REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

N° d'ordre : .....

UNIVERSITE DE M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

Département d'Informatique

MEMOIRE de fin d'étude

Présenté pour l'obtention du diplôme de MASTER

Domaine : Mathématiques et Informatique
Filière : Informatique
Spécialité : Systèmes d'Informations Avancés

Par: KHOMS SalahEddine

SUJET

Approche de transformation et vérification des architectures logicielle

Soutenu publiquement le : 25 /06 /2013 devant le jury composé de :

BOUBAKIR	**Université de M'sila**	**Président**
MOKHTARI Rabah	**Université de M'sila**	**Rapporteur**
BARKAT Abd Elbasset	**Université de M'sila**	**Examinateur**

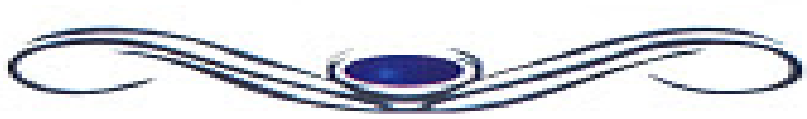
Promotion : 2012 /2013

ملخص

يعتبر المخطط احد الجوانب الهامة في عملية تصميم الأنظمة خاصة المعقدة منها ، وهناك الكثير من انواع المخططات ، ودراستنا ترتكز على المخططات بالمكونات او ما يعرف بهندسة البرمجيات .

ولوصف هندسة البرمجيات، يستخدم الكتاب عادة مصطلح ADL (لغة وصف هندسة البرمجيات)، UMLوغيرها من التقنيات. ونحن نسعى في هذا العمل لتوفير أداة EMF استنادا إلى وصف هندسة البرمجيات. تستخدم هذه الأداة مجموعة من القواعد ATL لتحويل هندسة البرمجيات الى ملف XML .

الكلمات المفتاحية : IDM, MDA, EMF, ATL، تحويل النماذج ، هندسة البرمجيات ، المكونات .



Abstract

The key design aspect of the software system in its highest level, is its overall architecture in which the system is as a set of components and interconnections between them.

To describe a software architecture, the authors use the so-called ADL (Architecture Description Language), UML and other techniques. We seek in this work to provide for EMF based on the description of the AL tool. This tool uses a set of ATL rules for transforming XML Software Architecture.

Keywords : IDM, MDA, EMF, ATL, transformation de modele, software architecture, component.

Résumé

L’aspect clé de conception du système logiciel, dans son plus haut niveau, est son architecture globale dans lequel le système se présente comme un ensemble de composants et d’interconnexions qui les relient.

Pour décrire une architecture logicielle les auteurs utilisent ce qu’on appelle les ADL (Architecture Description Language), le langage UML et d’autres techniques. Nous cherchons dans ce travail de fournir un outil basé sur EMF pour la description des AL. Cet outil utilise un ensemble de règles ATL pour la transformation des AL en XML.

Mots clé : IDM, MDA, EMF, ATL, model transformation, architecture logicielle, composant.

DEDICACES

Je dédie ce travail en premier lieu à mère ***LATIFA*** *qui me très cher en témoignage à leur soutienne pendant toute ma vie car aucun mot ne pourra exprimer ma haute gratitude et profonde affection.*

Et je demande à dieu tout puissant d'avoir pitié de mon père et la maison au paradis.

Je le dédie aussi :

Spécialement A ma petite sœur : NADJET

Et mon seul frère : HANI

A toutes les personnes qui ont contribué à la réalisation de ce modeste travail.

Remerciements

Avant tout, nous remercierons Allah de nous avoir donné la volonté afin d'arriver à la finalité de ce modeste travail.

Je tiens à remercier vivement ***Mr Mokhtari Rabah*** d'avoir accepté de diriger ce travail, je le remercie infiniment pour sa patience et son soutien.

Je tiens également à remercier Mme Milouh Amal de ma aidé et des conseils précieux.

Je tiens également à remercier les membres de jury.

En fin, je tiens à remercier tous ceux qui ont contribué d'une façon ou d'une autre à la réalisation de ce mémoire.

Table des matières

Chapitre 3 : l'architecture des logiciels et la description d'architecture.

Chapitre 4 : La génération d'un outil pour les architectures logicielles et la transformation des Architectures logicielle vers XML.

Liste des figures

Liste des tableaux

INTRODUCTION GENERALE

Actuellement, un grand intérêt est porté au domaine de l'architecture logicielle. Cet intérêt est motivé principalement par la réduction des coûts et les délais de développement des systèmes logiciels. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant que de le concevoir, le coder, le tester, le déboguer et le documenter. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre ces composants. Elle joue le rôle de passerelle entre l'expression des besoins du système logiciel et l'étape de codage du logiciel. Enfin, l'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage des composants logiciels.

Pour décrire l'architecture logicielle et modéliser ainsi les interactions entre les composants d'un système logiciel, deux principales approches ont vu le jour depuis quelques années : la modélisation par composants et la modélisation par modèle. La première approche, qui a émergé au sein de la communauté de recherche « Architectures Logicielles » décrit un système logiciel comme un ensemble de composants qui interagissent entre eux par le biais de connecteurs. Les chercheurs de ce domaine ont permis d'établir les bases intrinsèques pour développer de nouveaux langages de description d'architectures logicielles. L'architecture logicielle a permis notamment de prendre en compte les descriptions de haut niveau des systèmes complexes et de raisonner sur leurs propriétés à un haut niveau d'abstraction (protocole d'interaction, conformité avec d'autres architectures, etc.) La seconde approche est devenue un standard de description d'un système logiciel durant ces dernières années. Avec l'unification des méthodes de développement objet sous le langage UML.

Le manque de réutilisabilité et l'automatisation est l'un des problèmes majeurs des systèmes informatiques. Il est le résultat de l'incompatibilité des principes architecturaux des composants qui interagissent entre eux. Il devient alors difficile de contrôler les interactions et les interconnexions entre composants. L'implémentation peut s'avérer de plus en plus difficile. L'absence de quelques concepts architecturaux (connecteur, configuration, …etc.), ainsi que le non-respect de l'architecture logicielle dans les plates- formes d'exécution sont les conséquences des nouveaux problèmes de maintenance et d'interopérabilité. Pour affronter ces problèmes, la solution recommandée est la définition d'une approche pour la transformation et vérification l'architecture logicielle, et ensuite l'intégration de cette approche dans la démarche MDA

Alors L'objectif de nos travaux est de proposé une approche de transformation automatisé pour les architectures logicielle à l'aide des plugins éclipse (***EMF, GMF***) et le langage de transformation ***ATL***, puis la vérification des architectures logicielles à l'aide de l'outil de vérification et l'analyse ***LTSA*** (Labelled Transition System Analyser).

Notre travail est composé de quatre chapitres :

Le chapitre 1 est consacré à l'étude des concepts de base de l'ingénierie dirigée par les modèles (IDM) et leur approche l'architecture dirigée par les modèles (MDA), suive par la transformation des modèles et le processus de la vérification dans l'IDM.

Dans le chapitre 2 en nous avons introduit des outils et les plugins qui supportent la transformation des modèles.

Dans le chapitre 3 l'étude des concepts de base de l'architecture logicielle, Les concepts de base des langages de description d'architecture et les langages de description d'architecture les plus représentatifs.

Dans le chapitre 4 en a le méta-modèle des Architecture logicielle et les règle de transformation d'architecture logicielle vers XML et un exemple pour illustrer cette transformation.

CHAPITRE 1

L'INGENIERIE DIRIGEE PAR LES MODELES ET LA TRANSFORMATION DES MODELES.

Dans ce chapitre nous présentons de façon globale L'ingénierie dirigée par les modèles (IDM). Nous abordons les principes clés de l'ingénierie IDM et de ses différentes centrées sur les modèles. En suite on entreprend les approches de l'IDM et plus particulièrement l'architecture dirigée par les modèles (MDA), la transformation de modèle et son langage ATL.

1 Les principes généraux de l'IDM.

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du «***tout est modèle***». Cette nouvelle approche peut être considérée à la fois en continuité et en rupture avec les précédents travaux. Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.). [1]
L'IDM peut être vue comme une famille d'approches qui se développent à la fois dans les laboratoires de recherche et chez les industriels impliqués dans les grands projets de développement logiciels.

Cette approche vise non seulement à favoriser un « *génie* » logiciel plus proche des métiers en autorisant une appréhension des applications selon différents points de vues (*modèles*) exprimés séparément. Mais elle intègre également comme fondamentales la composition et mise en cohérence de ces perspectives. De plus elle se veut productive en automatisant la prise en charge des outils relatifs à la validation des modèles, les transformations et les générations de code. Malgré ses balbutiements initiaux, l'IDM cible une production logicielle bien fondée. [2]

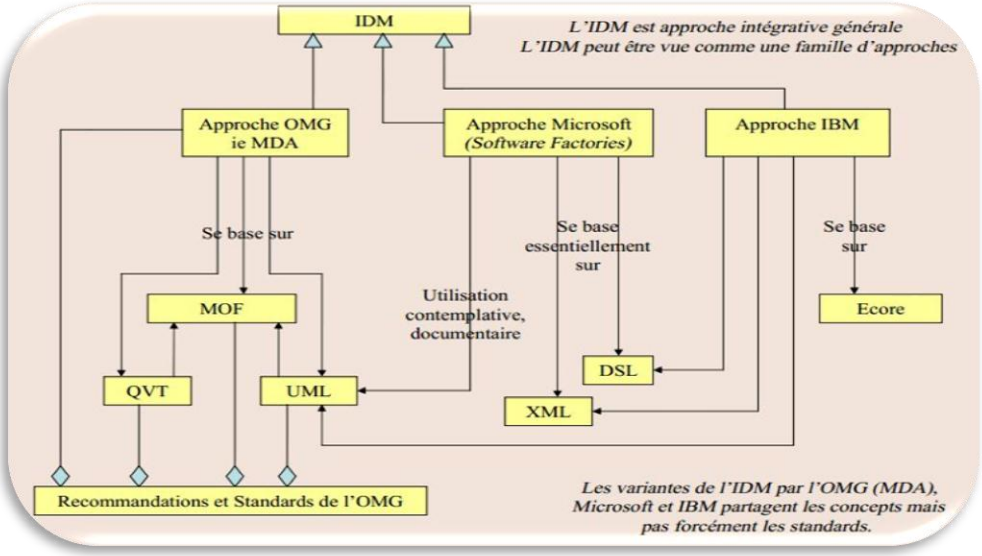


FIG 1.1 : Approches orientées modèles en ingénierie du logiciel, des systèmes et des données.

1.1 Les concepts de l'IDM.

a- Le système : au sens de l'IDM, est ce que l'on désire modéliser. Il est le réel. Une typologie des systèmes est proposée : système physique (chien, maison, vélo,...), système digital (fichier, diagramme, logiciel,...), système abstrait (compte bancaire, cercle, cours,..). [3] Un système peut aussi être dynamique (changer dans le temps) ou statique (immuable, du moins dans une échelle de temps relativement grande).

b- Un modèle : en réalité il n'existe pas à ce jour de définition universelle :

"***A model*** *represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*". [Jeff Rothenberg, the Nature of Modeling, 1989].

"***Un modèle*** *représente un système réel en se basant sur la sémantique et les règles qui conditionnent ses éléments; en d'autres termes il ne doit en aucun cas briser la structure ou les contraintes que les éléments du système réel respectent*."[4]

c- Méta-modèle : Un méta-modèle est un modèle qui définit le langage d'expression d'un modèle, c.-à-d. le langage de modélisation. La notion de méta-modèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée *conformeA* et nommée c sur la figure 1.2. En cartographie, il est effectivement indispensable d'associer à chaque carte la description du « *langage* »utilisé pour réaliser cette carte. Ceci se fait notamment sous la forme d'une légende explicite. La

carte doit, pour être utilisable, être conforme à cette légende. Plusieurs cartes peuvent être conformes à une même légende. La légende est alors considérée comme un modèle représentant cet ensemble de cartes (μ) et à laquelle chacune d'entre elles doit se conformer (c). [1]

Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) méta-modèle(s) qui jouent le rôle de modèle(s) de ce langage.

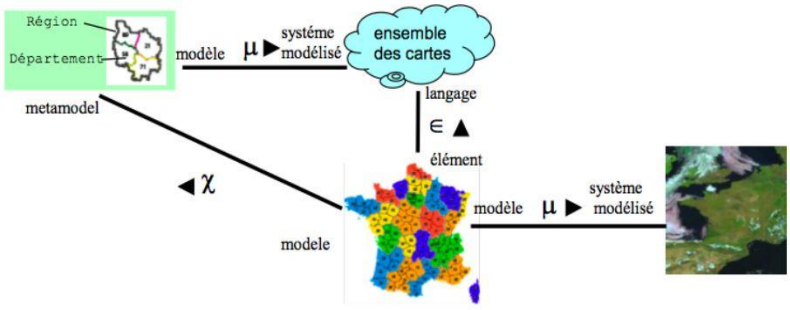


FIG 1.2 – Relations entre système, modèle, méta-modèle et langage.

d- Méta-méta-modèle : Un méta-méta-modèle est un modèle qui décrit un langage de méta-modélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (cf. figure 1.3). Le monde réel est représenté à la base de la pyramide (niveau M0). Les modèles représentant cette réalité constituent le niveau M1. Les méta-modèles permettant la définition de ces modèles (p. ex. UML) constituent le niveau M2. Enfin, le méta-méta-modèle, unique et méta-circulaire, est représenté au sommet de la pyramide (niveau M3).[4]

L'approche consistant à considérer une hiérarchie de méta-modèles n'est pas propre à l'OMG, ni même à l'IDM, puisqu'elle est utilisée depuis longtemps dans de nombreux domaines de l'informatique. Chaque hiérarchie définit un espace technique . Nous distinguons par exemple le *modelware* (espace technique des modèles), le *grammarware* (espace technique des grammaires définies par les langages tels que BNF 2 ou EBNF 3), le BDware (espace technique des bases de données), etc.

2 Les approches de l'Ingénierie dirigée par les modèles :

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

2.1 L'Architecture Dirigée par les Modèles (MDA) :

En novembre 2000, l'OMG (Object Management Group), consortium de plus de 1000 entreprises, initie l'approche MDA (Model Driven Architecture). Cette approche a pour but d'apporter une nouvelle vision unifiée de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux couts excessifs de migration technologique. Cette séparation autorise alors la capitalisation du savoir logiciel et du savoir-faire de l'entreprise.

