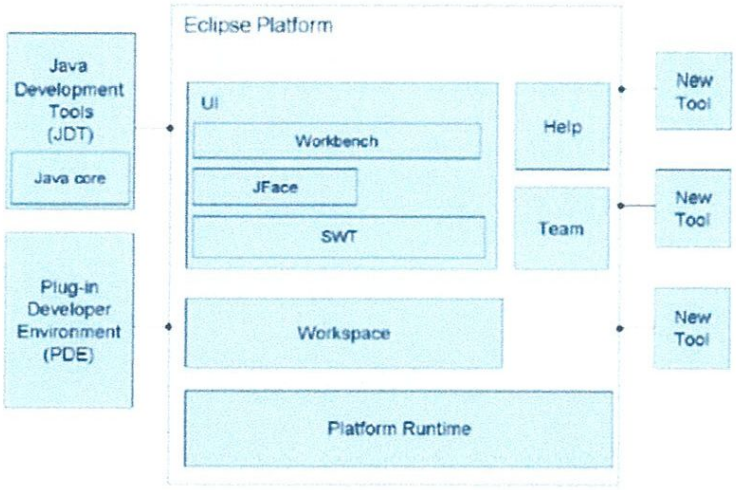

FIGURE 3.1 – Architecture d'Eclipse

teurs d'éclipse,tous les autres outils s'intègrent à la plateforme Eclipse. Dans ce qui suit nous donnons un appérçu sur chaque composant de la plateforme :

3.3.1 Platform Runtime

Le premier rôle du « plateform runtime » est de découvrir quels sont les plugins disponibles. Chaque plugin contient un document [3] XML qui décrit les services requis et offerts par ce dernier.
Vu leurs nombre très grands les plugins ne se chargent pas au démarrage mais au moment où ils sont invoqués, ce mécanisme [4] permet d'économiser l'espace mémoire occupé et de réduire la latence du démarrage du plateforme.

3.3.2 Le workbench

Décrire au détail près le workbench néccessiterait plus d'un livre ce n'est donc pas envisageable dans le cadre de notre projet, workbench est l'interface graphqiue d'eclipse, en plus d'afficher les menus et le bar d'outils familiers, il est organisé dans ce que nous appelons des perspective et qui contiennent à leurs tour des vues (views) et des editeurs (Editors).
Le workbench est construit avec la bibliothèque SWT et JFace qui contrairement aux bibliothèques standards de Java AWT et SWING utilisent les API graphiques natives du système d'exploitation en cours d'exécution It is possible to use SWT and JFace to create your own native-looking Java applications.

3. XML Manifest file
4. Plug-in Lazzy Loading

3.3.3 Le Workspace

le Workspace est responsable d'organiser les ressources de l'utilisateur final, les ressources sont répertoriées dans des projets, chaque projet correspond à un sous répertoire dans un répertoire nommé Workspace. Le Workspace mantient une historique des changements des ressources, il est responsable aussi de notifier les composants intéressés [5] sur les changements qu'ont subis certaines ressources. Les ressources peuvent porter une nature « java project » par exemple.

3.3.4 Support de travail en équipe

Eclipse fournit un outil pour travailler avec Concurrent Versions System (CVS), il n'y a pas besoin d'installer quoi que ce soit. Il suffit de se connecter à son serveur CVS. Ensuite, Eclipse fournit toutes les commandes propres à CVS. Les créateurs même d'Eclipse utilisent CVS pour le développement d'Eclipse

3.3.5 Aide

Comme Eclipse lui-même, le système d'aide d'Eclipse est extensible, des plugins peuvent contribuer avec des documentations en format HTML, XML, définir des schémas de navigation, expliquer les relations entre les plugins, etc.

3.4 Plugins Eclipse

Comme nous l'avons déjà expliqué au début de ce chapitre, le plugin est le plus petit composant extensible dans Eclipse. Il contient du code et/ou des ressources. La plateforme Eclipse contient plus de 100 plugins en synergie. Les frontières entres les plugins permettant une connexion entre ces derniers s'appellent les points d'extension. Un point d'extension est le mécanisme par lequel un plugin peut ajouter des fonctionnalités à un autre. Le couplage faible entres les plugins c'est l'une des caractéristiques les plus fortes d'Eclipse. Les concepteurs de n'importe quel outils ne sont jamais contraints à fournir des clés en main dans un seul coup « all in one » tools, mais plustôt, ils ont la capacité de sous traiter certains des fonctionnalités à d'autres plugins, Conceptuellement parlant, les plugins ont une structure très simple(voir Fig.3.2). Dans le répertoire d'installation d'Eclipse il y'a un sous répertoire Plugins contenant l'ensemble des plugins. Chaque plugin est contenu dans un sous répertoire de ce dernier et porte le même nom que le plugin du répertoire plugins. Dans chaque répertoire de plugin, il existe obligatoirement un fichier plugin.xml

5. Nous parlons du patterns Observer

```
C:\ECLIPSE
|
+---features
|
+---plugins
|       +---org.apache.ant_1.5.2
|       |       lib\
|       |       .options
|       |       about.html
|       |       antsupport.jar
|       |       plugin.properties
|       |       plugin.xml
|       |
|       +---org.eclipse.ant.core_2.1.0
|       |
|       ...
+---workspace
```

FIGURE 3.2 – structure d'un plugin (cas de plugin org.appachc.ant)

avec d'autres fichiers optionnels. Plugin.xml s'appelle fichier **manifest** et contient toutes les informations sur un plugin comme le nom, la version, les bibliothèques requises, les point d'extensions utilisé pour offrir les services du plugin et ainsi de suite.[Rubnc] Les fichiers et répertoire que nous trouvons dans un plugin sont :

Plugin.xml Plug-in manifest spécifie les extensions que le plug-in étend et les points d'extension que d'autres plug-ins pourront à leur tour étendre.Voir un exemple à la figure 3.3, les informations contenues dans ce fichier sont utilisées entre autres pour informer les autres plug-ins des extensions disponibles.

build.properties Garde toutes les informations concernant les fichiers, les dossiers,

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.0"?>
3
4  <plugin>
5
6     <extension point="org.eclipse.emf.ecore.generated_package">
7        <!-- @generated graph -->
8        <package
9              uri="http://graph/1.0"
10             class="graph.GraphPackage"
11             genModel="model/graph.genmodel"/>
12    </extension>
13
14
15 <extension point="org.eclipse.emf.emp.view.model.provider.xmi.file">
16 <file filePath="model/Graph.view"/>
17 </extension>
18 </plugin>
```

FIGURE 3.3 – Exemple d'un fichier plugin.xml(cas de plugin de GraphModel)

les librairies, etc.. qui seront par la suite insérées dans le ficchier JAR du plug-in, car ceux-ci lui sont nécessaires pour son fonctionnement.

manifest.mf Spécifie le nom du plug-in(Bundle-name), la version du plug-in (Bundle-version), le nom de la classe qui s'occupe du cycle de vie du plug-in(Bundle-Activator), les dépendances du plug-in par rapport aux autres plug-ins(Require-Bundle), etc. Ces informations contenues dans le manifeste permettent au plug-in d'être chargé au lancement de la plateforme.

about.html une location standard pour les informations concernant les licences.

***.jar** Tout code java nécessaire pour le plugin.

Répertoire lib pour plus d'archives JAR.

Répertoire des icônes Le répertoire des icônes est pour les icones utilisées dans le répertoire le plus souvent en format gif.

Autres Des fichiers selon le besoin.

[Rubnc]

3.4.1 PDE, l'environnement de développement des plugins

La création des plugins se fait à l'aide d'un plugin fourni en standard avec la distribution d'Eclipse qui s'appelle PDE « Plugin Development Environnement » Basé sur l'outillage Java. Le PDE contribue à Eclipse avec une perspective contenant des vue des éditeurs et des assistants pour la création le maintien, la publication, etc. [Rubnc]

3.4.2 Cycle de vie d'un plugin

Au démarrage d'Eclipse, le Platform runtime scanne le répertoire plugins pour recenser tous les plugins enregistrés. S'il trouve plus d'une copie pour le même plugin, celle de la version le plus récente est retenu. La liste des plugins recensés constitue ce que nous appelons le plug-in registry.
D'un point de vue technique, la Platforme runtime lit tous les fichiers manifest, mais ne charge aucun plugin à ce moment. Chaque plugin sera chargé uniquement lorsqu'il est invoqué pour la première fois.
Ce mécanisme qui s'appelle « chargement tardive » ou « plug-in lazy loading » permet à la plateforme Eclipse de supporter des milliers de plugins en gardant la même performance.

3.4.3 Création d'un nouveau plugin

Pour la création d'un nouveau plugin, on doit se procurer l'assistant(voir Fig.3.4) fourni par PDE qui offre en plus de l'opération de création plusieurs facilités pour la gé-

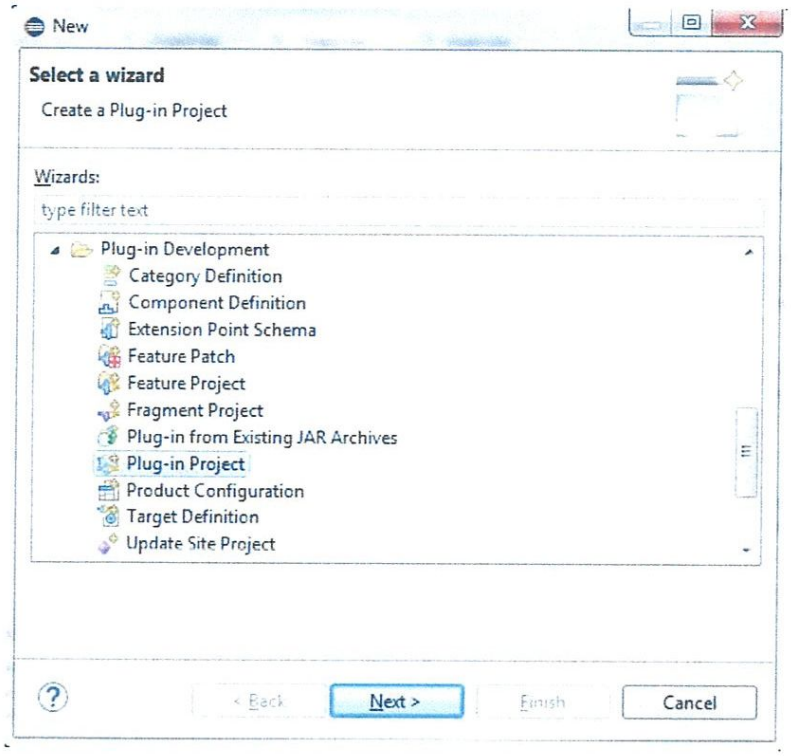

FIGURE 3.4 – Assistant de création d'un nouveau plugin

nération des composants qui contribuent à l'interface graphique de la plateforme Eclipse. Dans la première page de l'assistant, l'information importante est le nom du projet. Pour ce nom la convention est d'utiliser les mêmes règles de nommage que pour les packages. Pour la seconde page(voir Fig.3.5) de l'assistant, les valeurs par défaut sont généralement acceptables :

l'ID reprend le nom du projet, cette valeur est importante car elle est notamment utilisée par certaines API.

Version peut être librement modifiée.

Name le nom du plug-in, n'est pas une information vraiment importante, mais elle peut être parfois visible par l'utilisateur (Menu Help->About->bouton 'Plug-ins details').

Provider même remarque que pour le nom.

Classpath vide par defaut.

Options — La génération d'un 'Activator' est souhaitable, cette classe respecte une convention proposée par OSGi, elle permettera notamment de réagir à des événements liés au cycle de vie du plug-in (arrêt, démarrage, ...).

— Le fait d'indiquer que le plug-in contient une partie graphique ('This plug-in will make contributions to the UI') permet à l'assistant d'ajouter automatique-

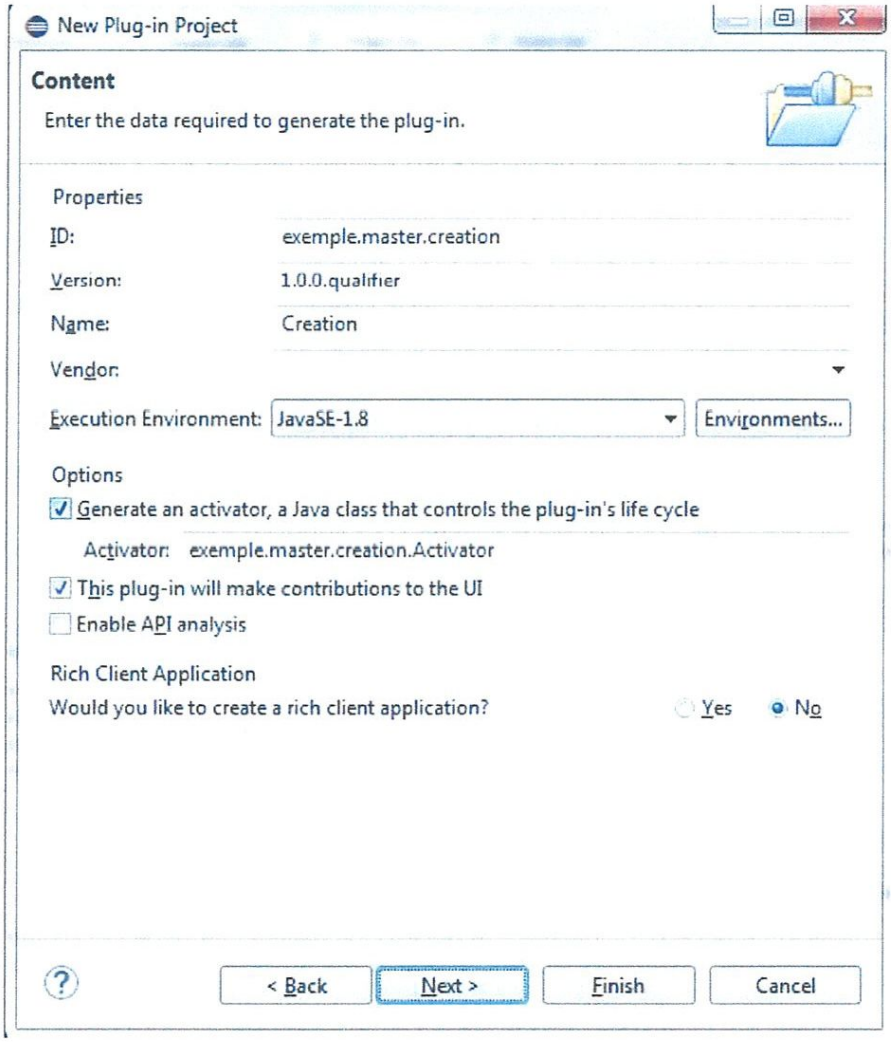

FIGURE 3.5 – Séconde page Assistant de création d'un nouveau plugin

ment les librairies graphiques d'Eclipse dans les dépendances du projet et aussi de choisir la classe dont héritera l'Activator ('org.eclipse.ui.plugin.AbstractUIPlugin' pour un plug-in graphique, 'org.eclipse.core.runtime.Plugin' sinon).

RCP Le dernier choix de cette assistant permet d'indiquer si ce plugin contiendra le point d'entrée d'une application Eclipse RCP.

Après quoi une troisième page offre la possibilité de choisir à partir de quel modèle le projet sera créé, La page suivante dépend du modèle choisi, elle permet de renseigner des informations utilisées pour générer le code du projet. Une fois le bouton 'Finish' sélectionné, Eclipse propose d'ouvrir la perspective PDE (celle-ci est très ressemblante à la perspective Java) et ouvre l'éditeur des fichiers manifestes dans la zone d'édition.

3.5 Quelques plugins usuels

3.5.1 Commands et Actions

Les commandes et les actions sont deux APIs différentent qui realisent la même chose : déclarer et implémenter des fonctions qui se concrétise comme des items de menus et des boutons de menus bars (toolbars) autrement dit si on implemente la même chose utilisant l'API Command et Action, alors le plu-gin résultant aura des menus et menus bar dupliquée.

3.5.1.1 Commands