La proposition de l'OMG recommande l'utilisation d'UML (Unified Modeling Language), MOF (Meta Object Facility) et XMI (XML Metadata Interchange). Le MOF est le méta-méta-modèle standard unique. XMI est le format qui va permettre l'échange de modèles et de méta-modèles. Il est basé sur XML (Extensible Markup Language). Enfin, UML est l'un des premiers méta-modèles basé sur le MOF adopté par l'OMG. [5]

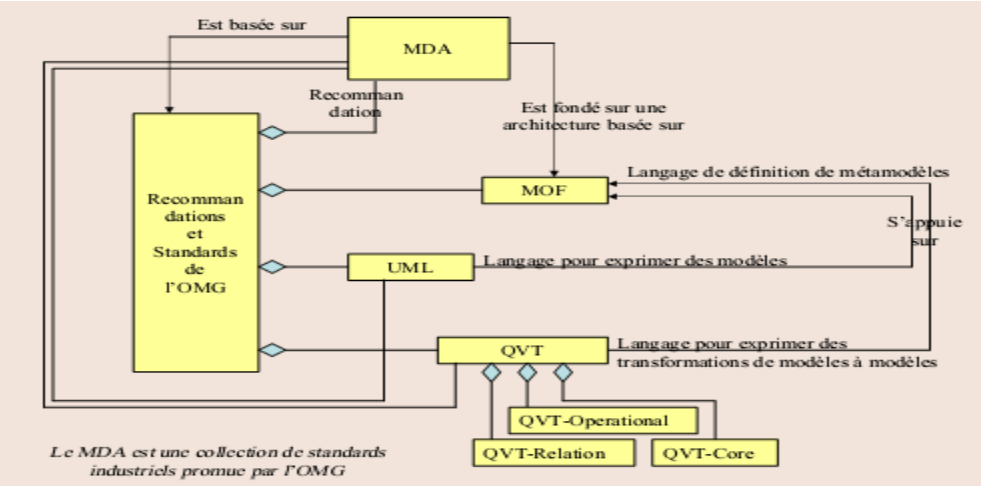


FIG 1.3 : Aperçu de l'approche OMG de l'IDM : MDA

2.1.1 Le principe de base : des pims vers les psms

L'approche MDA de l'OMG suscite actuellement un très grand intérêt. Elle concrétise une évolution très marquée de la technologie informatique visant à substituer aux approches interprétatives (à djonctions de couches logicielles pour prendre en charge les problèmes d'interopérabilité) des approches transformationnelles (traduction du logique métier neutre vers des modèles liés aux plate-formes d'exécution).

D'une façon macroscopique, le MDA peut se résumer à l'élaboration de modèles indépendants de plates -formes et à la transformation de ceux -ci en modèles dépendants de plates -formes.
Plus précisément, l'approche MDA définit deux types de modèles :[3]

- Les PIMs (Platform Independent Models) sont des modèles qui n'ont pas de dépendance avec les plates -formes techniques (c'est -à-dire EJB, CORBA, DotNet, XML, etc.). Les PIMs représentent par exemple les différentes entités fonctionnelles d'un système avec leurs interactions, exprimées uniquement en termes de la logique d'entreprise.
- Les PSMs (Platform Specific Models) quant à eux sont dépendants de plates -formes techniques. Les PSMs servent essentiellement de base à la génération de code exécutable vers ces mêmes plates –formes techniques.

Le MDA identifie plusieurs types de transformations:

1- PIM vers PIM : Ces transformations s'effectuent pour ajouter ou soustraire des informations aux modèles. Le fait de masquer par exemple quelques éléments afin de s'abstraire de détails fonctionnels est typiquement une transformation PIM vers PIM. Dans l'autre sens, le passage du problème à la solution est la plus naturelle des transformations PIM vers PIM. Il est important de noter que ces transformations ne sont pas toujours automatisables.

2- PIM vers PSM : Ces transformations s'effectuent lorsque les PIMs sont suffisamment complets pour pouvoir être "immergés" dans une plate-forme technique. L'opération qui consiste à ajouter des informations propres à une plate-forme technique pour permettre la génération de code est une transformation PIM vers PSM. A l'heure actuelle, les plates -formes techniques visées sont Dot Net, J2EE, XML et CORBA. Il apparaît clairement que ce sont les règles qui permettent ces transformations qui sont importantes et qui doivent être généralisées et capitalisées. Ces transformations ont donc pour but d'être fortement automatisées.

3- PSM vers PSM: Ces transformations s'effectuent lors des phases de déploiement, d'optimisation ou de reconfiguration. Notons de plus qu'une unique transformation PIM vers

PSM n'est pas toujours suffisante pour permettre la génération de code, il faudra alors parfois transformer les PSM en PSM en utilisant des formalismes intermédiaires (exemple de passage d'UML à SDL puis de SDL à C++).[5]

2.1.2 Architecture A Quatre Niveaux :

La modélisation logicielle a radicalement évolué au cours de ces dernières années. Ces changements majeurs sont principalement supportés par l'OMG. Il y a aujourd'hui un consensus autour d'une architecture à quatre niveaux adopté principalement par l'OMG et l'UML.

Comme le montre la figure 4, quatre niveaux caractérisent ce standard de méta-modélisation. Le niveau M0 qui est le niveau des données réelles, composé des informations que l'on souhaite modéliser. Ce niveau est souvent considéré comme étant le monde réel. Lorsqu'on veut décrire les informations contenues dans le niveau M0, cette activité donne naissance à un modèle appartenant au niveau M1. Un modèle UML appartient au niveau M1. Le niveau M2 est composé des langages de définition des modèles d'information, appel ´ es aussi méta-modèles. Un méta-modèle définit la structure d'un modèle. Le niveau M3 est le niveau le plus abstrait parmi les quatre niveaux dans cette architecture. Il définit la structure de tous les méta-modèles du niveau M2 ainsi que lui-même. Il est composé d'une unique entité qui s'appelle le MOF. [6]

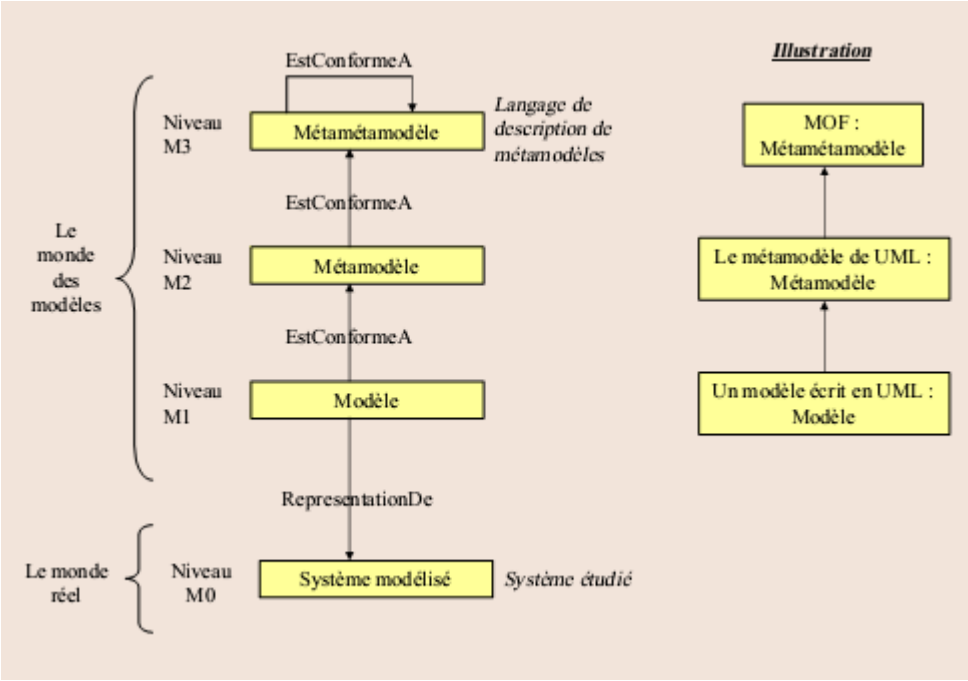


FIG 1.4 : Architecture à quatre niveaux. [2]

Dans l'architecture à quatre niveaux, la validation d'un modèle se fait avec le modèle du niveau suivant. Ce concept de méta-modèle nous semble fondamental. Il aide non seulement à réduire les ambiguïtés et les incohérences de la notation, mais il constitue aussi un précieux atout pour les concepteurs, dans le cadre d'une automatisation du processus de développement logiciel. L'inconvénient majeur de MOF par rapport à nos objectifs c'est qu'il n'offre pas des techniques de validation et des règles de passage d'un niveau à un autre. Dans notre approche, on s'intéresse uniquement aux trois premiers niveaux. M0 pour décrire l'existant. M1 pour spécifier l'existant avec UML et M2 pour spécifier le méta-modèle de M1.

2.1.3 Les standards de L'OMG :

L'OMG a déjà défini plusieurs standards pour le MDA : nous en dressons ici une liste des plus importants. [7]

2.1.3.1 MOF: Meta Object Facility

MOF pour (Meta-Object Facility) est un ensemble d'interfaces permettant de définir la syntaxe et la sémantique d'un langage de modélisation. Il a été créé par l'OMG afin de définir des méta-modèles et leurs modèles correspondants.

Il fait partie des standards définis par l'OMG et il peut être vu comme un sous-ensemble d'UML. Le MOF et l'UML constituent le cœur de l'approche MDA car ces deux standards permettent de créer des modèles technologiquement neutres. Le MOF spécifie la structure et la syntaxe de tous les méta-modèles comme UML, CWM (Common Warehouse Meta-model) et SPEM (Software Process Engineering Meta-model.

2.1.3.2 UML: Unified Modeling Language

La notation UML est décrite sous forme d'un ensemble de diagrammes. La première génération d'UML (UML 1.x), définit neuf diagrammes pour la spécification des applications. Dans UML 2.0, quatre nouveaux diagrammes ont été ajoutés : il s'agit des diagrammes de structure com-posite (Composite structure diagrams), les diagrammes de paquetages (Packages diagrams), les diagrammes de vue d'ensemble d'interaction (Interaction overview diagrams) et les diag-rammes de synchronisation (Timing diagrams). Les diagrammes UML sont regroupés dans deux classes principales :

– **Les diagrammes dynamiques** : regroupent les diagrammes de séquence, les diagrammes de communication (nouvelle appellation des diagrammes de collaboration d'UML), les diagrammes d'activités, les machines à états, les diagrammes de vue d'ensemble d'interaction, et les diagrammes de synchronisation.

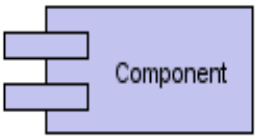
– Les diagrammes statiques : regroupent les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, et les diagrammes de paquetages.

Le diagramme des composants : dans notre étude en va concentrer sur l'architecture logicielle et les composantes logicielle alors en présente les diagrammes des composante généralement et plus de détail à chapitre III.

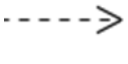
Les diagrammes des composants tombent sous la catégorie des diagrammes d'implémentation, un genre de diagramme qui modélise l'implémentation et le déploiement du système. Le diagramme des composants est principalement employé pour décrire les dépendances entre les divers composants logiciels tels que la dépendance entre les fichiers exécutables et les fichiers source. Cette information est semblable à celle contenue dans les fichiers *makefile*, qui décrivent des dépendances des codes sources qui doivent être employés pour compiler correctement une application.

✓ **Un Composant :**

Un composant représente une entité logicielle d'un système. (Fichier de code source, programmes, documents, fichiers de ressource .etc.). Un composant est représenté par une boîte rectangulaire, avec deux rectangles dépassant du côté gauche.



✓ **Dépendance :**

Une dépendance est utilisée pour modéliser la relation entre deux composantes 

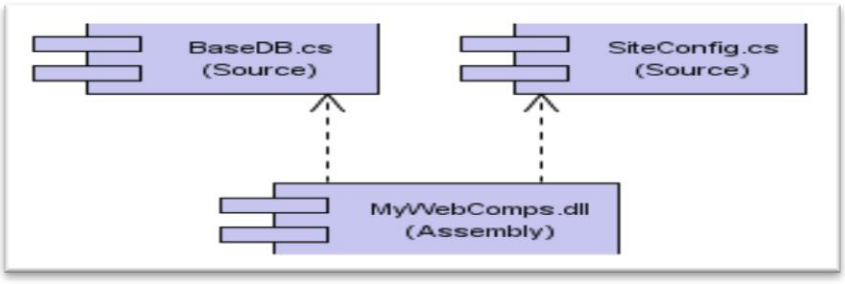


FIG 1.5: la représentation de diagramme des Composants

2.1.3.3 OCL : Object Constraint Language

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes souhaitées. Fort de ce constat, l'OMG a défini formellement le langage textuel de contraintes OCL, qui permet de définir n'importe quelle contrainte sur des modèles UML : le langage OCL

(Object Constraint Language) qui est un langage d'expression permettant de décrire des contraintes sur des modèles. Une contrainte est une restriction sur une ou plusieurs valeurs d'un modèle non représentable en UML.

2.1.3.4 XMI : Xml Metadata Interchange

XMI est le langage d'échange entre le monde des modèles et le monde XML (eXten-sible Markup Language). C'est le format d'échange standard entre les outils compatibles MDA. XMI décrit comment utiliser les balises XML pour représenter un modèle UML en XML. Cette représentation facilite les échanges de données entre les différents outils ou plateformes de modélisation. En effet, XMI définit des règles permettant de construire des DTD (Document Type Definition) et des schémas XML à partir de méta-modèles, et inversement. Ainsi, il est possible d'encoder un méta-modèle dans un document XML, mais aussi, à partir d'un document XML il est possible de reconstruire des méta-modèles. XMI a l'avantage de regrouper les méta-données et les instances dans un même document ce qui permet à une application de comprendre les instances grâce à leurs métadonnées.

Ceci facilite les échanges entre applications et certains outils pourront automatiser ces échanges en utilisant un moteur de transformation du type XSLT (Extensible Stylesheet Language Transformation).

3 La transformation de modèle

La deuxième problématique clé de l'IDM consiste à pouvoir rendre opérationnels les modèles à l'aide de transformations. Cette notion est au centre de l'approche MDA et plus généralement de celle des DSML. En effet, l'intérêt de transformer un modèle **Ma** en un modèle **Mb** que les méta-modèles respectifs MMa et MMb soient identiques (transformation endogène) ou différents (transformation exogène) apparaît comme primordial (génération de code, refactoring, migration technologique, etc.) [8]

3.1 Généralités

MDA étant basée sur la manipulation de modèles, le passage de l'un à l'autre (et vice versa) est une activité centrale de la méthode. La figure suivante présente les différentes opérations de transformation de modèles que l'on peut trouver dans MDA.

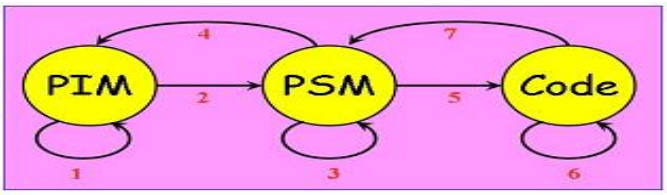


FIG 1.6 Opérations de transformation sur les modèles MDA

Le passage d'un modèle type UML à du code écrit en langage évolué (Java, C++,...), que l'on considère ici comme un modèle, est déjà bien implémenté dans les outils de modélisation (génération de squelette d'application).

Les transformations réflexives de modèles (PIM/PIM, PSM/PSM, Code/Code) seront effectuées soit lors d'ajouts d'éléments pour représenter des niveaux différents d'abstraction, soit pour optimiser un modèle lorsqu'une transformation non réflexive est incomplète ou pas assez précise (raffinage). Il est important de noter que ces transformations réflexives ne sont pas toujours automatisables. La transformation centrale de MDA (et aussi la plus délicate) est le passage du PIM vers le PSM Il est temps maintenant de s'y arrêter plus longuement. [3]

3.1.1 Le passage du PIM au PSM

Voici un schéma de cycle en Y pour obtenir un modèle de la plate-forme d'implémentation plus le passage du PIM vers PSM :

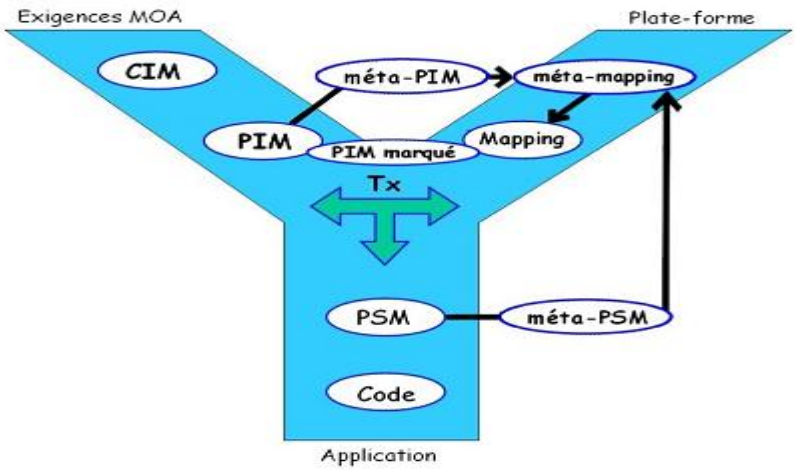


FIG 1.7 Le cycle en Y de MDA, adapté et complet

3.1.2 Exécution de la transformation

Toutes les informations sont alors réunies pour exécuter la transformation proprement dite.Selon le degré de précision et d'aboutissement des différents modèles construits lors des étapes précédentes (CIM, PIM, mapping) et de leur méta-modélisation (UML et profils, méta-

mapping), la transformation va avoir un degré d'aboutissement différent. Elle sera plus ou moins automatisable (donc outillable) et demandera plus ou moins d'interventions « manuelles »ensuite.

3.2 Standards et langages pour la transformation de modèle

De nombreux langages sont à ce jour disponible pour écrire des transformations de modèle de génération 3. On retrouve d'abord les langages généralistes qui s'appuient directement sur la représentation abstraite du modèle. On citera par exemple l'API d'EMF qui, couplée au langage Java, permet de manipuler un modèle sous la forme d'un graphe. Dans ce cas, c'est à la charge du programmeur de faire la recherche d'information dans le modèle, d'expliciter l'ordre d'application des règles, de gérer les éléments cibles construits, etc. [1]

3.2.1 ATL : ATLAS Transformation Language

ATL (ATLAS Transformation Language) est le langage de transformation développé dans le cadre du projet ATLAS, elle est développée au LINA à Nantes par l'équipe de Jean Bézivin , elle se compose : [9]

- ✓ d'un langage de transformation.
- ✓ d'un compilateur et d'une machine virtuelle.
- ✓ d'un IDE s'appuyant sur Eclipse.

3.2.1.1 Spécification du langage ATL

ATL a été conçu en fonction d'un certain nombre d'exigences que nous allons passer en revue. Certaines d'entre elles sont contradictoires avec les propriétés souhaitées pour QVT, mais elles seront adaptées lorsqu'une réponse définitive à la RFP sera promulguée.

- ✓ **Exigences :** Il s'agit de toutes les fonctionnalités ou propriétés souhaitées pour la langage ATL.

- la spécialisation et la composition de transformations.
- d'exprimer des séquences de transformations, des transformations atomiques et composites.
- d'utiliser la transformation d'ordre supérieur (HOT).
- de gérer la traçabilité en produisant pour chaque transformation, un modèle de traçabilité (mise en relation des éléments du modèle source avec ceux du modèle cible).
- la séparation entre spécification et implémentation d'une transformation.
- la détection de cas particuliers de transformation (i.e. endogène, exogène etc.).
- à l'utilisateur de guider la transformation et de vérifier la correction de celle-ci.
- principalement d'établir une librairie de transformations réutilisables et partagées au sein d'une large communauté.

3.2.1.2 Description du langage

ATL est un langage de transformation de modèles crée pour s'inscrire dans une approche MDA. Sa syntaxe abstraite a été décrite comme un méta-modèle MOF. Sa syntaxe concrète textuelle quant à elle, a été dénie en correspondance avec ce méta-modèle.

A- Transformations :

Un modèle de transformation ATL peut transformer un ensemble de modèles sources en un ensemble de modèles cibles, à la condition que les méta-modèles sources et cibles lui soient connus. Actuellement, ATL est capable de manipuler tous les modèles conformes à un méta-modèle MOF, ce qui implique aussi bien la transformation de méta-modèles que du méta-méta-modèle, grâce à la rétivité de celui-ci.

B- Les règles de transformation :

Il existe plusieurs types de règles de transformation selon la manière dont elles sont appelées et le type de résultat qu'elles retournent (FIG 9).

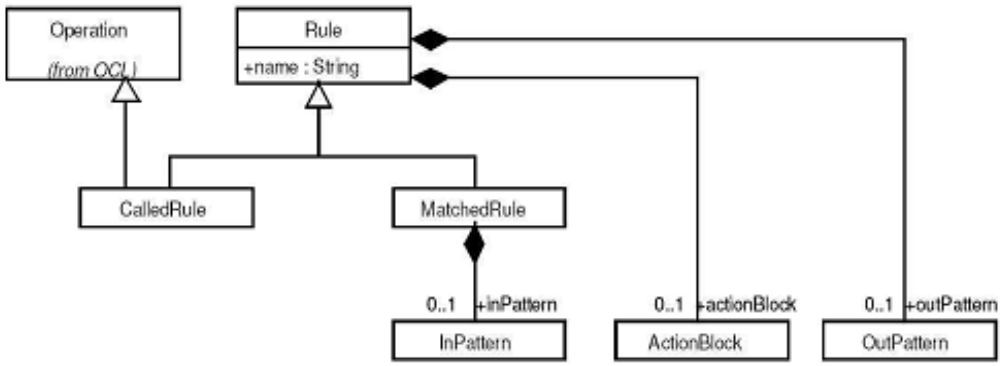


FIG 1.8 règles de transformation ATL

- ✓ **CalledRule** : règle appelée explicitement en utilisant son nom et en initialisant ses paramètres.
- ✓ **MatchedRule** : régle exécutée lorsque qu'un schéma type (InPattern) est reconnu dans le modèle source.

Le résultat d'une règle est soit un ensemble d'éléments de modèles prédéfini (OutPattern), soit un bloc d'instructions impératives (ActionBlock).

Une règle est dite déclarative s'il s'agit d'une MatchedRule et si son résultat est un ensemble d'éléments du modèle cible (OutPattern). Une règle de type CalledRule dont le résultat est un

bloc d'instructions est appelée Procédure. Le cas échéant, il s'agit d'une règle hybride (déclarative et impérative).

C- Pattern :

Il existe deux types de pattern utilisés par les règles de transformation

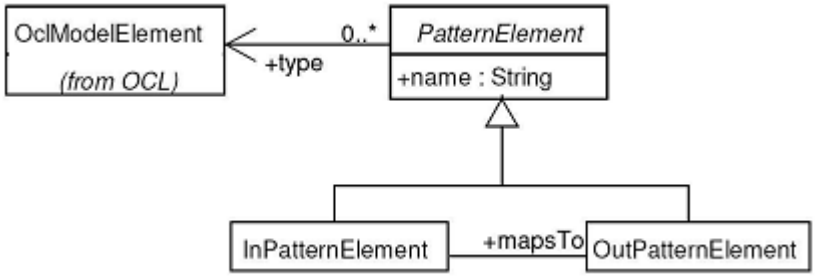


FIG 1.9 : Les Patterns cibles et sources des règles ATL

Les InPatterns, d'une part, utilisés pour la détection d'un ensemble d'éléments particulier dans le modèle source et permettant le déclenchement d'une MatchedRule. Il s'agit en fait, d'un ensemble de types issu du méta-modèle cible.

Les OutPatterns, d'autre part, sont dénis comme des ensembles de types issus du méta-modèle cible. Ils représentent les éléments de modèles résultant de l'application d'une règle de ransformation.

3.2.1.3 ADT : ATL Development Tooling

ADT est l'atelier de développement ATL créé par l'équipe ATLAS à l'université de Nantes. Il s'agit d'un IDE basé sur la plateforme Eclipse. Eclipse propose une architecture ouverte à base de plugins. L'idée était donc de se raccorder à la plateforme Eclipse en créant un plugin ADT. ADT utilise des fonctionnalités du plugin EMF développé par IBM.

Le but d'ADT est de fournir un environnement complet de développement de transformations de modèles en ATL. Afin d'aider le développeur de transformation dans son travail. ADT dans sa version actuelle fournit déjà des outils précieux pour le développeur :

- ✓ Un éditeur textuel fournissant une coloration syntaxique adaptable, et un mécanisme de reconnaissance d'accolades (indentation automatique). Cet éditeur est associé à une fenêtre a chant les propriétés de la transformation en cours de développement (content outline view).
- ✓ Une perspective complète éclipse (environnement graphique sous éclipse).
- ✓ Un compilateur, et un debuggé. La compilation fournit un rapport d'erreurs détaillé.
- ✓ Une aide à la création de projet ATL(Project Wizard).

Il reste encore quelques outils à implémenter notamment en ce qui concerne la prise en compte des rapports d'erreurs d'exécution. ATL s'accompagne donc, d'un effort de la part de l'équipe ATLAS, quant au développement d'interface et d'outils d'aide au développeur. En revanche l'installation d'ADT en tant que plug-in éclipse n'est pas réellement intuitive, et son utilisation nécessite un effort de prise en main conséquent.

4 Processus de Vérification en IDM :

Dans le contexte de l'ingénierie dirigée par les modèles, le développement des systèmes complexes fait appel à différentes techniques de modélisation qui dépendent:

- ✓ du domaine du système ou du sous-système.
- ✓ Des différentes phases du cycle de développement.
- ✓ Des différents niveaux d'abstraction aux quels le système est étudié.
- ✓ Des différents aspects spécifiques à analyser et à vérifier lors de la conception.

Dans le but d'analyser le comportement modélisé pour assurer le bon fonctionnement du système et détecter d'éventuels problèmes, différentes techniques telles que la simulation, le test ou les méthodes formelles sont alors utilisées dès les premières étapes de sa conception. Dans ces techniques, les modèles peuvent être utilisés comme support pour ces activités de vérification et de validation. Certaines analyses peuvent être conduites directement en utilisant les modèles. Par exemple, des simulations ou des tests sur des modèles comportementaux sont souvent favorisés. En effet, il s'agit d'une méthode simple et relativement peu coûteuse, consistant à générer des cas de tests pertinents pour vérifier les scénarios attendus. En revanche, même pour un système d'une taille raisonnable, l'analyse des comportements possibles ne peut pas être exhaustive et l'on court le risque de ne pas identifier des situations potentiellement dangereuses pour le système.

D'autres analyses nécessitent la transformation des modèles réalisés dans des formalismes formels adaptés à un type de vérification formelle désirée. Les méthodes formelles, telles que la preuve de théorème, les réseaux de Petri ou le Model-Checking, s'appuient sur un cadre mathématique précis et non ambigu permettant de modéliser à la fois le système et les propriétés qu'il doit vérifier. Par exemple, la déduction d'un réseau de Petri global modélisant le comportement du système à partir des modèles comportementales permet de mettre en œuvre les outils de preuve formelle associés aux réseaux de Pétri. [1]

Conclusion

Nous avons introduit dans ce chapitre les principes généraux de l'IDM, c'est-à-dire la méta-modélisation d'une part et la transformation de modèle d'autre part. Ces deux axes constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement.

Les premiers résultats de la méta-modélisation ont permis d'établir les concepts (modèle et méta-modèle) et relations (représentation De et conforme A) de base dans une architecture dirigée par les modèles. Malgré tout, les modèles sont actuellement construits à partir de DSML décrits principalement par leur structure. Nous avons vu que si la définition des syntaxes abstraites et concrètes était maîtrisée et outillée. Il est aussi possible de vérifier structurellement la conformité d'un modèle par rapport à son DSML.