La déclaration ou l'implémentation d'un menu ou d'un menu bar utilisant l'API command consite à déclarer une commande(répresente le concept de la fonction,eg 'coller' ; il ne définit ni l'endroit la fonction devra apparaître ni ce qui arrivera quand un utilisateur cliquera déssus), au moins un menu 'contribution' et au moins un handler (sa fonction ou comme on dit souvent son comportement) pour la dite commande (Voir Fig.3.6 et 3.7).

3.5.1.2 Actions

Une action Eclipse est composée de plusieurs parties incluant sa déclaration XML dans le plugin manifest, l'objet IAction instanciée par l'UI éclipse pour representer l'action et l'IActionDelegate contient le code correspondant à l'action(Voir Fig.3.8).

3.5.2 Views

Plusieurs plu-gins que ce soit pour ajouter un nouveau view ou d'améliorier un view existant pour fournir des informations à un utilisateur. Les views sont sous-classes de org

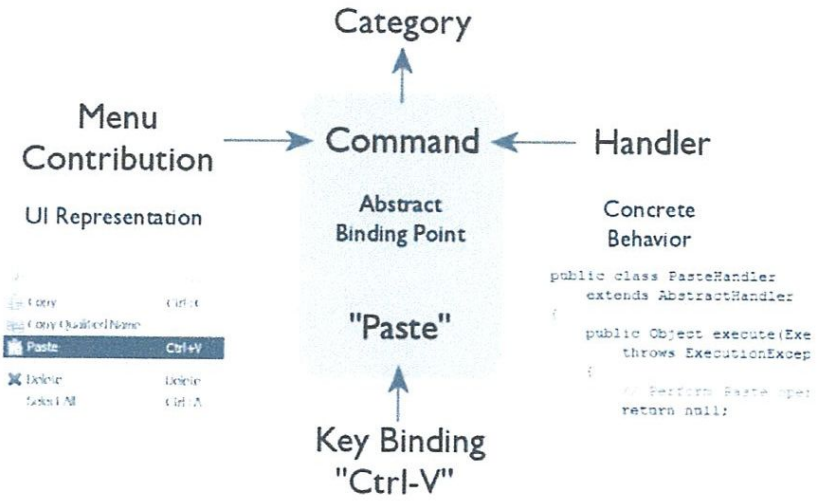

FIGURE 3.6 – API Command

eclipse ui part ViewPart donc indirectement celui de org.eclipse ui part WorkbenchPart cally, views are subclasses of org eclipse ui part ViewPart(voir Fig.3.9) Trois étapes pour la création d'un view :

1. définir une categorie de view dans le manifest.
2. définir un view dans le manifest.
3. créer la partie qui contiendra le code.

3.5.3 Editors

Utilisé comme son nom l'indique pour éditer l'état d'une ressource, constitue l'outils éssentiel pour l'utilisateur pour modifier ses ressources (e.g les fichiers). Ils sont sous-classes de org.eclipse.ui.part.EditorPart donc indirectement celui de org.eclipse.ui.part.WorkbenchPart, views are subclasses of org.eclipse.ui.part.ViewPart(voir Fig.3.10). Deux étapes à suivre pour la création : Define the editor in the plug-in manifest file Create the editor part containing the code.

1. déclaration de l'éditeur dans le manifest du plu-gin
2. créer la partie qui contiendra le code.

3.5.4 Perspectives

Les perspectives sont un moyen de reunir plusieurs views et commands éclipse pour une tâche particulière telle que le codage, pour la création org.eclipse.ui.perspectives comme point d'extension et définir sa perspective.

FIGURE 3.7 – API Command et ses extensions

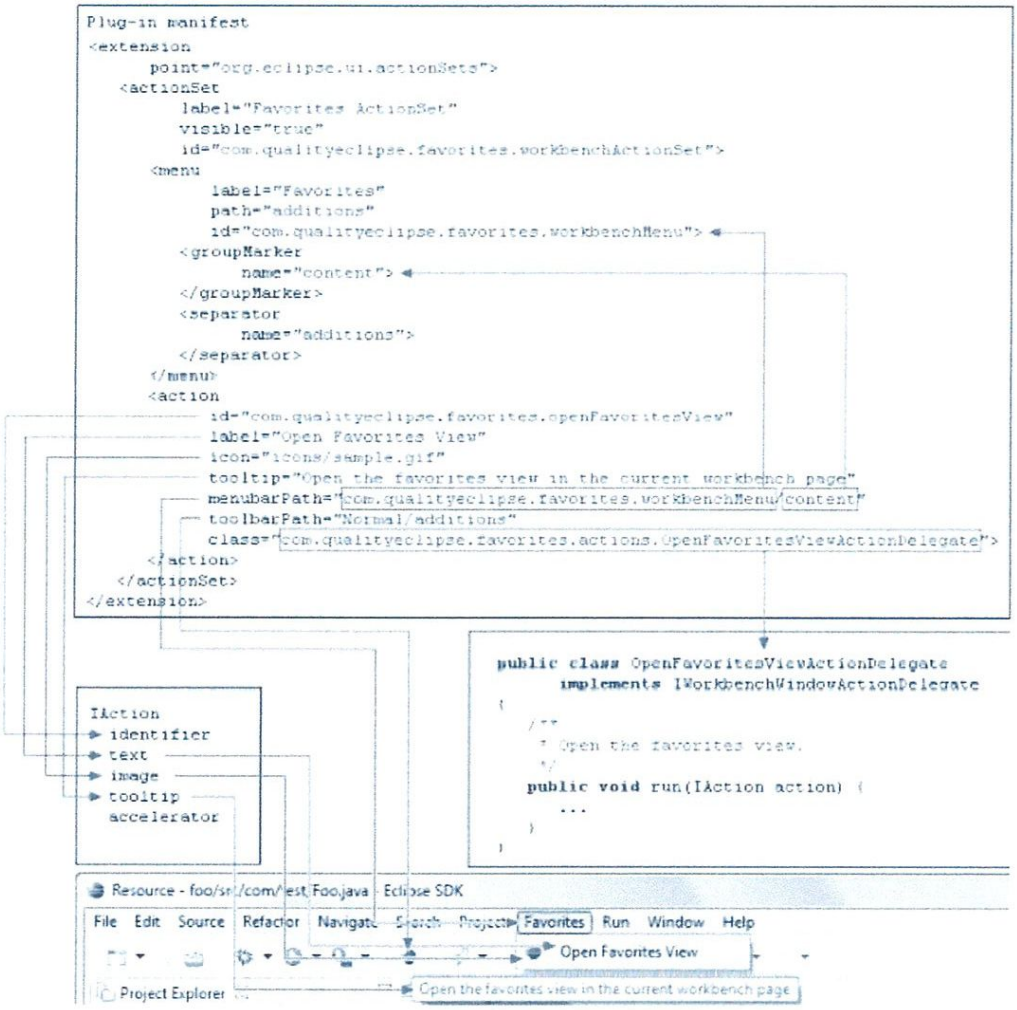

FIGURE 3.8 – API Action et ses extensions

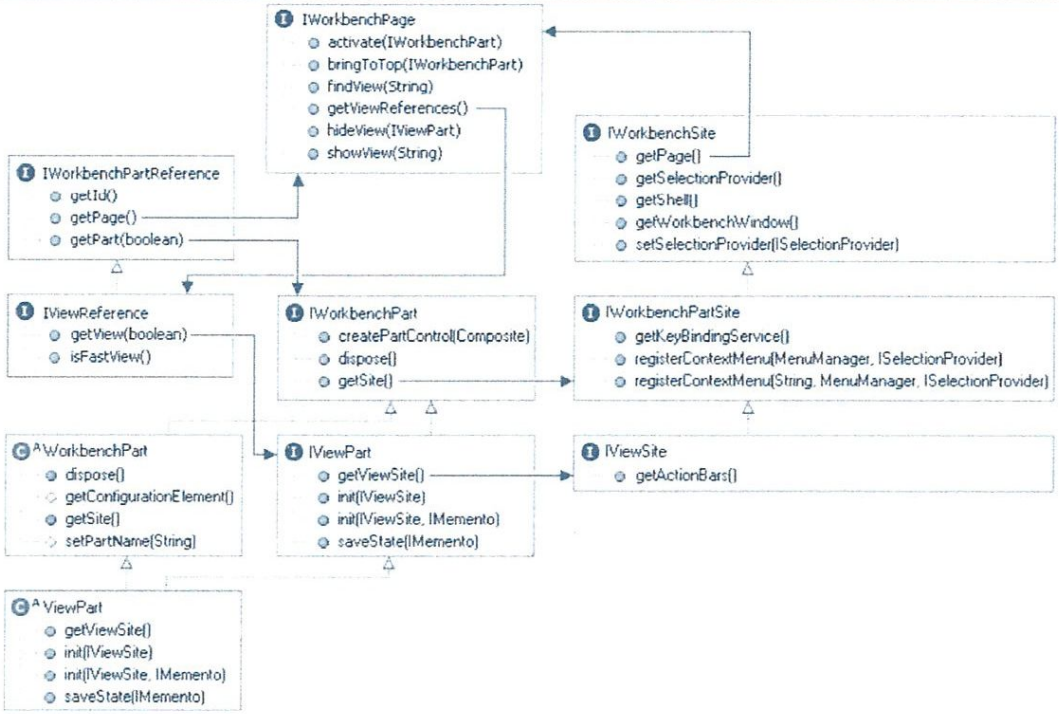

FIGURE 3.9 – Class ViewPart

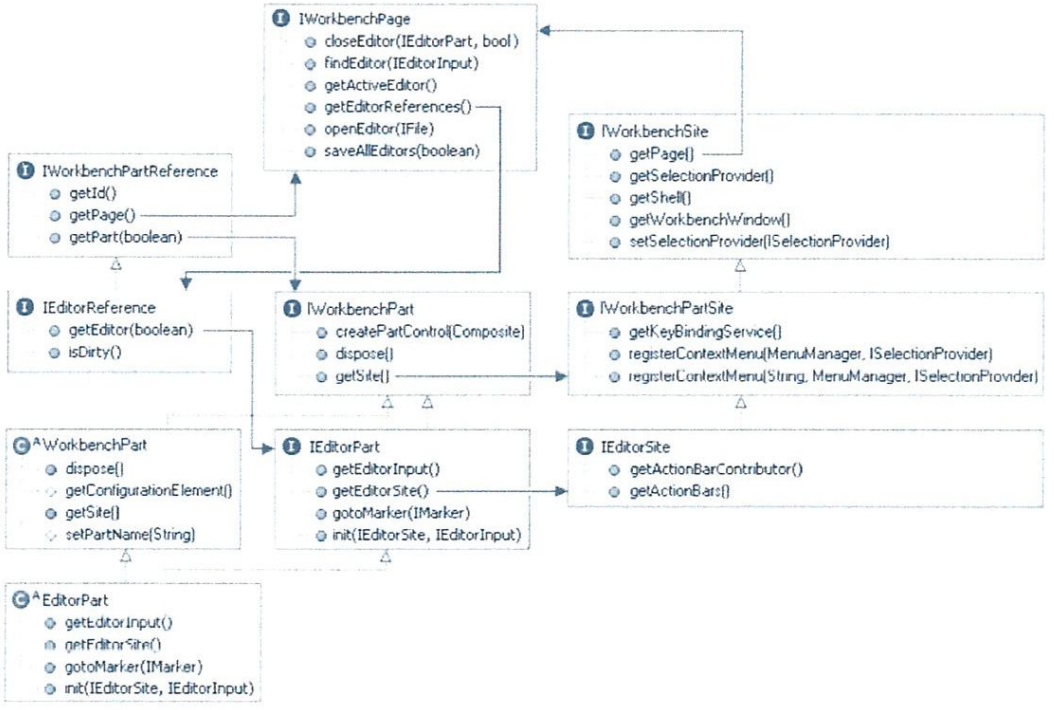

FIGURE 3.10 – Class EditorPart

FIGURE 3.11 – Hiéarchie Class Wizard

3.5.5 Wizards

Outils de présentation, d'édition qui sont parfois preferés aux editor/view, org eclipse jface wizard.WizardDialog est une spécialisation de la classe Dialog [6] (voir Fig.3.11), une illustration de la structure par défaut d'un wizzard à la figure 3.12 En outre nous avons la main sur les pages de préferences, Help, ...

3.6 Conclusion

Comme nous l'avons vu tout au long de ce chapitre qui ne se veut pas complet, il est très aisé de réaliser un programme en utilisant Eclipse. La platforme Eclipse comporte beaucoup de plu-gin prêt à être utilisé et offre l'avantage de développer tout aussi de façon rélativement simple de nouveaux plugins. Donc Basé sur une architecture modulaire permettant le développement d'un projet plugin en occurence.
Dans le prochain chapitre, nous tenterons de donner une description de la phase de conception et d'implémentation de notre plugin de transformation de modèle.

6. Class pour la manipulation de boite de dialog simple

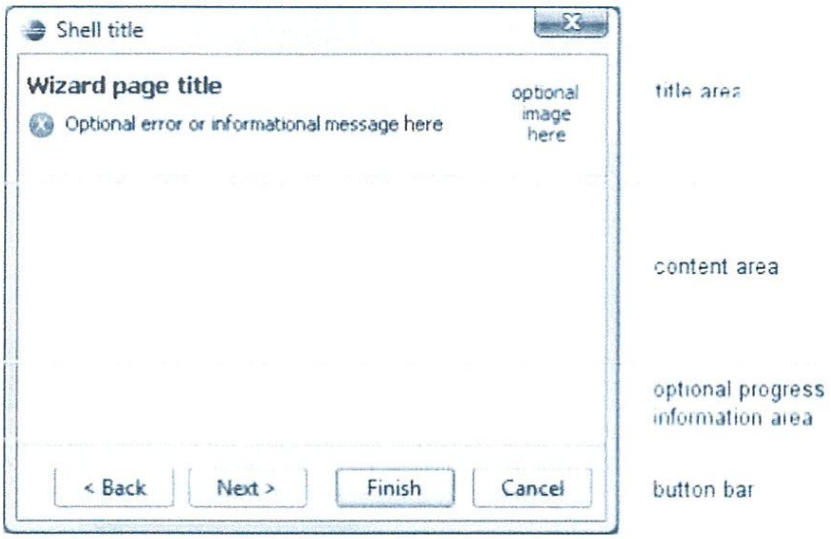

FIGURE 3.12 – Structure par défaut d'un Wizard

CHAPITRE 4

Conception et Implémentation

Plan du chapitre

4.1 Introduction

La réalisation d'un système signifie la creation d'un produit fini à partir d'un document nommé cahier de charge dans lequel on spécifie les besoins fonctionnels du système. Dans ce chapitre, nous nous intéressons à la construction de notre environnement, sera dédié donc ce chapitre à l'implémentation, répresentation des détails utilisées dans le cadre de la conception de notre plugin. La mise en place de notre système consistéra au développement d'un éditeur visuel et pour un graphe et pour une grammaire ; par la suite la mise en place d'un mécanisme qui permettra de pouvoir appliquer les règles de cette dernière sur ce graphe donné.

4.2 Conception

Dans ce qui suit, nous tentérons de faire une description exhaustive de la marche suivie lors de la mise en oeuvre de notre système de transformation de modèle EMF. Nous avons déjà présenté les différentes théories d'approche algébrique, s'appuiyant sur les grammaires pour exprimer des transformations où les parties gauches et droites des règles de production de la grammaire répresentent des modèles, et dans l'optique de notre étude qui tourne autour du MDA s'articulant sur deux concepts clés : les modèles et la transformation des modèles. Disposant d'un des outils permettant la mise en oeuvre de ces concepts, EMF dont on a parlé au paravant, il parraissait clair que le travail serait centré sur la réalisation de modèle et le comment de sa transformation ; tentons de décrire les différents points par lesquelles nous sommes passés.

4.2.1 Architecture Globale

Avoir un régard globale sur un projet s'avère être un point crucial pour sa bonne conduite car elle nous aidera à mieux 'penser' le sujet. D'un point de vue générale, la figure 4.1 répresente une vue globale de notre système de transformation.

- deux méta-modèles, de graphe et de grammaire.
- deux éditeurs, qui serviront d'écrans entre modèles et méta-modèles i.e., s'occupéront et de la création, et de la modification et de la sauvegarde de graphe et de grammaire.
- Graphe pour indiquer le modèle qui subira les transformations dont les directives sont répresentés par Grammaire.