La technique de transformation de modèle est la clé du succès de l'IDM du succès de l'IDM afin de pouvoir rendre opérationnelle la manipulation du modèle. Cependant, les travaux récents de normalisation et d'implantation d'outils ainsi que les nouveaux principes de l'IDM ont permis de faire évoluer ces techniques. Les transformations s'expriment maintenant directement entre les syntaxes abstraites des différents DSML et permettent ainsi de se concentrer sur les concepts alors en abstraction. Ainsi, des travaux sont encore nécessaires afin de formaliser ces techniques et pouvoir ainsi valider et vérifier les transformations écrites. Par exemple, les transformations permettant de générer du code à partir d'un modèle sont assimilables à une compilation qu'il est indispensable dans certains cas de certifier. D'autre part, si QVT offre un cadre très large pour la description de transformation de modèle en couvrant une large partie des approches possibles, ce n'est généralement pas le cas des implantations actuellement disponibles. En effet, la plupart des langages n'implantent qu'une partie du standard QVT, prévu grâce aux niveaux de conformité définis par l'OMG.

CHAPITRE 2

LES OUTILS ET PLUGINS ECLIPSE SUPPORTE POUR MODELISATION ET LA TRANSFORMATION DU MODELES.

Dans ce chapitre nous présentons le standard EMF, les outils et les plugins **Eclipse** qui supportent l'approche MDA.

1 les plugins éclipse supportés pour la modélisation et la transformation de modèles :

Le principe de base du Framework Eclipse est d'être conçu pour être extensible. Au cœur d'Eclipse se trouve les mécanismes permettant la découverte et le chargement de modules : les plugins. Eclipse et l'ensemble des sous-projets sont construits sous-forme de plugins. Le noyau d'Eclipse gère le cycle de vie des plugins (découverte, chargement, mise à jour, déchargement…). Eclipse ajoute à des fonctionnalités permettant de gérer la coopération entre plugins : les plugins peuvent déclarer des points de branchements (les points d'extension) et peuvent aussi enrichir un plugin existant en se branchant sur l'un de ses points d'extension. Cette coopération se déclare par l'intermédiaire d'un fichier XML propre à chaque plugin.

1.1 Eclipse : généralités

Le projet Eclipse a été créé en 2001 par IBM qui a fait don du code initial. Dès le lancement du projet, IBM a joué la carte des partenariats en constituant un consortium de sept sociétés (dont Borland). Jusqu'en 2004, l'organisation en consortium donnait à IBM un pouvoir important sur le projet. [10]

La fondation est une structure indépendante régie par des règles clairement formalisées.

Les Projets : les projets développés dans le cadre de la fondation obéissent à des règles clairement formalisées et sont organisées en plusieurs catégories. Ces catégories sont nommées 'Top-Level projects', elles correspondent à des projets principaux découpés en sous-projets.
Début 2006, ces projets sont au nombre de 10 :

- **Eclipse** : développement du socle et de l'outillage Java.
- **Eclipse Tools** : divers sous-projets pouvant servir de socle à d'autres projets ou bien inclassables dans les autres catégories.
- **Web Tools Platform (WTP)** : outils de développement Web et J2EE.

- **Test and Performance Tools Platform (TPTP)** : outillage de test et de mesure de performance.
- **Data Tools Platform (DTP)** : outils de manipulation de structures de données.
- **Device Software Development Platform (DSDP)** : outils de développement pour les systèmes embarqués.
- **SOA Tools Platform (STP)** : outillage pour la mise en œuvre d'architectures orientées services.
- **Modeling** : divers sous-projets concernant la modélisation.
- **Technology project** : rassemble une vingtaine de sous-projets aux objectifs très variés abordant des sujets innovants. Les sous-projets arrivant à maturité sont amenés à rejoindre l'un des autres projets principaux.

Le but initial du projet Eclipse était de fournir un socle, écrit en Java, pour la création d'environnements de développement. Depuis 2004, cet objectif a été étendu en prenant en compte l'utilisation du Framework Eclipse pour tous les types d'applications.

Le socle pour la création d'environnement de développement est implémenté dans le cadre du sous-projet nommé « **Eclipse Platform** ».

2 EMF : Eclipce Modiling Framework :

Eclipse Modeling Framework est la partie Model du pattern MVC (Model-Vue-controler) (à noter que le framework ne propose pas de visuel pour représenter le modèle). Le modèle peut être persisté sous différentes manières : XSL, fichiers java avec annotations, XMI, puis donne la possibilité de rajouter son système de persistance. A noter qu'EMF gère la persistance sous forme de plusieurs fichiers ressources reliées, et qu'en implémentant son propre système de persistance, vous ne perdez pas cet atout [9].

Cette définition « à la » *Wikipédia,* si elle souffre de quelques manques, permet une entrée en matière :

- EMF est un « framework » qui traite des modèles: cela peut s'entendre ici sous le sens que EMF offre à ces utilisateurs un cadre de travail pour la manipulation des modèles (sous-entendu, à propos des applications informatiques, credo de *Eclipse*),
- EMF permet de stocker les modèles sous forme de fichier pour en assurer la persistance,
- EMF permet de traiter différents types de fichiers: conformes à des standards reconnus (XML, XMI) et aussi sous des formes spécifiques (code Java) ou tout

simplement sur mesure (au bon gré du concepteur),

- EMF ne propose pas d'outil graphique (de dessin) pour la modélisation.

Mais cette définition ne dit rien sur ce dont est capable EMF, sur la manière dont il y arrive et sur sa mise en œuvre à travers *Eclipse*. C'est ce que nous allons tenter d'éclairer dans cette section.

2.1 Objectif d'EMF

L'objectif est de montrer comment il est possible d'utiliser les langages de programmation objet pour coder des opérations sur les modèles, telles que la génération de code et de documentation ou la transformation de modèles.

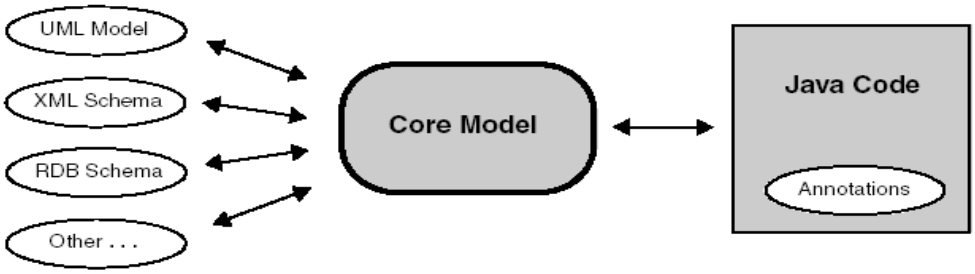


FIG 2.1 : Organisation générale d'EMF.

Comme le montre la figure ci-dessus, l'objectif général de EMF est de proposer un outillage qui permet de passer du modèle au code *Java* automatiquement. Pour cela le *framework* s'articule autour d'un modèle (le *Core Model*).

EMF va proposer plusieurs services :

1. la transformation des modèles d'entrées (à gauche sur la figure 15), présentés sous diverse formes, en *Core Model*,
2. la gestion de la persistance du *Core Model*,
3. la transformation du *Core Model* en code Java.

Les doubles flèches symbolisent que les transformations inverses (ou les « imports/exports ») Sont aussi gérées par EMF.

En cela, certains considèrent que EMF est à la fois :

- Un outil de réunification de divers standards de modèles (XMI, UML, code Java - si du moins on considère ce dernier comme un modèle),
- Un pont entre deux mondes du génie logiciel (celui des gourous de la modélisation

et celui des partisans du code avant tout) en prenant une position médiane et en prenant le meilleur de chaque monde,..... et tout cela pour un coût minimum. Le rêve en quelque sorte ! [11]

2.2 Les formats d'entrée standards

EMF peut de base gérer en entrée des modèles présentés sous trois formats : UML, XMI et en code Java Annoté. [12]

2.2.1 UML :

Pour cette option, il existe trois possibilités :

- ✓ **L'édition directe conformément au méta-modèle Ecore**

Il s'agit là, d'éditer des modèles graphiques UML conformément au méta-modèle *Ecore.* Cela sous-entent l'utilisation d'un outil de modélisation qui en soit capable[4]. Par voie de conséquence, la transformation d'entrée du modèle n'est plus : on dispose du *Core Model* sous le bon format.

- ✓ **L'importation de modéle UML**

Il s'agit d'importer, à l'aide de la fonction ad hoc de EMF, un modèle dans son format natif (qui dépend de l'outil de modélisation utilisé). Seul le format IBM Rational (*.mdl*) permet de profiter de cet avantage. En effet, la gamme Rational et *Eclipse* sont des projets «frères» donc « génétiquement » compatibles.

- ✓ **L'exportation de modèles UML**

C'est à peu près le même principe que l'option d'importation, sauf que la conversion du format natif en Ecore ne se fait pas avec EMF, mais avec l'outil de modélisation d'origine.

À propos des trois options :

- La première option ci-dessus présente l'avantage d'être simple et directe, de ne pas nécessiter d'opération d'importation ou d'exportation. Aucune synchronisation entre l'outil de modélisation natif et EMF n'est nécessaire.
- Les deux autres options (équivalentes du point de vue de la conversion) offre l'avantage que l'outil de modélisation peut servir à autre chose que la « simple » modélisation. Par exemple, il peut proposer sa propre fonction de génération de code. Il donne aussi la possibilité de créer son propre *Core Model*, de le transformer en Ecore pour l'utiliser ainsi par la suite.

2.2.2 XMI

Ce format de fichier, standard de l'OMG (*Object Managment Group*), est utilisé conjointement à UML :

- UML se charge de décrire les contenus des modèles,
- XMI se charge de formater ces contenus pour permettre de leur assurer une persistance standardisée.

Malgré quelques problèmes de maturité (en phase de résolution) qui demandent une attention particulière quant à l'association de différentes versions de UML avec les différentes versions de XMI [BLANC 2005], ce format tend à devenir le standard pour l'échange de données (et notamment des modèles) entre les différents outils du génie logiciel.

La figure 15 ne fait pas directement référence à ce format, mais il est permit de penser que ce choix est pertinent pour les quelques raisons suivantes :

- Comme nous l'avons vu ci-dessus, IL tend à devenir un standard, du moins pour le développement orienté objet (aussi le credo de Java),
- IL est le standard utilisé par Ecore pour sa propre persistance,
- tout cela concoure à combler le fossé qui existe entre les modèles UML et les fichiers de code Java.

2.2.3 Java Annoté

Une des solutions tentantes pour modéliser les classes, qui vont être concrétisées par une application Java, est d'utiliser les interfaces Java :

- Elles n'implémentent pas les méthodes: on s'abstrait donc de cette implémentation,
- Les méthodes *get /set* peuvent être utilisées pour modéliser les attributs,
- une classe pourra implémenter plusieurs interfaces, ce qui est une manière détournée d'autoriser l'héritage multiple (impossible en Java de classe à classe et possible en UML),
- Cela permet une évolution « douce » vers la modélisation des plus irréductibles codeurs.

Les annotations sont des *tags @model* placés dans la *java doc* des interfaces. Ces *tags*, et leurs attributs éventuels, sont détectés par EMF qui considère ainsi les entités concernées (interfaces, méthodes) comme des éléments de modélisation.

2.3 Le (méta ?) méta-modèle pivot : Ecore

Nous avons vu que le *Core Model*, pivot des transformations possibles avec EMF, doit pouvoir modéliser les correspondances entre plusieurs types de modèles. On dirait, pour être puriste, que ces différents modèles sont conformes à autant de méta-modèles.

Le tableau ci-dessous synthétise les cas en présence pour EMF.

Modèles (M1)	**Méta-modèles (M2)**	**Méta-méta-modèles (M3)**
Diagramme de classes (entrée)	UML	MOF
Fichier XML (entrée)	XMI	MOF
Ensemble d'interfaces Java (entrée)	Java Annoté	EBNF
Programme Java (sortie)	Java	EBNF

Table 2.1 : Les espaces techniques de modélisation à fédérer.

Pour être fidèle à l'IDM, les transformations sont définies au niveau des méta-modèles. Cette stratégie permet de modéliser les règles de transformation (il faut là aussi un méta-modèle), et ainsi de capitaliser les efforts faits pour leur définition.

Le tableau ci-dessous représente l'espace technique standard d'EMF.

Modèle (M1)	**Méta-modèle (M2)**	**Méta-méta-modèle (M3)**
MEcore (pivot)	MMEcore	Ecore

Table 2.2 : L'espace technique de modélisation fédérateur.

Ecore est un méta-méta-modèle très proche de MOF. Il est en fait un sous ensemble de MOF, Il restreint celui-ci. En effet, une des particularités de Ecore est qu'il accepte des méta-classes (dans le niveau *M2).* sans associations. Pour associer deux classes, il faudra stéréotyper un attribut comme étant une association. Cette possibilité à été mise au point car en langage Java le concept d'association n'existe pas .

En Java, les associations d'un diagramme de classes UML s'implémentent par la création d'un attribut ayant pour type la classe partenaire. L'attribut doit être créé dans l'une, dans l'autre ou dans les deux classes partenaires (dans le cas d'une association binaire) selon la navigabilité de l'association. Au niveau du méta-modèle l'on définira les caractéristiques du modèle de niveau M1. Par exemple, on définira les concepts d'un diagramme de classes UML si c'est cela

que l'on veut traiter.

En l'occurrence, la seule entrée UML possible est le diagramme de classe, mais on peut imaginer de méta-modéliser d'autres diagrammes (cas d'utilisation, séquence,...): cela permettrait d'utiliser EMF à d'autres stades du cycle de développement (transformation CIM vers PIM de MDA).

2.4 Génération du code

Il est temps maintenant d'observer ce que va produire EMF : le code. C'est bien là son objectif premier. EMF répond bien à son objectif d'améliorer la productivité du développement d'application. Il y arrive en automatisant la génération du code à partir du modèle. Effectivement, une fois le modèle créé, « quelques clics » suffisent à cette génération

Nous n'allons pas ici entrer dans une analyse détaillée du code contenu dans les éléments générés nous contenterons, dans cette introduction à EMF, d'une liste (non exhaustive) de ce qui est généré. Une analyse détaillée demanderait, au préalable, une étude plus approfondie d'EMF et d'Ecore.

2.4.1 Organisation du code généré

Un choix de conception a été fait par ses concepteurs et est imposé par EMF : la séparation interface/implémentation dans le code généré.

Cela va se concrétiser par la génération de deux ensembles (issus du modèle d'entrée, assimilable à un diagramme de classes UML) :

- un ensemble d'interfaces Java,
- un ensemble de classes implémentant ces interfaces.

Les raisons principales qui justifient ce choix sont : la correspondance avec un *pattern* utilisé par de nombreuses API, la nécessité de pouvoir disposer d'un héritage multiple (impossible en Java sans la notion d'interface).

En plus des classes correspondant au modèle d'entrée, EMF génère deux autres éléments importants : une interface *Factory* et une interface *Package* ainsi que leurs classes d'implémentation.

La 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*.

LePacketage : Cette classe apporte des facilités pour accéder aux méta-données Ecore du modèle. Il contient des accesseurs aux *EClasses*, EA*ttributes* et *EReferences*[9] implémentées dans le modèle par exemple.

2.4.2 Le modèle générateur

En plus du modèle conforme au méta-méta-modèle Ecore, EMF utilise un modèle dit générateur (fichier d'extension *.genmodel*). Ce modèle, comme le modèle Ecore, est généré automatiquement (donc de manière transparente pour l'utilisateur) lors de la transformation du modèle d'entrée en modèle Ecore.

La plupart des informations nécessaires sont contenues dans le modèle « core » : le nom des classes les attributs, les références,... Mais un certain nombre d'informations n'y sont pas telles que : les règles de préfixation du nom des classes, où mettre le code généré,...

Ces informations de paramétrage de la génération seront stockées dans le modèle générateur.

L'avantage de cette séparation (du générateur et du « core ») est que le méta-méta-modèle Ecore reste indépendant de toutes informations relevant de la stricte génération du code. L'inconvénient est qu'il faut assurer une synchronisation des deux modèles en cas de modification (pour en garder la cohérence). EMF assuré automatiquement cette synchronisation.

2.5 EMF et les standards OMG

Des discussions ont cours à propos des relations qu'entretient EMF avec les différents standards de l'OMG que sont UML, XMI, MOF et MDA.

Ce paragraphe est de pure culture, en faire l'impasse ne nuira pas à la compréhension et l'expérimentation de EMF. [13]

2.5.1 Pour UML

UML est un méta-modèle très utilisé pour modéliser les applications du monde objet. Les différents diagrammes conformes à UML sont prévus pour permettre autant de représentations d'une même application. Entre autres :

- La vue utilisateur: les cas d'utilisation,
- La vue dynamique: le diagramme de séquence,
- La vue architecturale: le diagramme de composants,
- La vue statique de conception: le diagramme de classes.

C'est sur cette dernière qu'intervient actuellement EMF : c'est le diagramme de classes qui sera

utilisé pour générer le code (Java en l'occurrence).

2.5.2 Pour MOF

MOF (*Meta Object Facility)* est le méta-méta-modèle standard de l'OMG. Il est utilisé pour définir les méta-modèles promus par cette organisation. Citons les plus caractéristiques de l'IDM : UML, QVT, CWM (*Commun Warehouse Metamodel*).

Ecore et MOF ont de nombreuses similitudes. Les différences se situent au niveau de la couverture des différents concepts tels que ceux de classes, de types de données, d'attributs, de relations entre paquetages et classes (voir § 4 de cette partie).

L'on peut aussi noter que le projet EMF, enfant du projet *Eclipse*, et son retour d'expérience ont une influence non négligeable sur les travaux de standardisation de MOF et/ou UML.

2.5.3 Pour XMI

XMI est un standard créé par l'OMG basé sur XML. Ce dernier est lui-même un standard porté par le W3C (*World Wide Web Consortium*).

Ce standard a été créé pour faciliter la sérialisation et les échanges de données, dans le cadre de la modélisation, entre les différents outils qui interviennent dans le cycle de vie d'une application informatique.

Il s'appuie sur les mécanisme de DTD (*Document Type Definition*) ou de schéma XML pour définir les structurations de balises nécessaires et suffisantes à la représentation des modèles MOF au format XML [ ].

XMI peut être utilisé pour sérialiser toute sortes de modèles utilisés par EMF, il est aussi utilisé pour le méta-méta-modèle Ecore lui-même et comme forme canonique des fichiers « Ecore » (*.ecore*).

2.5.4 Pour MDA

Model Driven Architecture est une démarche d'ingénierie promue par l'OMG. Elle est basée sur la manipulation de différents modèles représentant l'application cible (indépendant de l'informatisation, indépendant de la plate-forme d'exécution, spécifique à cette plate-forme, de la plate-forme elle-même, le code) et, par voie de conséquence, sur des transformations de modèles.

EMF est bien dans la philosophie de cette démarche et peut s'intégrer dans l'outillage nécessaire comme générateur de code à partir de modèles (ce qui est l'objectif premier de EMF).

2.6 Autres services proposés par EMF

En plus d'être un outil d'amélioration de la productivité, EMF a d'autres apports, tels que : la notification de modification du modèle, la gestion de la persistance par une sérialisation XMI, une API réflective pour la manipulation générique des objets EMF, l'EMF dynamique. La combinaison de tous ces services (et d'autres non cités) fait qu'EMF offre un socle pour l'inter-opérabilité avec d'autres outils et applications basés sur EMF.

Nous ferons ici une simple évocation de ces possibilités. Pour en savoir plus, je renvoie le lecteur aux différentes documentations.

2.6.1 La notification et les Adaptateurs

Lors de l'observation du code des classes d'implémentation générées par EMF, on trouve dans les accesseurs de type *set* le code ci-dessous:

```
public void setNom(String newNom) { String
      oldNom = nom;
      nom = newNom;
      if (eNotificationRequired())
            eNotify(new
            ENotificationImpl(this,
            Notification.SET,
            BibliothequePackage.AUTEUR
            NOM, oldNom, nom));
}
```

Le test conditionnel permet de vérifier si une notification de modification a été demandée sur l'élément (ici un attribut) et, le cas échéant, de l'opérer.

Cette possibilité d'EMF est-il intéressante : l'on va pouvoir transmettre automatiquement aux objets dépendants une modification faite sur un attribut ou une référence.

Dans EMF les observateurs de notification (les *listener* Java) son appelés *Adaptaters* (Adaptateurs). En effet, en plus de leur statut d'observateur, ils peuvent modifier les comportements des objets auxquels ils sont attachés.

2.6.2 L'API réflective

Comme tous les éléments des modèles EMF héritent de la classe *EObject* du méta-méta-modèle Ecore, l'utilisateur peut se servir de l'API réflective pour manipuler leurs instances.

Il pourra, par exemple, accéder directement à la valeur d'un attribut d'objet sans faire appel aux accesseurs (*get*, *set*) de sa classe. Des méthodes d'accès plus génériques existent.

Bien que cette technique d'accès est moins performante que l'accès direct par les accesseurs des classes, elle permet une ouverture vers l'extérieur des modèles grâce à la généricité des méthodes de l'API réflective.

2.6.3 L'EMF dynamique

Jusqu'à présent nous avons utilisé EMF pour générer une implémentation de nos modèles. Dans de nombreux cas, nous ne désirerons pas implémenter le modèle Ecore de notre application. Il suffit pour une observation de l'architecture.

La particularité de l'API réflective est qu'elle peut générer dynamiquement les classes nécessaires (et uniquement celles-là) lors de l'utilisation d'une de ses méthodes.

En fait, le modèle Ecore est le seul nécessaire pour pouvoir utiliser l'application : seul un allongement du temps d'accès sera perçu.

3 EMF Tiger

EMF Tiger est un environnement de l'outil pour la transformation de modèles de champs électromagnétiques. Les transformations sont définies et exécutées directement sur les modèles EMF, assurer la sécurité et l'efficacité de type. Transformation du modèle peut être exécuté soit à l'aide du code généré dans un mode ou d'un interprète.

Le langage de transformation basé sur des règles de Tiger EMF est inspiré par les concepts de transformation de graphes et combine deux concepts déclaratifs et procéduraux. En particulier, les règles de transformation sont déclaratives dans le sens où un modèle structural est utilisé pour définir la condition sine qua non d'une règle. Concepts de procédure, c'est-à-strutures de contrôle de flux tels que les couches et les boucles peuvent être utilisés pour appliquer un ensemble de règles de transformation d'une manière contrôlée. [14]

4 GMF & GEF: Graphic Modeling Framework ET Graphical Editing Framework:

L'utilisation de deux EMF et GEF pour la construction d'une fonctionnalité basée sur Eclipse est assez fréquent. Beaucoup de références ci-dessous fournissent des informations sur la façon d'utiliser ces cadres ensemble, dont certaines inspirées du projet GMF lui-même. Avant de plonger dans un nouveau projet GMF, nous allons explorer un peu de la façon dont GMF rapproche de la tâche d'utiliser EMF et GEF de façon générative.

Le projet GMF a adopté le terme «tools mith» pour désigner les développeurs qui utilisent GMF pour construire des plug-ins, tandis que «praticien» est utilisé pour référer à ceux qui utilisent

ladite plug-ins, et qui peuvent également être des développeurs. D'un point de vue ergonomie, le nombre et les types de «modèles» utilisés par le FMV interne doit être caché dans la mesure du possible. Cependant, il est probable que la plupart des tools miths sont intéressés à savoir ce qui se passe sous les couvertures, ainsi qu'une description de chaque modèle est lié à la Documentation GMF page. [15]

5 Topcased

TOPCASED est un logiciel d'ingénierie assistée par ordinateur. Il contient un IDE basé sur le framework de la plateforme de développement Eclipse, à laquelle il ajoute des fonctionnalités essentiellement liées à la mise en œuvre de la première branche du cycle en V pour l'ingénierie du logiciel, du matériel ou de systèmes mixtes logiciel/matériel.

Sont implémentés (ou en cours d'implémentation) des moyens d'analyse d'exigences, modélisation, simulation de modèles, implémentation, test, validation, rétro-ingénierie, génération de code, de modèles et de documentation et gestion de projet.

S'appuyant principalement sur des langages standardisés pour la modélisation du logiciel (UML, SysML, AADL...), TOPCASED travaille avec des fichiers XMI. Tous ces standards sont implémentés dans leurs dernières versions stables, soit directement par le projet TOPCASED, soit par les modules de la dernière version stable de la plate-forme Eclipse. TOPCASED UML est ainsi le modeleur UML le plus complet parmi les solutions gratuites, et le plus respectueux des standards actuels. [16]

6 ATOM3

AToM3 (A Tool for Multi-formalism and Meta-Modelling [Vangheluwe, 2002] est un outil de modélisation multi-paradigmes développé par le laboratoire MSDL (Modelling, Simulation and Design Lab.) à l'université de McGill Montréal, Canada. Cet outil a été conçu en collaboration avec Juan de Lara de Universidad Autónoma de Madrid (UAM), Espagne.

Les deux principales fonctionnalités d'AToM3 sont **la méta-modélisation** et la **transformation de modèles**. La méta-modélisation est la description ou la modélisation de différents types de formalismes. La transformation de modèles est une technique consistant à transformer, traduire ou modéliser automatiquement un modèle décrit dans un certain formalisme, vers un autre modèle qui peut être décrit soit dans le même formalisme soit dans un autre. Il permet aussi de définir la syntaxe abstraite et concrète des langages visuels. [17]

Conclusion

Nous avons abordé dans ce chapitre le standard EMF (Eclipse Modeling Framework) et les différents outils de transformation de modèle. Nous pouvons conclu que l'approche MDA est devenue plus maitrisée du fait de l'apparition de ce nombre important d'outils et de standards qu'ils utilisent.

Le "**modèle**" est l'unité de traitement principale sur laquelle s'articulent toutes ces techniques l'MDA, l'IDM etc. Cependant, un nombre étonnant de concepts et de définitions sont apparu dans la littérature dans ce domaine. Ainsi, plusieurs niveaux d'abstractions. On doit toujours répondre à une question :

- On doit aller de quoi à quoi ?
- Nous somme dans quel niveau ?
- Que représente un modèle dans chaque niveau ?
- ….

Pour répondre à toutes ces questions on a abordé tous les niveaux d'abstraction du l'**objet** jusqu'à le méta-méta-modèle en passant du la notion méta-modèle.

Le chapitre suivant décrit un nouveau champ de travail **"Les architecture Logicielles".** Nous cherchons dans ce chapitre d'utiliser les différents concepts étudiés dans le chapitre précédent et plus précisément le modèle, la méta-modélisation et la transformation de modèle.

CHAPITRE 3

L'ARCHITECTURE DES LOGICIELS ET LA DESCRIPTION D'ARCHITECTURE.

L'architecture logicielle occupe une position centrale dans le processus de développement des systèmes complexes. Vu le rôle important que l'architecture joue dans le développement de systèmes complexes, il est devenu indispensable de disposer des méthodologies formelles ou semi-formelles et de bénéficier d'outils de support pour valider et analyser les modèles.

Dans ce chapitre, nous présentons l'architecture logicielle et le langage de description les plus connus comme : Wright, Acme, Darwin, Archjava, Fractal et ses outils de support. On introduit ensuite, le standard XMI, qui permet de représenter les modèles d'architectures sous forme XML, et qui favorisent leurs échanges et leurs pérennités.

1 Les concepts de base d'architecture logicielle

Nous intéressons dans cette section aux concepts d'architecture logicielle. L'objectif n'est pas de proposer une nouvelle définition ni de comparer les définitions proposées, mais plutôt de s'interroger sur les concepts eux-mêmes.

1.1 Architecture logicielle

Une architecture logicielle est définie comme un niveau de conception qui contient la description des composants à partir desquels un système est construit, les spécifications comportementales de ces composants, les modèles et les mécanismes de leurs interactions (connecteurs) et enfin un modèle définissant la topologie (configuration) d'un système. Dans le chapitre IV présente un méta-modèle montrant les concepts de base des architectures logicielles ainsi que leurs relations [18].