- Transformations, étape cruciale correspondant à la mise en oeuvre concrète du passage entre théorie et pratique, de bases mathématiques vers leurs automatisations pour avoir un modèle de graphe reécrit.

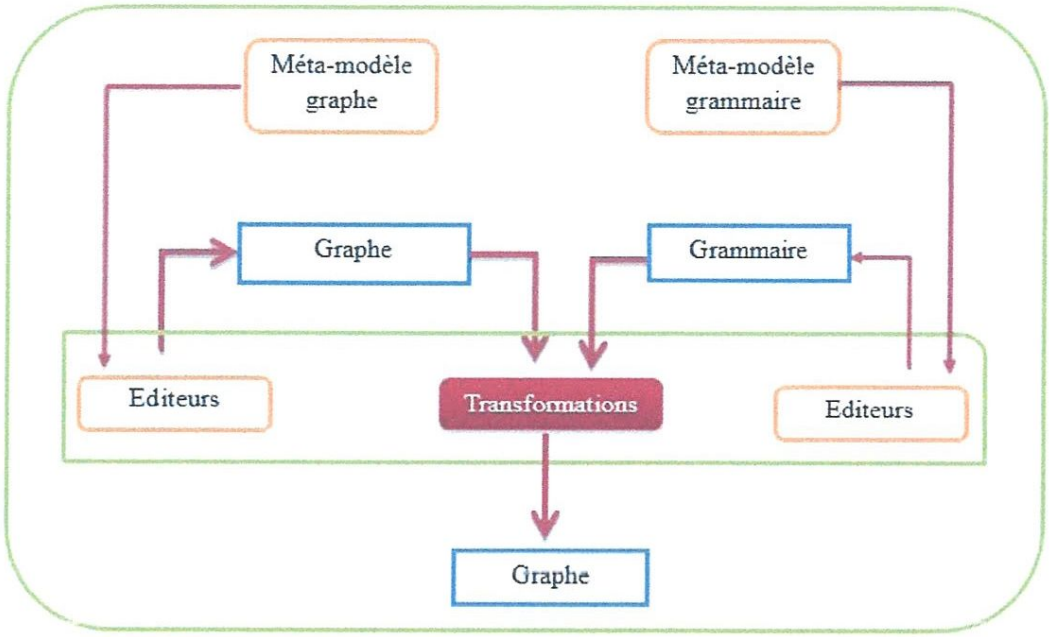

FIGURE 4.1 – Vue globale du plu-gin

4.2.2 Méta-modèle de Graphe

Comme défini precedenment, un graphe est constitué d'un ensemble de noeuds et de liens ; considérant qu'un noeud/lien a un type et qu'un type comme un graph possede un nom ; logiquement que les noueds ou liens possedent chacun un nom aussi.Pour la création d'un méta-modèle de graphe nous aurons besoin donc des entités suivante :

Named la classe qu'héritera toutes les autres, elle possede un attribut **name** de type EString.

Graph l'entité principale on peut dire, avec comme super-classe, la classe Named.

Typed classe répresentant le type d'un noeud ou d'un lien donné dont elle est la super-classe, hérite aussi de Named.

Node la classe répresentant un noeud, relié à Graph par une relation de composition et pouvant avoir un ou plusieurs **ingoingEdge** (lien rentrant vers le noeud) et **outgoingEdge** (lien sortant du noeud)

Edge la classe répresentant un lien, relié à Graph par un une relation de composition et pouvant avoir qu'une et un seule **Target**(destination) et qu'une et une seule **Source**.

Voir Fig.4.2 répresentant notre méta-modèle de graphe.

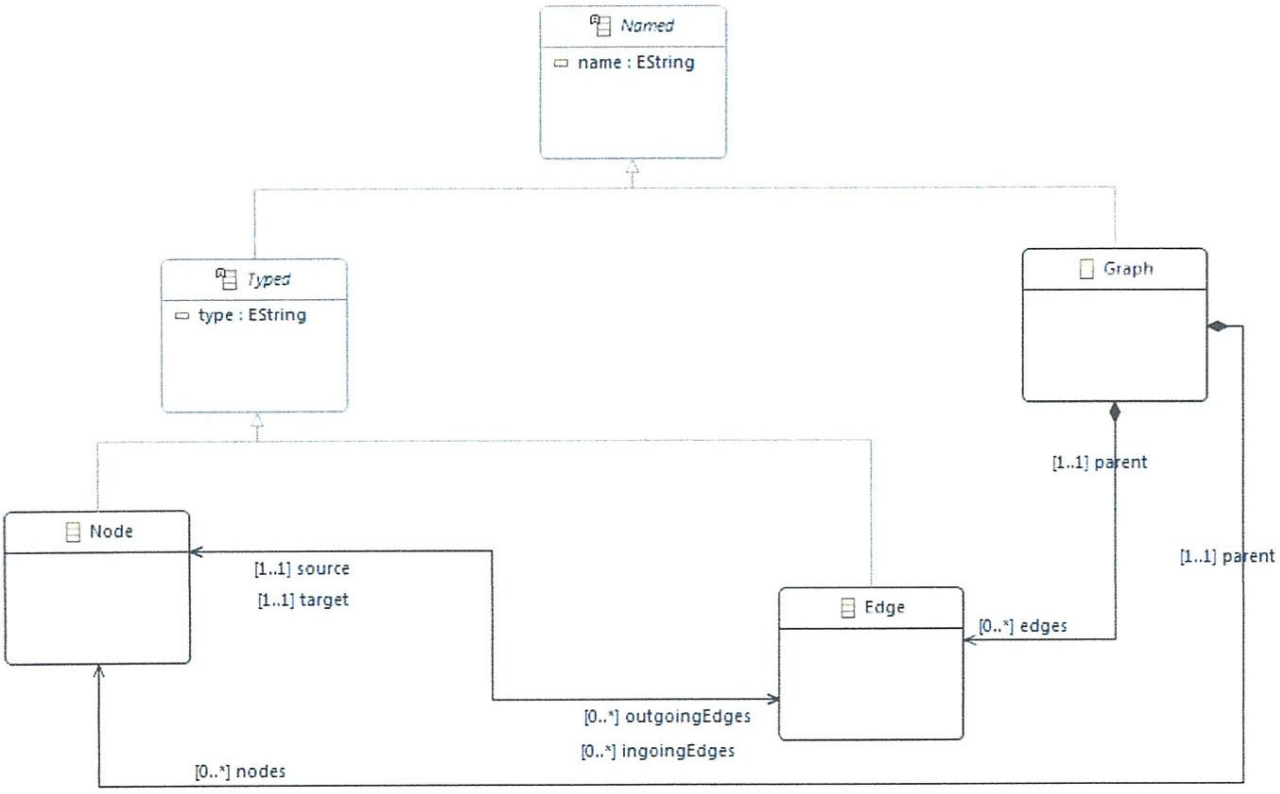

FIGURE 4.2 – Méta-modèle de graphe

4.2.3 Méta-modèle de Grammaire

Les différentes entités recensées pour la réalisation d'un méta-modèle de grammaire sont les suivantes :

Grammar La classe principale répresentant une grammaire, elle est composée de Rule.

Rule Répresentative des règles d'une grammaire, ne peut avoir qu'un ou un seul ParentGrammar(désignant la grammaire auquelle appartient une règle). Elle contient deux attributs name(nom de la règle) de type EString et de priority (priorité de la règle) ; elle ne peut avoir qu'une et une seule LHS et RHS ainsi qu'une existence de rélation de composition avec ces derniers.

LHS correspond à la partie gauche d'une règle, elle sera typé par un noeud et n'a qu'un seul parentRule(désignant la règle auquelle appartient un LHS).

RHS correspond à la partie droite d'une règle, elle sera typé par un graphe et n'a qu'un seul parentRule(désignant la règle auquelle appartient un LHS).

Embedding répresente le troisième élement du fameux triplet (M,D,E), n'a qu'un seul parentRule et n'est que EmbeddingMechanism (désignant les mécanismes d'intégration d'une règle) d'une et une seue règle mais pouvant avoir une ou plusieurs instructions de connexion.

ConnexionInstruction contient l'attribut m de type EString comme le m d'une connexion d'instruction, ce qu'elle répresente notament. IL appartient à une et une seule Embedding ce qui est traduit par parentEmbedding.

Voir Fig.4.3 répresentant notre méta-modèle de grammaire.

4.2.4 Algorithmes de réecriture de graphe

Disposant des deux méta-modèles, penchons nous maintenant sur comment automatiser les techniques on va dire d'application de grammaire sur un garphe.Nous nous permettons de shématiser ce mécanisme, l'illustration à la figure Fig.4.4. A été recensés quatres algorithmes de réecriture de graphe.

A partir de ce algorithme, dérrivera trois autres algorithmes à savoir celui de base en tenant compte de la direction de l'arc, en tenant compte et de la direction de l'arc et du label de l'arc et enfin en tenant compte et de la direction de l'arc et du label de l'arc avec option du changement de la direction et du label de l'arc. La différence entre les algorithmes réside au niveau de quelques propriété.

Intuituvement, sans ténir compte des preferences comme l'ordre d'exécution des règles et ou en ténant compte des priorités sur les différentes règles ou autres, l'execution d'une grammaire sur un graphe sérait comme suit :

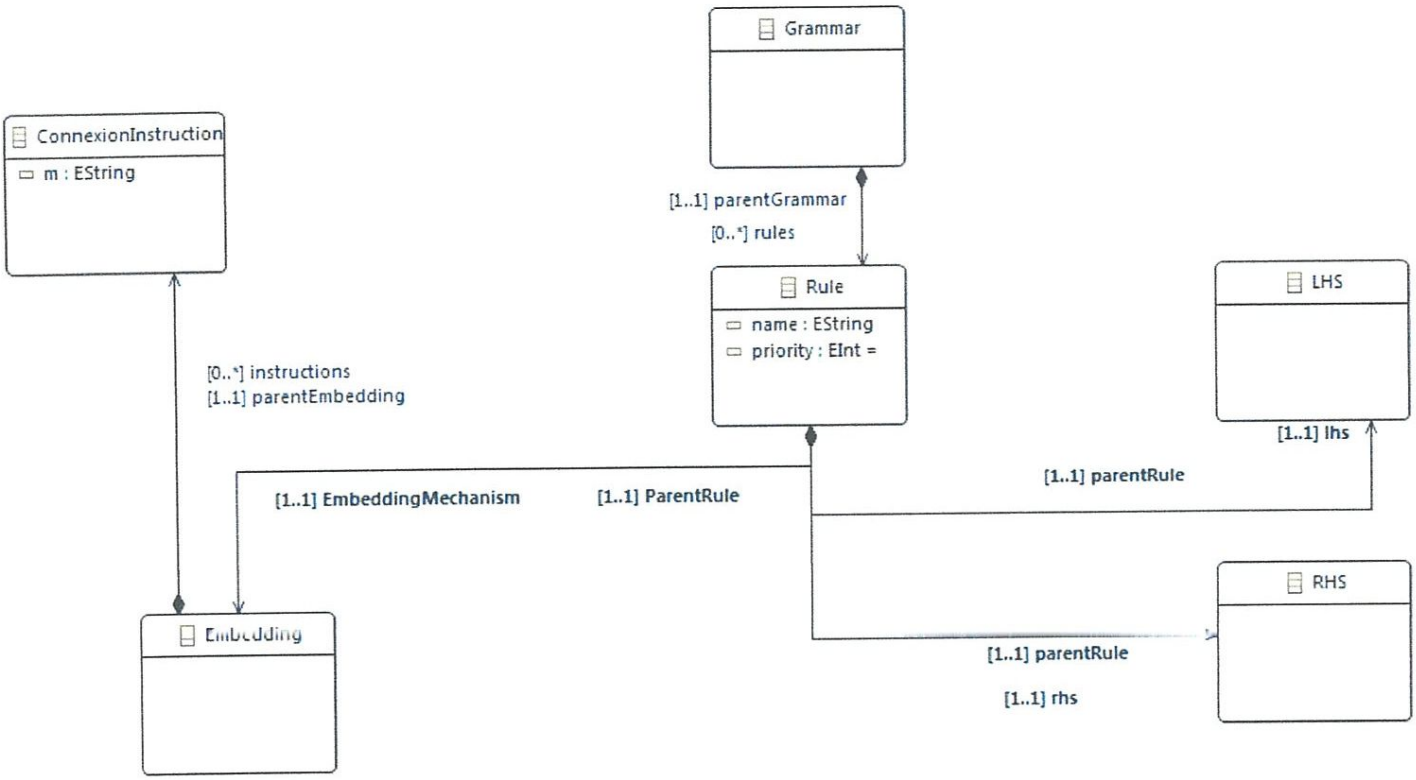

FIGURE 4.3 – Méta-modèle de grammaire

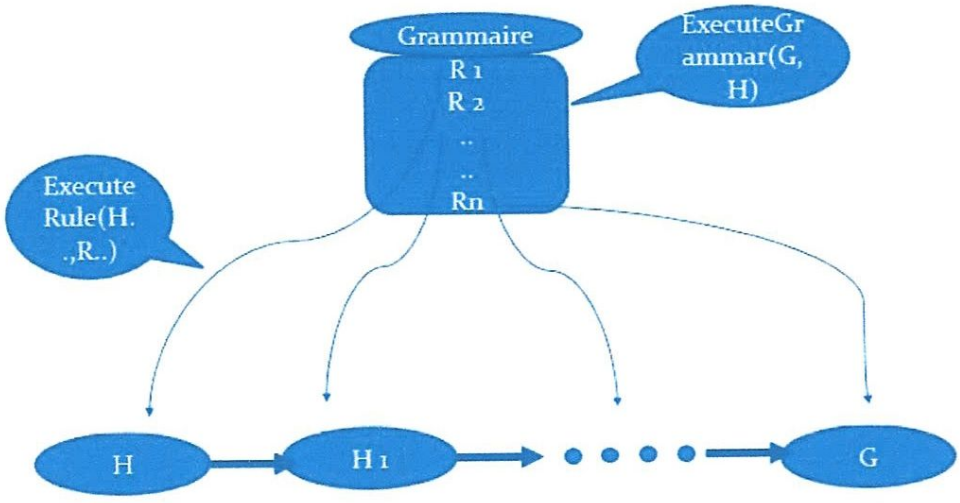

FIGURE 4.4 – REEcriture d'un graphe par une grammaire

Algorithm 1 Execute Rule, version simple

Require: H : Graph, R :Rule
Ensure: G : Graph
Integer $position \leftarrow -1$;
List $nodeH \leftarrow H.Nodes$;
Node $LhsNode \leftarrow R.Lhs().Node$;
$position \leftarrow occurence(nodeH, position, LhsNode)$;
if $position = -1$ **then**
write("Aucune occurence de "+ LhsNode) ;
else
supprimer(H, nodeH.get(position)) ;
ajouter(H, R.Rhs().SubGraph(), R.EmbeddingMechanism()) ;
end if
Prends en entrée un graphe et une règle, une prémiere fonction s'occupe de renvoyer la position d'un noeud correspondant au Lhs de la règle si elle existe, puis supprimer et ajouter s'occupent respectivement à la suppression du dit noeud avec ses liens et de l'ajout du Rhs de la règle au graphe H.

Algorithm 2 Execute Grammar

Require: H : Graph, Gr :Grammar
Ensure: G : Graph
List $rules \leftarrow Gr.Rules()$;
Iterator $R \leftarrow rules.iterator()$;
while R.hasnext() **do**
Rule $rul = R.next()$;
executeRule(H, rul) ;
end while
Prends en entrée un graphe et une grammaire à présent, elle consiste simplement au parcours de la liste des règles de la grammaire puis de les appliquées à tour de rôle au graphe à l'aide d'un des quatres (4) algorithmes de réecriture suppression du dit noeud avec ses liens et de l'ajout du Rhs de la règle
au graphe H.

4.2.5 Algorithmes annexe

Sont présentes dans cette sous section les algorithmes utilisés par ceux cités ci-déssus.

Algorithm 3 occurence, renvoie la position d'un noeud si $\subset$ liste de noeud

Require: nodesH : Liste de règle, position : Integer , LhsNode : Node
Ensure: position : Integer