1.2 Composant

Un composant est une entité de calcul ou de stockage de données spécifiant, par contrat, ses interfaces (fournies et requises). Un composant logiciel peut être déployé indépendamment et peut être sujet de composition par un tiers pour la conception d'applications logicielles [19].*"A Software component is a unit of computation with contractually specified interfaces. A software component can be deployed independently and is subject to composition by third parties".*

De cette définition, il résulte que :

- Un composant est une unité de composition spécifiant, par contrat, ses interfaces (fournies et requises),
- Un composant logiciel peut être déployé indépendamment (installation sur différentes plates-formes, collaboration et coopération avec d'autres composants),
- Un composant peut être typé, et peut avoir une sémantique formelle ou informelle. Il peut exporter des contraintes d'utilisation et présenter des propriétés fonctionnelles ou non fonctionnelles.

Un composant logiciel possède, principalement, les trois éléments suivants [20]:

• ***L'interface*** **:** est la partie visible d'un composant. L'interface d'un composant consiste en un ensemble de points d'interaction entre le composant et son environnement qui permettent l'invocation des services. . Deux types de ports peuvent être distingués : les ports *services* (*fournis*), qui exportent les services des composants et les ports *besoins* (*requis*), qui importent les services vers les composants.

• ***Les propriétés***: généralement, ce sont des attributs. Elles permettent d'adapter et d'évoluer des composants par des modifications de certaines propriétés (interface, comportement). Il existe deux types de propriétés : les propriétés fonctionnelles et les propriétés non fonctionnelles. Les propriétés fonctionnelles concernant la sémantique des fonctions des composants alors que les propriétés non fonctionnelles représentent d'autres besoins tels que la sécurité, la portabilité et la performance. Elle peut être configurable en fonction du contexte d'exécution.

• ***Les contraintes*** : propriétés spécifiques qui permettent de spécifier des restrictions sur les éléments architecturaux sur lesquels elles s'appliquent. Elles doivent être spécifiés afin de représenter les utilisations prévues d'un composant et d'établir les dépendances parmi ses éléments internes ; contraintes qui peuvent être, la sécurité, la persistance ou les transactions, etc...

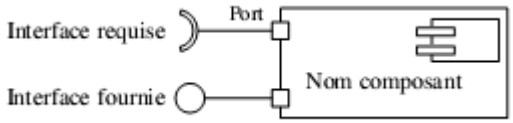


FIG 3.1 Représentation graphique d'un composant

1.2.1 Les trois dimensions d'un composant

Un composant peut être de deux natures :

- *produit* : il s'agit d'une entité *« building block »* autonome passive (entité logicielle ou entité conceptuelle) qu'il est possible d'adapter. Les composants logiciels et les bibliothèques de fonctions mathématiques en font partie.
- *processus* : un composant processus correspond à une suite d'actions qu'il faut réutiliser pour obtenir un produit final. Ces actions sont souvent encapsulées dans un processeur (unité de traitement). Un composant processus possède en général des fragments de démarche.

Au regard des travaux existants un composant produit ou processus doit refléter les trois dimensions qui sont le niveau d'abstraction, le mode d'expression et le domaine [21].

La première dimension ou *niveau d'abstraction* permet d'exprimer les degrés de raffinement d'un composant et ce depuis sa spécification. Elle est considérée comme étant le plus haut niveau d'abstraction.

La deuxième dimension ou *mode d'expression* permet de décrire les différents modèles de représentation d'un composant (représentation textuelle, graphique, flot de données, implémentation).

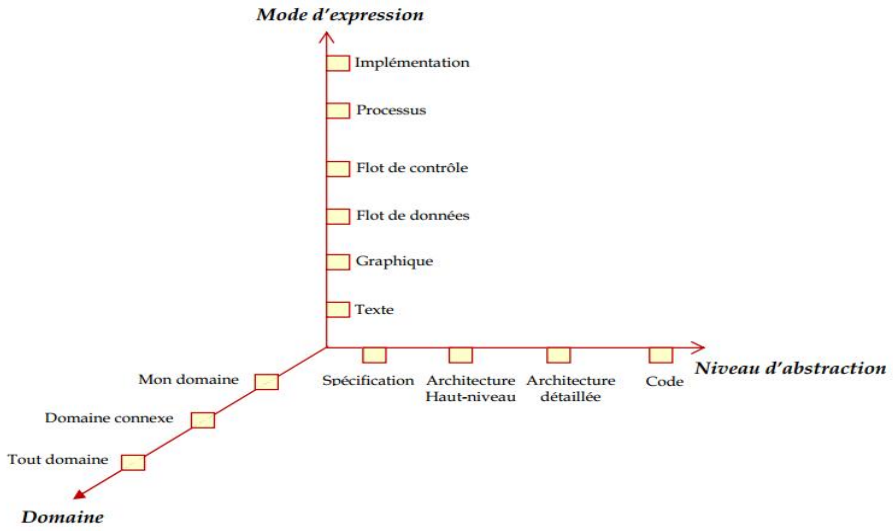


FIG 3.2 *Les trois dimensions d'un composant.*

1.3 Connecteur

Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions (transfert de contrôle et de données) entre les composants. Ils contiennent des informations concernant les règles d'interaction entre les composants. Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants. En effet, la raison de l'existence des connecteurs est de faciliter le développement d'applications à base de composants logiciels.

De nouveaux connecteurs peuvent être spécifiés de la même façon que les composants. Par exemple, on pourrait spécifier des connecteurs pour implémenter un protocole de communication spécifique. Medvidovic a classé les services d'interaction offerts par les connecteurs en quatre types. Chaque type de connecteur offre un ou plusieurs services d'interaction. Ces services sont les suivants :[22]

- **Le service de communication :** un connecteur assure ce service s'il s'occupe des transmissions de données entre composants.
- **Le service de coordination :** supporte le transfert de contrôle entre composants. Les appels de fonctions sont un exemple de cette catégorie de connecteurs.
- **Le service de conversion :** convertit les interactions inter-composant si nécessaire. Il permet aux composants hétérogènes d'interagir. Les services de conversion permettent aux composants qui n'ont pas été spécialement conçus pour fonctionner les uns avec les autres, d'établir et de mener des interactions.
- **Le service de facilitation :** négocie et améliore l'interaction entre composants.

Les propriétés fonctionnelles concernent les fonctions des connecteurs. Les propriétés non fonctionnelles d'un connecteur ne sont pas forcément obtenues à partir des spécifications de sa sémantique. Elles spécifient les besoins du connecteur pour une implémentation correcte. Par exemple, elles peuvent concerner la performance, la sécurité, la simulation de leurs comportements, leur analyse et la sélection de connecteurs appropriés et leurs correspondances.

Des contraintes de connecteurs doivent être spécifiées afin d'assurer les protocoles prévus, d'établir les dépendances intra-connecteurs et de fixer les conditions d'utilisation des connecteurs. Un exemple d'une contrainte simple, est la restriction du nombre de composants qui interagissent à travers un connecteur donné.

1.4 Configuration

Composants et connecteurs peuvent être assemblés a` partir de leurs interfaces pour former une configuration. Celle-ci décrit l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que leurs connexions. On parle aussi de topologie.

Une configuration représente en fait une instance possible d'un style architectural. Plus précisément, une configuration décrit les instances de composants intervenants et les relations qu'elles entretiennent entre elles [23].

La conception d'une architecture logicielle, a une importance capitale pour la réussite d'un projet informatique. Elle est souvent liée au savoir-faire de l'architecte. Une architecture logicielle doit tenir compte des contraintes suivantes :

– **La réutilisabilité** : est la capacité à rendre générique des composants et à concevoir et construire des boites noires susceptibles de fonctionner avec des langages et des environnements variés.

– **La maintenabilité** : est la capacité de modifier et d'adapter une application afin de la maintenir sur une période de vie assez longue. Une architecture bien spécifiée doit être maintenue tout au long de son cycle de vie. La prévision de l'intégration des extensions à l'architecture et la correction des erreurs sont nécessaires dès la phase de conception.

– **La performance** : c'est l'optimisation du temps mis par une application pour répondre à une requête donnée. La performance d'une application dépend de l'architecture Logicielle choisie, de son environnement d'exécution, de son implémentation et de la puissance des infrastructures utilisées.

2 La description de l'architecture logicielle

La description de l'architecture logicielle est fondée sur deux techniques de modélisation : la modélisation d'architecture logicielle à base de composants décrite par les ADLs (Architecture Description Languages) et la modélisation orientée objet utilisant le langage UML (Unified Modeling Language). Ces langages de modélisation d'architectures permettent la formalisation des architectures logicielles, la réduction du coût et l'augmentation de la performance du système logiciel ainsi que la compréhension des gros logiciels en les représentants à un niveau d'abstraction élevé.

2.1 ADL : Langage de description d'architecture :

Les langages (ADLs) sont des langages formels qui peuvent être utilisés pour représenter l'architecture d'un système logiciel. Comme l'architecture est devenue un thème important dans le développement de systèmes logiciels, les méthodes pour spécifier de façon non ambiguë une architecture, deviennent indispensables .

Les ADLs offrent une grande diversité de notations. Néanmoins, comme le montre la figure suivante , les concepts de composant, de connecteur et de configuration sont généralement considérés comme essentiels.

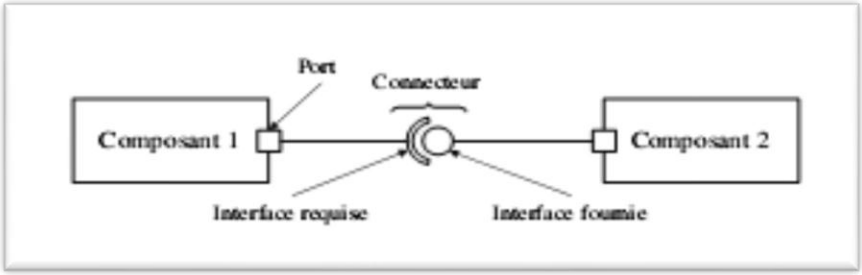


FIG 3.3 Les concepts des ADLs

2.2 Description des principaux ADLs

Les ADLs les plus connus et que nous étudions dans cette section sont Wright, Darwin, Archjava, Fractal. Différents langages de description d'architecture logicielle existent. Même s'ils possèdent parfois des caractéristiques communes, ils diffèrent sur d'autres relatives aux composants, à la composition de composants, à leurs cycles de vie, etc.

Les ADLs permettent la description et la visualisation des éléments architecturaux, l'analyse d'architectures, la documentation et la génération automatique de codes et offrent des outils spécifiques. Dans ce contexte les ADLs étudiés sont :

- ***Wright***: il fournit un modèle pour les architectures logicielles, il est largement utilisé pour l'analyse des protocoles d'interaction entre les composants architecturaux. [24]
- ***Darwin***: il supporte l'analyse des systèmes de transmission de message distribués. [25]
- ***ArchJava*** : pour les langages combinant la modélisation et la programmation. [26]
- ***Fractal*** : est un modèle de composants développé par France Télécom R&D et l'INRIA. Contrairement à d'autres modèles comme les EJB ou CCM dont les composants sont plutôt de grain moyen et destinés aux applications de gestion tournées vers l'Internet, la granularité des composants Fractal est quelconque. Leurs caractéristiques font qu'ils conviennent aussi bien à

des composants de bas niveau (par exemple un pool d'objets) que de haut niveau (par exemple une IHM complète).[27]

2.3 Outils de support des ADLs

L'utilité d'un ADL est directement liée aux outils et aux environnements permettant la description, l'analyse et l'exploitation d'architectures. Les outils de support sont principalement des :

- éditeurs graphiques et textuels des architectures,
- outils de vérification structurelle,
- outils d'analyse comportementale, utilisés pour l'exécution d'architectures (simulation et supervision),
- outils de production automatique des interfaces graphiques,
- outils de raffinement et de génération de code exécutable,
- outils permettant l'évolution d'architectures,
- outils de transformation de descriptions architecturales entre différents ADLs.

La plupart des ADLs décrits dans la section précédente possèdent des outils ou des environnements de développement permettant de définir et d'exploiter des descriptions architecturales [28] par exemple *Fractal ADL* fournit une palette d'outils :

- Fractal API : ensemble d'interfaces JAVA pour l'introspection et la reconfiguration dynamique de composants et d'assemblage de composants,
- Farclet : outil d'annotations des composants et des interfaces java et de génération de code associé aux annotations.
- FractalGUI: outil de conception d'architectures Fractal et de génération de code Fractal 2.0,
- Fractal pour Eclipse : environnement de développement d'applications Fractal.

3 La vérification et l'analyse des architectures logicielle :

les problèmes d'interactions entre composants simultanées causé logiciel réinitialise périodique réduction de la disponibilité pour exploration, aussi les interfaces partagé et leur service soit fourni ou requise causé des problèmes d'inter blocage , imbrication et conflit entre les composants , Alors pour la vérification des architectures logicielle, il faut défini toute les actions et les interactions entre les composante de notre système**,** et comme nous avons dites les interfaces dans un architecture logicielle c'est les interactions et la relation qui reliée les composants entre eux ,puis simulé le comportement de l'interaction entre les composant avec un

outil de vérification par exemple l'outil présenté à la sous-section suivante .

3.1 L'outil de vérification des architectures logicielles (LTSA)

LTSA est un outil de vérification de systèmes concurrents. Il vérifie mécaniquement que la spécification d'un système concurrent satisfait les propriétés requises de son comportement. En outre, LTSA soutient spécification animation pour faciliter l'exploration interactive du comportement du système.

Un système à LTSA est modélisé comme un ensemble d'interagir machines à états finis, Les propriétés requises du système sont également modélisés comme des machines d'état. LTSA effectue une analyse de la composition d'accessibilité exhaustive recherché les violations des propriétés souhaitées. Plus formellement, chaque composante d'un cahier des charges est décrit comme un système de transition Labellisée (LTS), qui contient tous les états d'un composant peut atteindre et toutes les transitions, il peut effectuer. Toutefois, la description explicite d'un LTS en termes de ses états, ensemble d'étiquettes d'action et relation de transition est lourd pour autre chose que les petits systèmes. Par conséquent, LTSA prend en charge une notation d'algèbre de processus (FSP) pour une description concise de comportement du composant. L'outil permet aux LTS correspondant à une spécification FSP pour être visualisés graphiquement.

LTSA a une architecture extensible qui permet à des fonctionnalités supplémentaires qui seront ajoutés au moyen de plugins.