```
for (i from position to nodesH.size()) do
  if (i = −1) then
    i ← 0
  end if
  Node n ← nodesH.get(i);
  if (n = LhsNode) then
    position ← i
    sortir ;
  end if
end for

return position ;
```

Recherche une occurence de LhsNode dans la liste nodesH
par le biais d'un parcours simple, en testant à chaque fois l'égalité
entre le noeud courant et lhs ; si cela est concluant il renvoie la
position de ce dernier dans position et le retourne.

4.3 Implémentation

4.3.1 Diagramme de cas d'utilisation

La figure 4.5 répresente le diagramme de cas d'utiliation de notre plugin.

4.3.2 Diagramme de séquence

La figure 4.6 répresente le diagramme de cas d'utiliation de notre plugin.

4.3.3 Outils de développement

4.3.3.1 Aspects matériel

le projet a été développé sur un micro-portable :

- Processeur : Intel (R) Core (TM) i3
- Capacité Mémoire (RAM) : 2 Go
- Vitesse d'horloge : 2.20 Ghz

Algorithm 4 supprimer, supprimer un node d'un graphe

Require: H : Graph, N :Node
Ensure: G : Graph
String *label* ← *N.name*();
H.nodes.remove(N) ;
List *edge* ← *H.Edges*();
for (i from 0 to edge.size()) **do**
e ← *edge.get*(*i*);
if (label = e.Source().Name() ou label = e.Target().Name()) **then**
H.Edges().remove(e) ;
i ← *i* − 1
end if
end for

A pour objectif, la suppression d'un noeud, d'un graphe donné ;
la fonction remove() joue son rôle parfaitement après des recherches concluantes.

Algorithm 5 Ajouter, ajout de D à H en utilisant les règles C

Require: H, D : Graph, C :Embedding
Ensure: G : Graph
List *Nh* ← *H.Nodes* ;
List *Nd* ← *D.Nodes* ;
for (i from 0 to Nd.size()) **do**
Node *d* ← *Nd.get*(*i*) ;
Node *node* ← *CreateNewNode*();
node.Name ← *d.Name*();
node.markNode() ;
node.parent ← *H*;
for (j from 0 to C.Instructions.size()) **do**
e ← *C.Instructions*().*get*(*j*);
for (k from 0 to Nh.size()) **do**
n ← *Nh.get*(*k*);
m ← *n.Mark*();
if (*m* = 0) **then**
if (*n.Name*() = *e.getM*() et *d.Name*() = *e.getD*().*getName*()) **then**
Edge *edge* ← *CreateNewEdge*();
edge.Source ← *n*;
edge.Target ← *node*;
edge.parent ← *H*;
end if
end if
end for
end for
end for

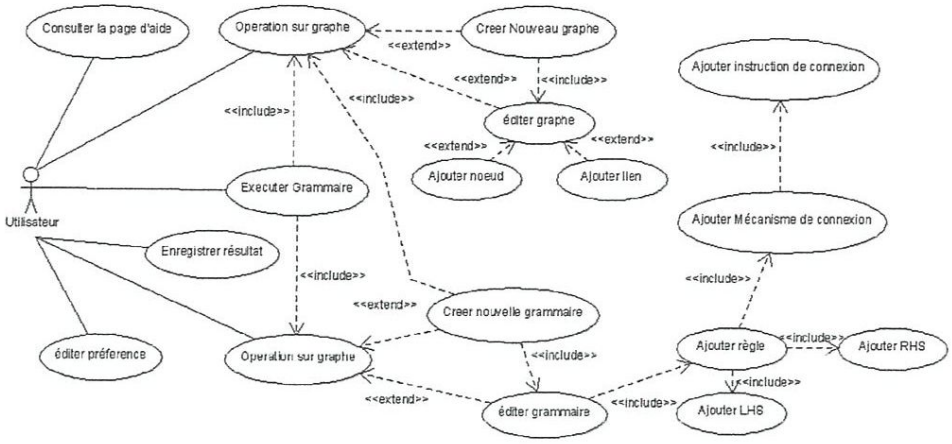

FIGURE 4.5 – Diagramme de cas d'utilisation

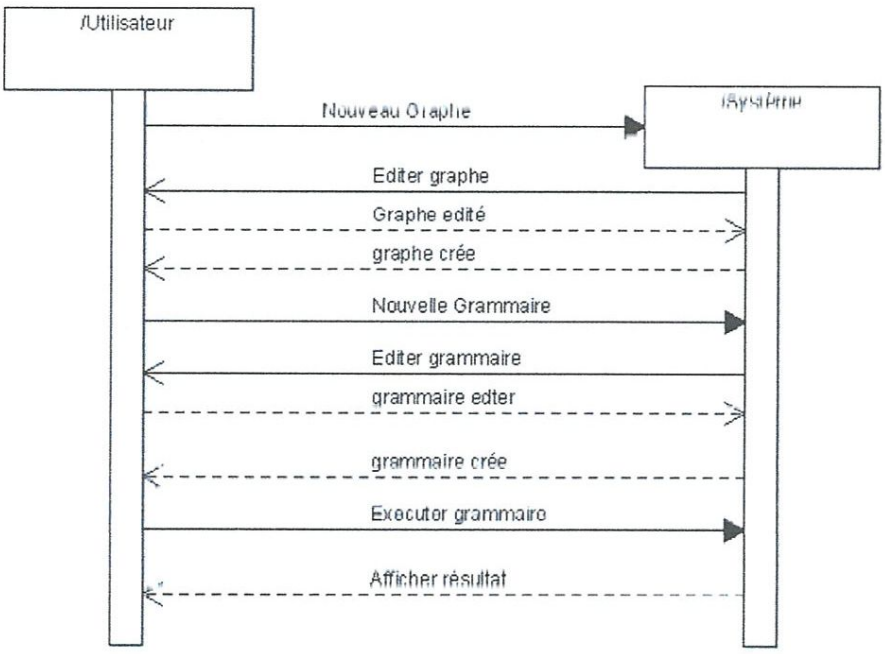

FIGURE 4.6 – Diagramme de séquence globale

— Capacité disque dur : 220 Go
— Système d'exploitation : Windows 7

4.3.3.2 Système de gestion de version GitHub

Un gestionnaire de version est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment. Ayant démarré sous forme de plateforme collaborative pour développeurs, GitHub est désormais le plus grand espace de stockage de travaux collaboratifs dans le monde.
Lien du repertoire du projet : https ://github.com/nberrehouma/Master-Project

4.3.3.3 Eclipse (EMF)

Le projet EMF a été lancé par IBM dans le but d'unifier les outils de développement développés par IBM utilisant des modèles basés. C'est dans l'optique de la concrétisation de l'approche MDA que nous Avons utilisé ce environnement.

4.3.4 Génération de code java correspondant aux Méta-modèles

Les méta-modèles étant définis, nous pouvons aisement générer les codes correspondants à l'aide d'EMF comme nous l'avons mentionés auparavant dans le chapitre deux. Nous aurons donc évidenment trois packages pour chaque méta-modèle : (graph, graph.impl, et graph.util) et (grammar, grammar.impl, et grammar.util). Ci-dessous commc cxcmplc, l'interface d'un noeud de notre modèle de graphe généré dans le package graph et d'une partie de son implementation(se trouvant dans graph.impl).

```
public interface Node extends Typed {
   Graph getParent();
   void setParent(Graph value);
   EList<Edge> getOutgoingEdges();
   EList<Edge> getIngoingEdges();
   //interfaces ajoutés manuellement
   public void markNode();
   public void unMarkNode();
   public int Mark();
   public void setId(int id);
   public int getId();
```

```
public class NodeImpl extends TypedImpl implements Node {
   ....
```

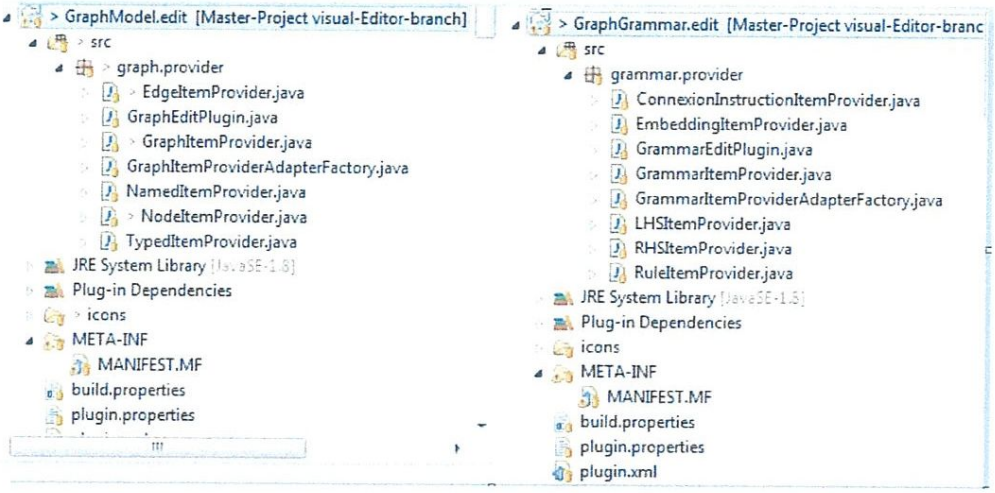

FIGURE 4.7 – Capture plu-gin .edit