4 Les modèles d'architectures en XMI

Le standard XMI (XML Metadata Interchange) offre une représentation concrète des modèles d'architectures sous forme de documents XML. L'OMG utilise les mécanismes de définition de structure de balises XML DTD (Document Type Definition) et XML Schema. XMI permet de définir des structurations de balises nécessaires à la représentation des modèles au format XML. XMI s'appuie sur les documents XML et leurs structurations (modèles des instances d'une architecture) et ses documents DTD et leurs structurations (métamodèle d'une architecture). XMI est un standard assurant l'interopérabilité pour l'échange des modèles entre outils [18].

Conclusion

Pour représenter, les systèmes logiciels complexes de manière architecturelle, des notations expressives s'imposent. Les notations orientées modèle et les notations spécifiques des langages de descriptions d'architectures (ADLs) sont les deux approches de description des architectures logicielles des systèmes complexes, qu'on retient. Ces deux approches de description des architectures logicielles :

- L'approche orientée objet basée sur de construction logicielle depuis des entités encapsulées définissant des interfaces fournit un ensemble des services.
- L'approche orientée composant, fournit les composants et les connecteurs qui comme concepts clés de description des systèmes logiciels.

On peut conclure que la cœxistence de qualité des deux approches est sans aucun doute profitable pour la description de l'architecture logicielle. Sa modélisation nécessite absolument, l'utilisation de langages spécifiques dédiés, qui font l'objet du chapitre suivant.

.

Chapitre 4

La génération d'un outil pour les architectures logicielles et la transformation des Architectures logicielle vers XML.

Nous avons basé sur le méta-modèle suivant qu'on a développé par l'outil EMF dont on a parlé dans le chapitre 02.

1 Un méta-modèle pour les architectures logicielles

Pour définir un méta-modèle (ou une ecore) EMF utilise le diagramme de classe. Donc, une architecture logicielle (comme le montre le méta-modèle de la figure 4.1) est composée de 5 classes: ***Architecture, Component, interface, Realize et Use.***

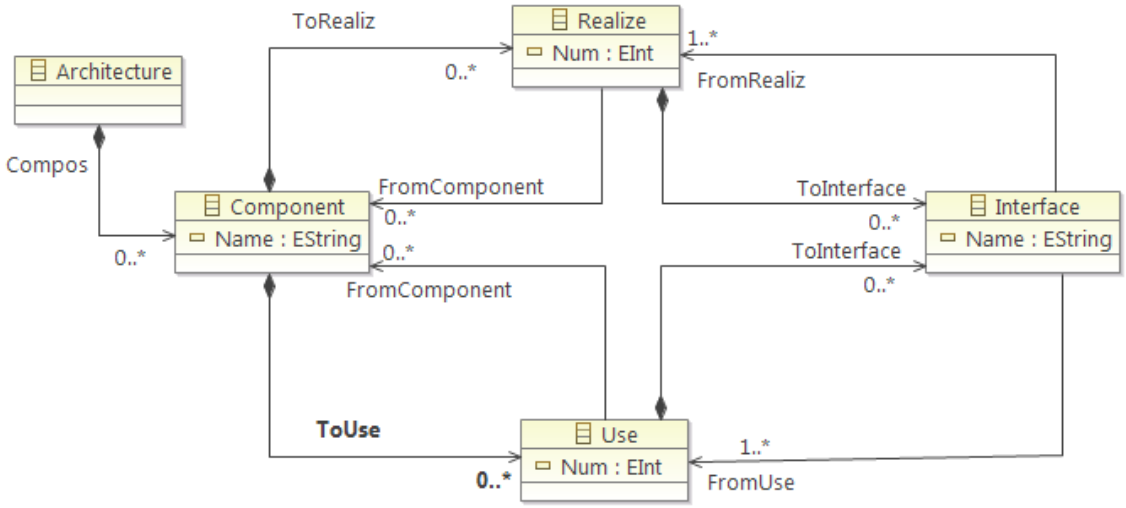


FIG 4.1 Le méta-model des Architecture logicielle

2 Un méta-modèle pour les documents XML :

Dans notre approche en concentré juste sur les éléments de base pour représenter le méta-modèle, alors les éléments de base pour les documents XML sont les 3 classes de méta-modèle suivant (FIG 4.2) : ***Document, Element, Atributte*** .

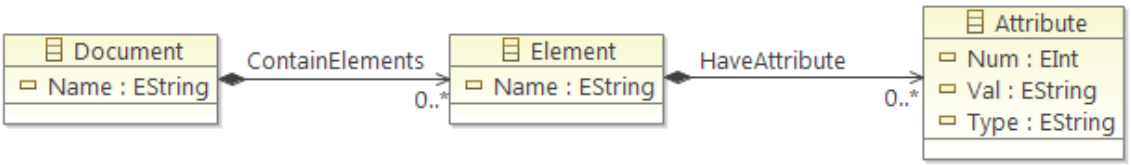


FIG 4.2 Le méta-model de document XML.

3 La génération d'un outil pour les architectures logicielles

En se basant sur le méta-modèle (voir la figure 4.1) on peut générer un outil qui nous permettra de créer des exemples d'architectures logicielles (des instances) avec les étapes suivantes :

- Création d'un projet EMF vide (File -> New -> Project…)

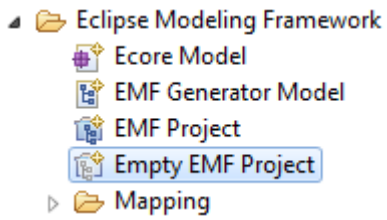


FIG 4.3 Crée projet EMF vide.

- Création d'un diagramme Ecore (New -> Other->)

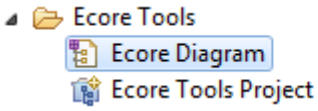


FIG 4.4 Crée diagramme Ecore.

- Défini les éléments de notre méta-modèle

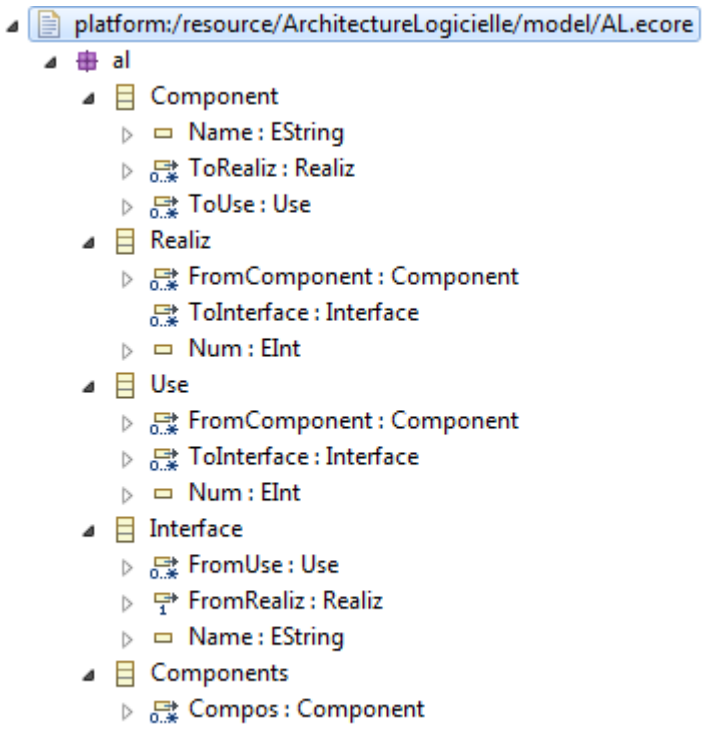


FIG 4.5 les éléments de méta-modèle d'architecture logicielle.

.

- Création de *genmodel*

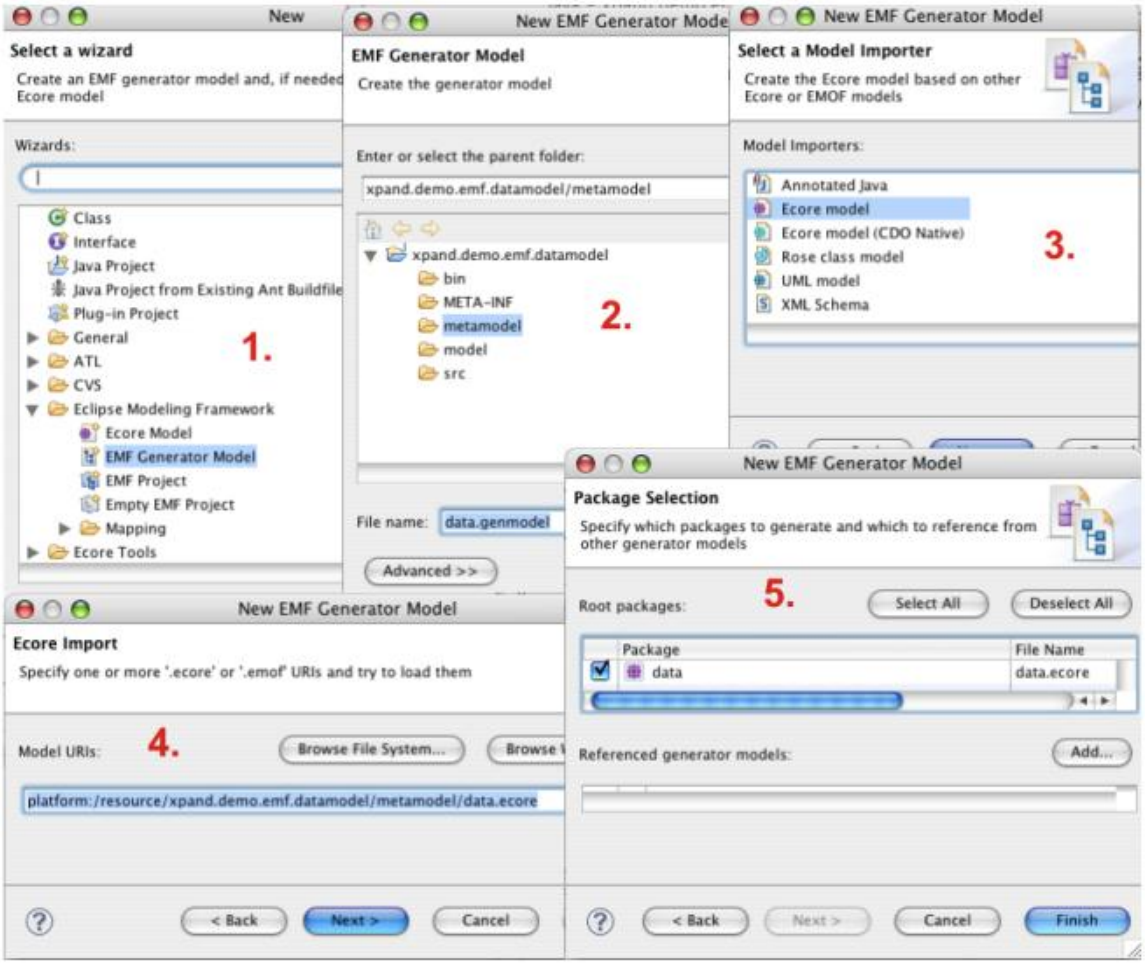


FIG 4.6 Création de genmodel.

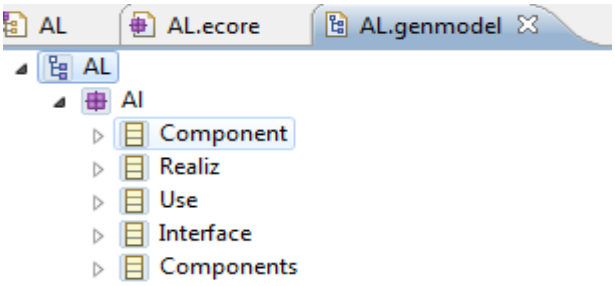


FIG 4.7 les éléments de genmodel de Architecture logicielle.

.

- Généré les projets édité

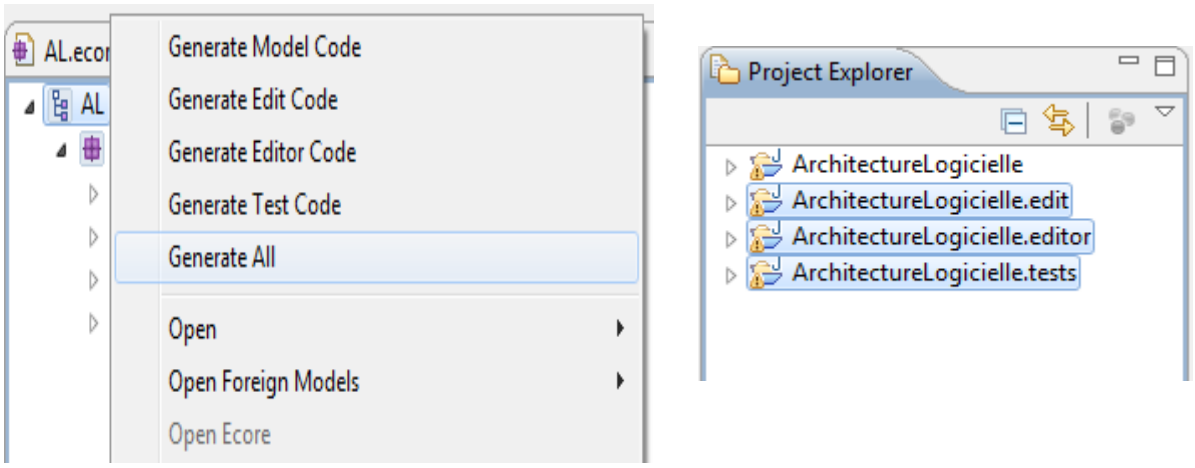


FIG 4.7 génération des projets Edité, Editor et Testes

- Lancer l'outil généré

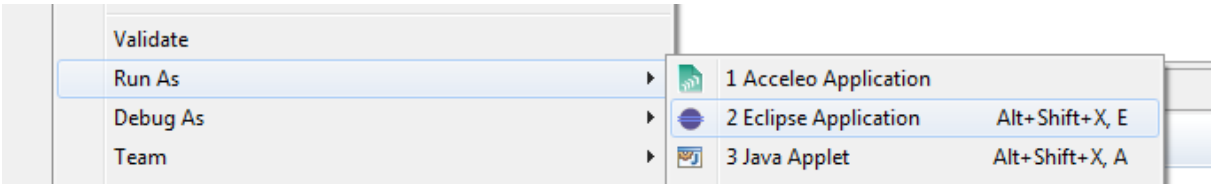


FIG 4.8 l'exécution de l'outil généré

3.1 Exemple : Méta-modèle d'une consultation de compte par l'outil généré

! L'exemple suivant illustre comment on peut utiliser notre outil généré précédemment pour la création d'une AL. Cet exemple décrit une structure globale d'une consultation d'un compte.