```
....
private int marker = 0;
private int id;
public int getId() {
   return id;
}
public void setId(int id) {
   this.id = id;
}
public void markNode(){marker=1;}
public void unMarkNode(){marker=0;}
public int Mark(){return marker;}
.....
......
```

NB : remarquez bien qu'EMF nous permet de rajouter tout à fait sans le moindre problème dans chaque entité si neccessaire de nouveaux attributs et ou méthodes.

4.3.5 Génération de plu-gin .edit correspondant aux Méta-modèles

Respectivement nous aurons donc **GraphModel.edit** et **GraphGrammar.edit**, ils contiennent les classes adaptateurs nécessaire pour les éditeurs non graphique généré. Voir la figure 4.7 pour illustration.

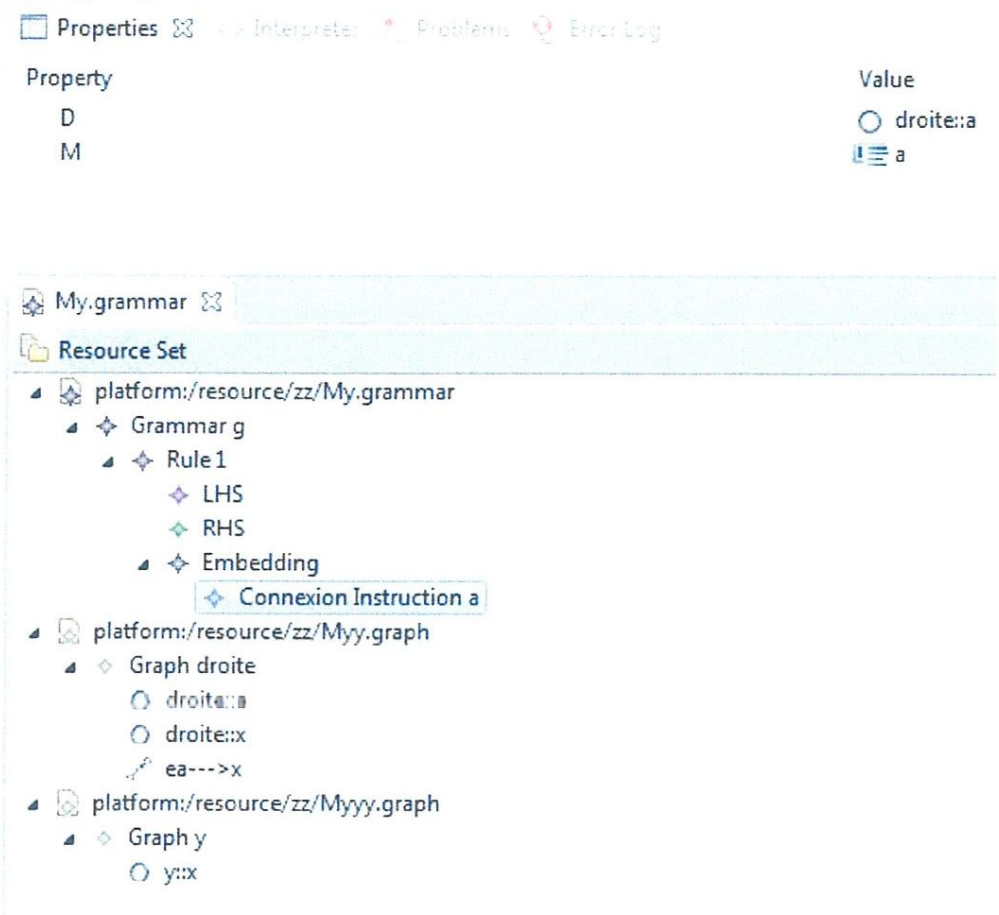

FIGURE 4.8 – Exemple grammaire dans l'éditor

4.3.6 Génération de plu-gin .editor correspondant aux Méta-modèles

Respectivement nous aurons donc GraphModel.editor et GraphGrammar.editor, plu-gin d'éditeur non visuel de graphe et de grammaire, il fournit une interface d'éditions des différents composants de chaque modèle. Voir la figure 4.8 pour un exemple d'édition de grammaire.

4.3.7 Wizards (création graphe et grammaire)

En éffets deux wizards ont été déployé pour la création d'un nouveau modèle de graphe et pour un nouveau modèle de grammaire (voir Fig.4.9 et 4.10).

— nouvelle extension org.eclipse.ui.newWizards(voir Fig.4.11).
— creation de la categorie Graph Transformation.
— définition des deux wizards (avec les propriétés).
— implémentation des comportements.

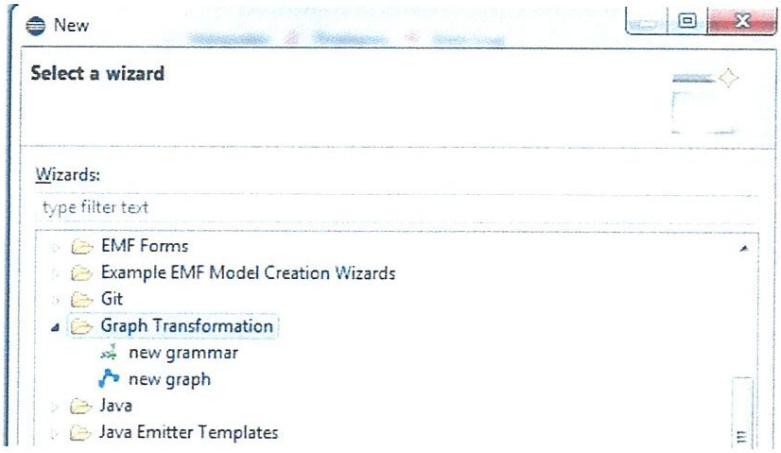

FIGURE 4.9 – Assistant wizard Graph Transformation

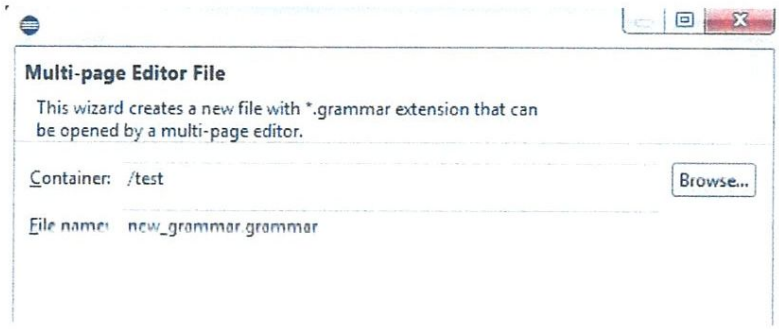

FIGURE 4.10 – Page 2 wizard (grammar creation)

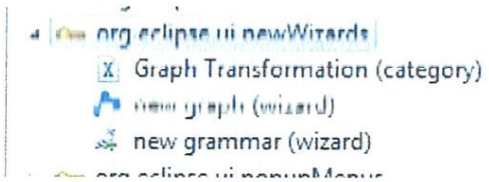

FIGURE 4.11 – New wizards

4.3.8 Les Editeurs graphiques

On aura besoin de deux éditeurs, un pour les graphes simples et l'autre pour les graphes orientés, Réalisés à l'aide de Sirius dont on avait parlé auparavant dans le chapitre 2. Pour ce qui est des grammaires,vue sa complexité, on s'en tiendra à l'édition en Tree offert par l'environnement EMF du moins pour ce qui est d'avoir une vue globale d'une grammaire sinon les éditeurs de graphes séront bien adaptés pour visualiser par exemple le Lhs (left hand side/partie gauche d'une règle) vue qu'il correspond à un noeud ou encore le Rhs(right hand side/ partie droite d'une règle) comme c'est un graphe etc ...
Les deux ont pratiquement tout en commun mise à part le style d'arc utilisé, simple sans orientation d'une part et orienté de l'autre. La figure Fig.4.12 montre une illustration de notre démarche de mise en place du système d'édition graphique de graphe(Non-orienté et orienté).

4.3.8.1 Description de la spécification

1. Création d'un Viewpoint Specification Project, qui contiendra la spécification.
2. Création d'un View Point (representation) dans le fichier .odesign généré par la création du projet, principale pproprieté l'extension GraphModel comme il est censé pouvoir éditer une instance de ce dernier.
3. Nouveau diagram de description.
4. Création des élements Node et Edge based Element pour la visualisation d'un noeud et d'un arc, définir leurs propriétés(styles, couleurs, bordures, label , ...)
5. Debut Ajouter les pallettes pour l'édition d'un noeud ou d'un arc.
6. Création de la section qui contiendra la pallette.
7. Enfin ajouter les outils de création de noeud et d'arc.

Cette description se veut d'être minimale et simpliste donc n'ont été mentionés que les grands points, voir Fig.4.13 pour des captures rélatives à quelques paramétrages et la figure 4.14 pour un exemple.

4.3.9 Page de préference

Une page de préference définis via le plugin PreferencePage d'éclipse (voir Fig.4.15.

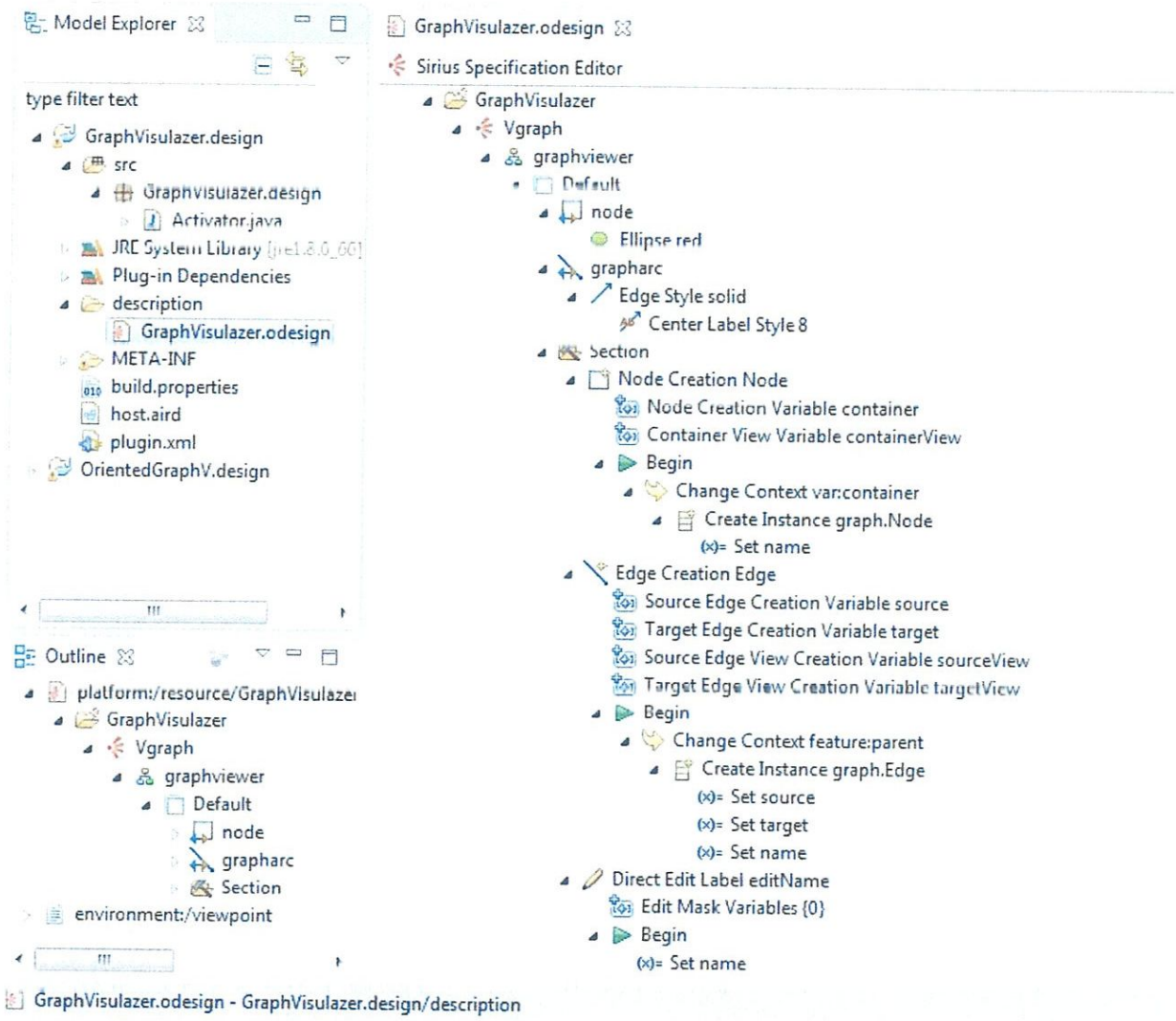

FIGURE 4.12 – Spécification Graphe (Sirius)

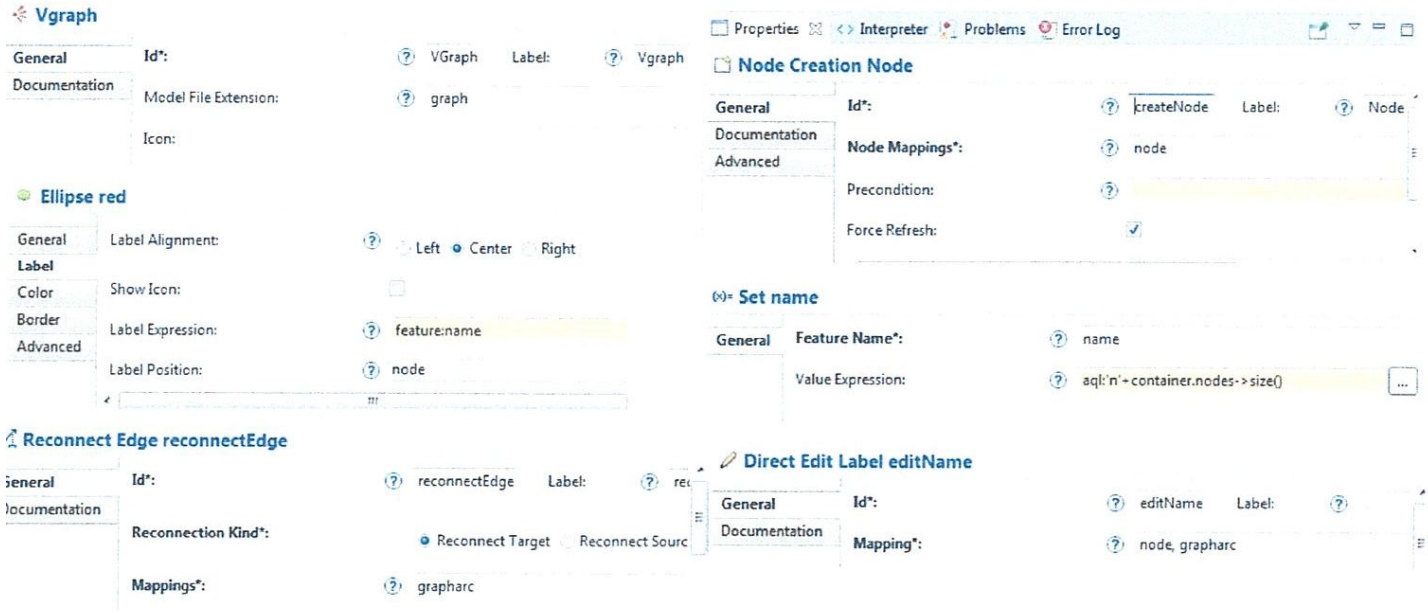

FIGURE 4.13 – Edition de quelques propriétés

FIGURE 4.14 – Exemple d'un graphe Orienté dans l'éditeur Graphique

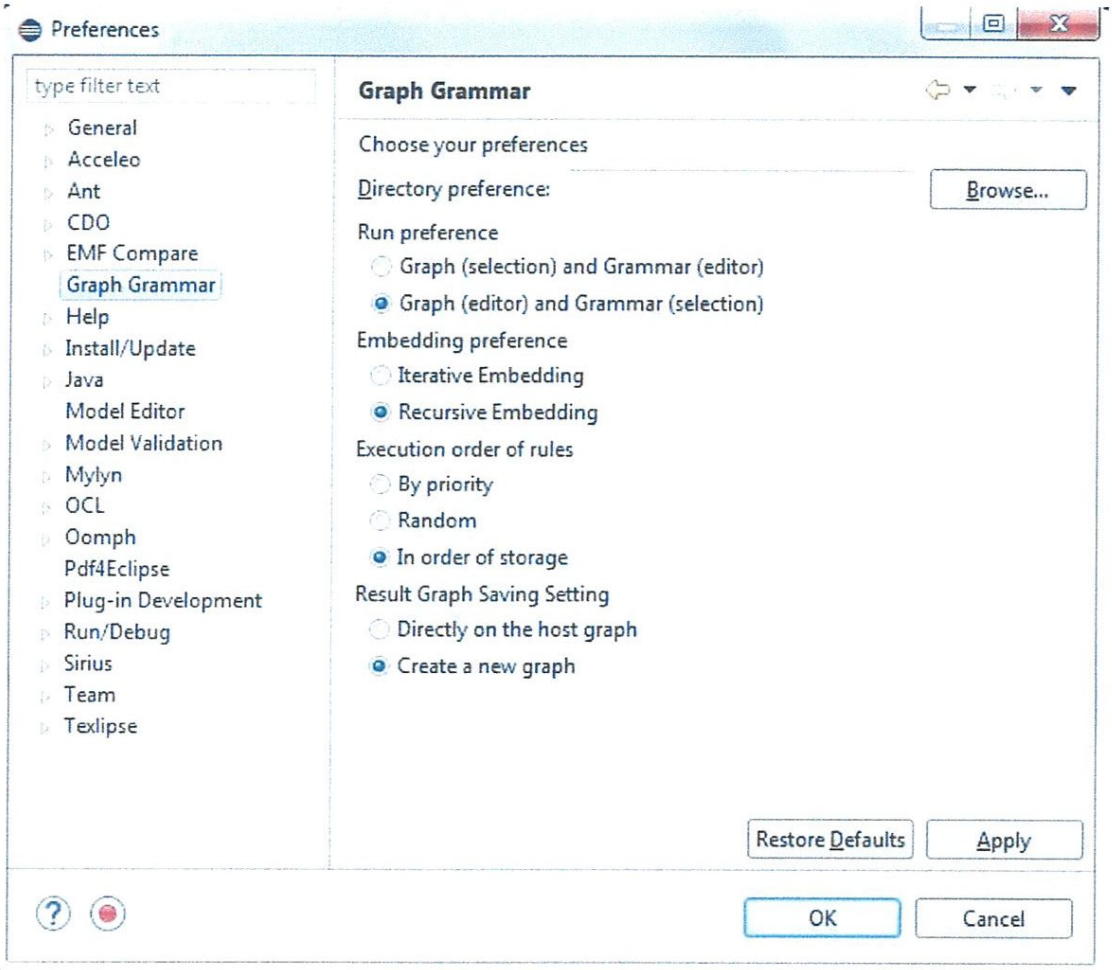

FIGURE 4.15 – Page de preference

FIGURE 4.16 – Graphe H (éditeur graphique)

FIGURE 4.17 – Graphe H (éditeur non graphique)

4.3.10 Tests et expérimentations

Illustration d'un exemple simple de reécriture d'un graphe par notre plugin.

— soit un graphe $H = \{X\}$, i.e., contient un seul noeud X voir Fig.4.16 et 4.17.

— soit un graphe $D = \{[a,X],[(a,X)]\}$, i.e., deux noeuds a et b reliés entre eux.

Soit $G = (\sum, \Delta, P, C, S)$ une grammaire de graphe avec :

— $\sum = \{a,b,X\}$

— $\Delta = \{X\}$

— $P = X \rightarrow D$

— $C = \{(a,a)\}$

L'illustration de l'exécution de la grammaire G sur le graphe H à la figure 4.18.

4.4 Conclusion

Dans ce chapitre, nous avons présenté toutes les différentes étapes de la phase de conception avec en occurence une vision globale puis algorithmiquement parlant,l'implementation concrète de ces derniers ; une mise en pratique par la suite avec des tests et différentes expérimentations du système ont été menés.

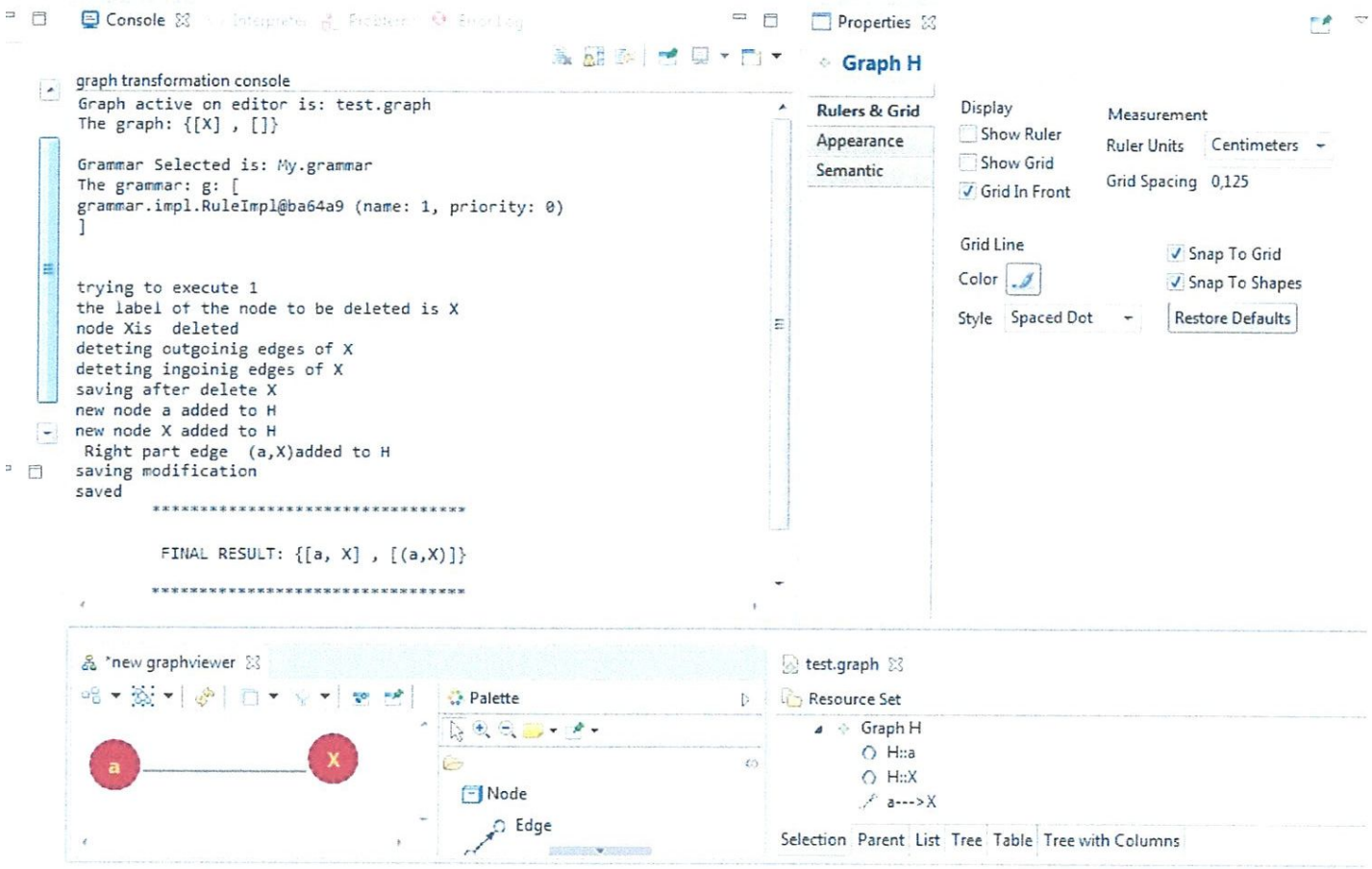

FIGURE 4.18 – Résultat G sur H

Conclusion Générale

Ce projet que nous avons eu l'occasion de conduire à bout, rentre dans le cadre de la mise en oeuvre de l'approche MDA (Model driven Architecture) donc de l'utilisation des modèles de données qui consistait à la mise en place d'un système de transformation. Théorie de transformation algébrique, base du travail a été le prémier point qu'il fallait s'approprié. Pendant la réalisation de ce projet nous avons été amenés à travailler de très près avec l'environnement eclipse (Eclipse Modeling Framework) en exploitant au mieux les possibilités qu'offraient la plateforme en matière de développement de plugin (editor, wizard, views, ...), plus une initiation dans ce monde qui nous étaient totalement inconnus. Ce travail nous a permis de connaitre l'utilisation de certains outils comme Latex, ArgoUML, github etc.

Comme tout travail de recherche évidenment, nous avons rencontré des difficultés tout au long du projet. Le problème majeur était de pouvoir maitriser les outils de travail principalement les plugins usuels d'éclipse en pas moins de quelques mois.

En guise de perspectives, on pourrait :

- Utiliser un graphe comme partie gauche d'une règle au lieu d'un seul noeud ;
- Prendre en compte les actions qui se passent lors de l'execution d'une règle ;
- Rajouter au plugin un parser(analyseur syntaxique) de grammaire.
- Définir un éditeur graphique pour une grammaire.
- etc...

ANNEXE A

Rédaction

Pour la rédaction de ce présent mémoire, nous avons utilisé Latex.
C'est un langage informatique de description de document. Ce langage a été conçu pour rendre la création de document facile pour l'auteur d'un coté, et produire des documents lisibles et clairs du côté des lecteurs.
Latex contraire d'open Office ou MS-Office n'est pas WYSIWYG [1], ce n'est pas vraiment un traitement de texte mais un composeur de texte,Nous avons utilisé la version 1.5.0 TeX-lipse, qui est un plugin éclipse pour Latex (https ://marketplace.eclipse.org/content/texlipse) [Chofr]

1. What you see is what you get

ANNEXE B

Quelques code source important

B.1 Chargement d'un graphe

```
public static Graph loadGraph(String path){
      GraphPackage graphPackage = GraphPackage.eINSTANCE;
      ResourceSet resourceSet = new ResourceSetImpl();
      resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("graph",
          new XMIResourceFactoryImpl());
      URI uriGraph = URI.createURI(path);
      Resource resourceGraph = resourceSet.getResource(uriGraph,true);
      try
      {
         resourceGraph.load(null);
         Graph g = (Graph)resourceGraph.getContents().get(0);
         return g;
      }
      catch (IOException e)
      {
      System.out.println("failed to read from " + uriGraph );
      return null;
      }
   }
```

B.2 Sauvegarde d'un graphe

```
   public static void saveGraph (Graph graph)
```

```
    {
     Resource resourceGraph = graph.eResource();//
         resourceSet.getResource(uriGraph,true);
     try
     {
          resourceGraph.save(null);
      } catch (IOException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
          out.println("saving problem\n"+e.getMessage() );
        }
     out.println("saved");
       }
  }
```

B.3 Chargement d'une grammaire

```
public static Grammar loadGrammar(String path){
      GraphPackage graphPackage = GraphPackage.eINSTANCE;
      GrammarPackage grammarPackage = GrammarPackage.eINSTANCE;
      ResourceSet resourceSet = new ResourceSetImpl();
      resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("graph",
          new XMIResourceFactoryImpl());
      resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("grammar",
          new XMIResourceFactoryImpl());
      URI uriGrammar = URI.createURI(path);
      Resource resourceGrammar = resourceSet.getResource(uriGrammar,true);
      try
      {
       resourceGrammar.load(null);
      Grammar g = (Grammar)resourceGrammar.getContents().get(0);
        return g;
        }
      catch (IOException e)
      {
      out.println("failed to read from "+ uriGrammar);
      return null;
      }
```

```
    }
}
```

Bibliographie

[416] BE 4. *Ingénierie dirigée par les modèles*, 2016 (accès le 8/06/2016).

[All13] Freddy Allilaire. Eclipse sirius, la modélisation graphique à la portée de tous. Technical report, octobre 2013.

[Bar08] Jacques Barzic. Eclipse et ses plugins de modélisation (emf gef gmf). Technical report, janvier 2008.

[Chofr] Daniel Choï. *Exemple darticle ou de rapport avec Latex*, choi@meca.unicaen.fr.

[DEGM05] Matthew Dickerson, David Eppstein, Michael T Goodrich, and Jeremy Yu Meng. Confluent drawings : Visualizing non-planar diagrams in a planar way. *J. Graph Algorithms Appl.*, 9(1), 2005.

[DS08] Marcelo Paternostro et Ed Merks Dave Steinberg, Frank Budinsky. *Eclipse Modeling Framework*, 16/12/2008.

[Édu06] Hachette Éducation. Déclic terminale es, enseignement obligatoire et option, 2006.

[ecl16] *Eclipse Modeling framework https ://fr.m.wikipedia.org »ki »emf*, 2016 (accès le 09/06/2016).

[edi16] *Présentation d'EMF.edit , http ://www.ibm.com/support/knowledgecenter/fr/*, 2016 (accès le 07/04/2016).

[eK11] Maghmouli et Kouahla. Editeur visuel des diagrammes de classe uml, master , université de guelma. Technical report, 2011.

[Gai07] Alban Gaignard. *Eclipse Modeling Framework, Genie logiciel SI5*, 12/11/2007.

[htt16] The Eclipse Foundation http ://www.eclipse.org/modeling/emf/. *Eclipse Modeling framework*, 2016 (accès le 18 Mars 2016).

[Mer08] Dave Steinberg Frank Budinsky Marcelo Paternostro Ed Merks. *EMF Eclipse Modeling Framework, second edition*, 16 Decembre 2008.

[Roz82] G. Rozenberg. *HANDBOOK of GRAPH GRAMMARS and COMPUTING by GRAPH TRANSFORMATION, World Scientific*, 1982.

[Rubnc] Eric Clayberg Dan Rubel. *Eclipse plugin, third edition*, 2009 Pearson Education Inc.

[Sei10] Xavier Seignard. Théorie des transformations de graphes : une approche algébrique. Technical report, juin 2010.

[SF14] Benkirat Soumia Salhi Fatima. Transformation des modèles martes vers system c, master université de guelma. Technical report, juin 2014.

[sss16] *Syntaxe abstraite https ://fr.m.wikipedia.org»ki »syntaxe*, 2016 (accès le 08/04/2016).

[wik16] *Eclipse https ://fr.m.wikipedia.org*, 2016 (accès le 03/05/2016).

Glossaire

Abréviations	Significations
NLC	Node label Controlled mechanism
NCE	Neighbourhood controlled Embedding
dNCE	directed Neighbourhood controlled Embedding
eNCE	edge Neighbourhood controlled Embedding
edNCE	edge Directed Neighbourhood controlled Embedding
RHS	Right Hand Side: Partie droite d'une règle
LHS	Left Hand Side: Partie gauche d'une règle
EMF	Eclipse Modeling Framework
XML	Extensible Markup Language
XMI	XML Metadata Interchange
MOF	Meta Object facility
MDA	Model Driven Architecture
PIM	Platform Independent Model
PSM	Platform Specific Model
CIM	Computation Independent Model
UML	Unified Modeling Language : langage de modélisation unifié
SDL	Specification and Description Language
URI	Uniform Resource Identifier
IHM	Interface Homme Machine
OCL	Object Constraint Language
SQL	Structured Query Language : langage de requête structurée
GMF	Eclipse Graphical Modeling Framework
GEF	Graphical Editing Framework
IBM	International Business Machines
JDT	The Java Development Kit
PDT	The Plugin Development Kit
AWT	Abstract Window Toolkit: Bibliothèque graphique pour Java
API	Application Programming Interface
SWT	Standard Widget Toolkit
CVS	Concurrent Version System
HTML	Hypertext Markup Language

PDE	Plugin Development Environment
OSGI	Open Services Gateway Initiative
RCP	{Rich Client Plateform: Plateforme client riche
OGM	Organisme Génétiquement Modifié
IDE	Integrated Development Environment: Environnement de développement intégré
UI	User Interface : Interface utilisateur
PMC	Project Management Committee