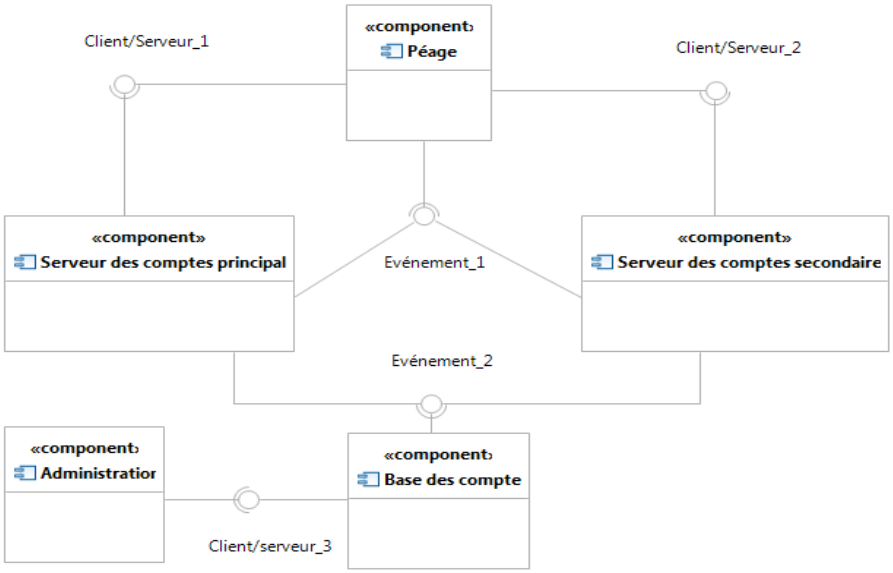


FIG 4.9 AL d'une consultation d'un compte

.

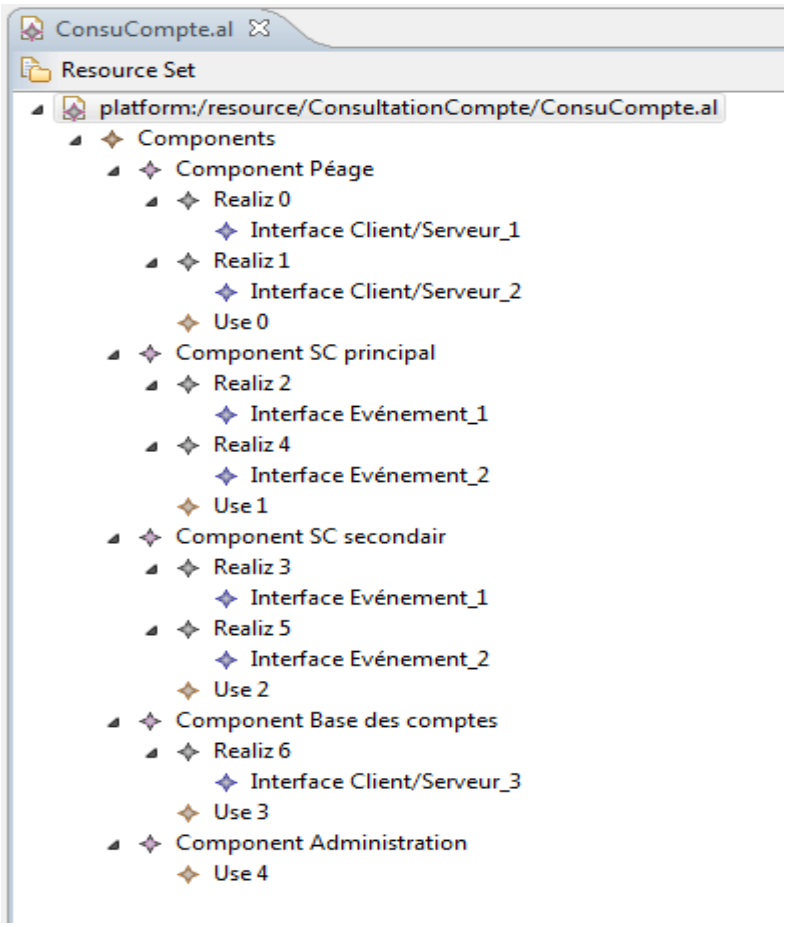


FIG 4.10 AL d'une consultation d'un compte par l'outil généré

4 Présentation des correspondances entre architecture logicielle et XML.

Les règles de transformation proposée consiste à présenter les correspondances entre Les deux modèles (Architecture logicielle et XML) suivante :

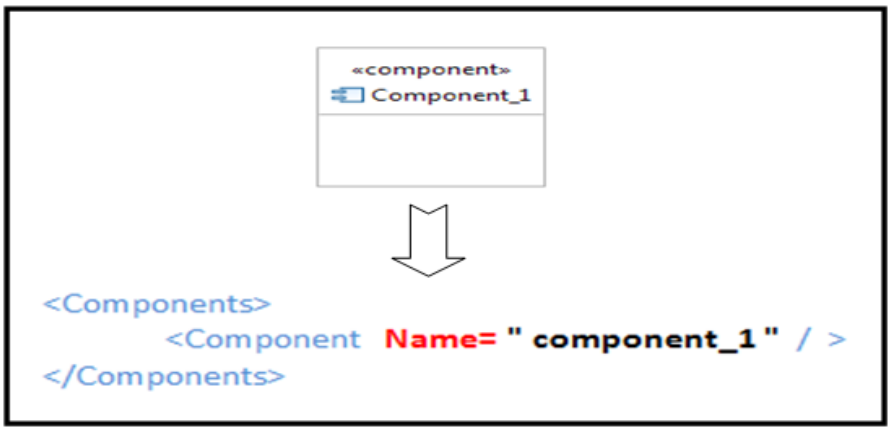


FIG 4.11 ***LHS*** & ***RHS*** de règle « ***CompoenentToXMLcomponent*** ».

.

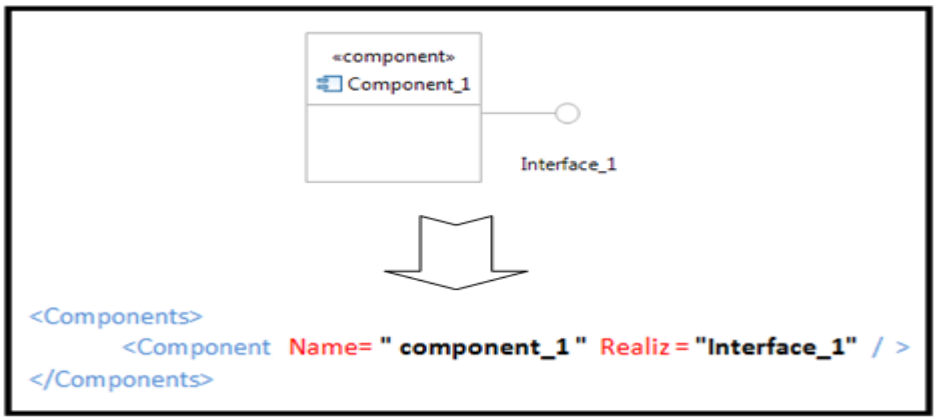


FIG 4.12 ***LHS*** & ***RHS*** de règle « ***RealizedInterfaceToXML*** ».

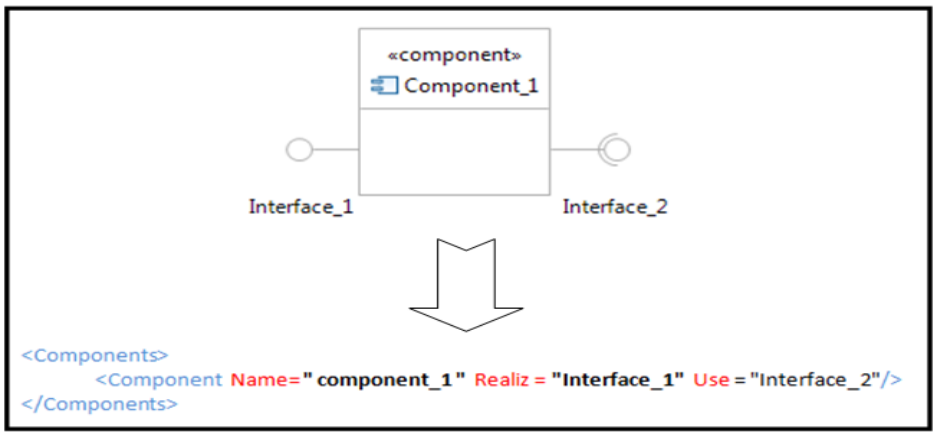


FIG 4.13 ***LHS*** & ***RHS*** de règle « ***UsedInterfaceToXML*** ».

5 Les règles de transformation ATL (AL2XML.ATL) :

Toute transformation est basée sur un ensemble de règles. Chacune a un ***input*** (ou ce qu'on appelle LHS : Left Hand Side) à son entrée et un ***output*** (ou RHS: Right Hand Side). Notre transformation est composée de **3** règles. Tous les inputs de nos règles représentent en vérité des éléments d'une AL. Ainsi, les outputs représentent des éléments XML qu'ils correspondent.

Règle 1: Architecture2Document

Cette règle ATL contienne la transformation de l'élément ***architecture*** de méta-modèle AL.ecore vers l'elément ***Document*** de méta-modèle XML.ecore.

Règle 2: Component2Element

Cette règle ATL contienne la transformation de l'élément ***Component*** de méta-modèle AL.ecore vers l'élément ***Element*** de méta-modèle XML.ecore.

Règle 3 :Realize2Atribute

Cette règle ATL contienne la transformation de l'élément ***Realize*** de méta-modèle AL.ecore vers l'élément ***Atributte*** de méta-modèle XML.ecore. Au l'attribut ***Val*** dans la classe Atributte c'est le nom de l'interface fourni par ***Component*** transformé.

Règle 4 : Use2Atributte

Cette règle ATL contienne la transformation de l'élément ***Use*** de méta-modèle AL.ecore vers l'élément ***Atributte*** de méta-modèle XML.ecore. Au l'attribut ***Val*** dans la classe ***Atributte*** c'est le nom de l'interface requise par ***Component*** transformé.

```
-- @path AL=/ARCH2XML/Al/AL.ecore
-- @path XML=/ARCH2XML/XML/XML.ecore
module AL2XML;
create OUT : XML from IN : AL;
rule Architecture2Document {
    from
        A : AL!Architecture
    to
        D: XML!Document ( Name <- A.Name,D.ContainElements <- a.ContainComponents )
    }
rule Component2Element {
    from
        C : AL!Component
    to
        E: XML!Element ( Name <- C.Name,
            if C.ToRealize then
            E.HaveAttribute <- C.ToRealize ,
            else
                E.HaveAttribute <- C.ToUse
            )
    }
rule Realize2Atributte {
    from
        R : AL!Realize
    to
        A: XML!Atributte(Num <- R.Num,type<-"Realize",Val <- R.getInterfaceName() )
    }
rule Use2Atributte {
    from
        U : AL!Use
    to
        A: XML!Atributte ( Num <- U.Num,type <- "Use" , Val <- U.getInterfaceName() )
    }
```

FIG 4.14 le fichier AL2XML.atl

En utilisant notre règles qu'on a défini (FIG 4.14) notre outil visite tous les composants et les interfaces de l'architecture logicielle qui se trouve sur son canevas afin d'exécuter le code nécessaire pour la génération du model XML correspondant. Le résultat c'est un fichier XMI contient les éléments XML correspondance de architecture logicielle.

5.1 Exemple : transformation de méta-modèle de consultation de compte vers XML

Le document XML suivant (Figure 4.13) est le résultat de la transformation de l'architecture Logicielle Consultation d'un compte (voir la figure 4.13) créée dans l'exemple précédent.

```
<Architecture>
    <Component Name=" Péage " Realize = "Client/serveur_1"  Realize =" Client/serveur_2"Use = "Evénement_1"/>
    <Component Name=" SC principal " Realize = "Evénement_1" Realize = "Evénement_2" Use = "Client/serveur_1"/>
    <Component Name=" SC secondaire" Realize = " Evénement_1" Realize = "Evénement_2" Use = "Client/serveur_2"/>
    <Component Name=" Base des comptes " Realize =" Client/serveur_3 "Use = "Evénement_2"/>
    <Component Name=" Administration" Use =" Client/serveur_3 "/>
</Architecture>
```

FIG 4.15 document XML généré pour la consultation d'un compte.

6 La vérification de notre architecture logicielle transformée :

A l'aide de l'outil de vérification LTSA que en présenté dans le chapitre 3 en peut faire une petite vérification sur une opération de notre exemple transformé (FIG 4.14), cette opération est 'LOGIN'.

- En commencé par la définition des étapes de login :

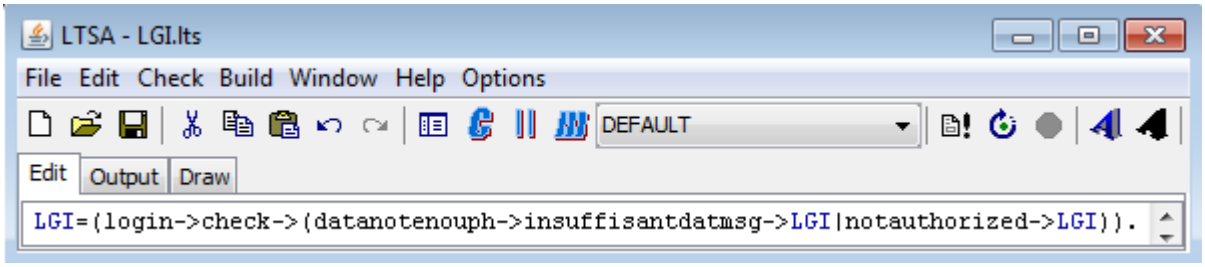


FIG 4.16 définition des étapes pour LOGIN

- Puis vérifié 'Check -> Saftly' :

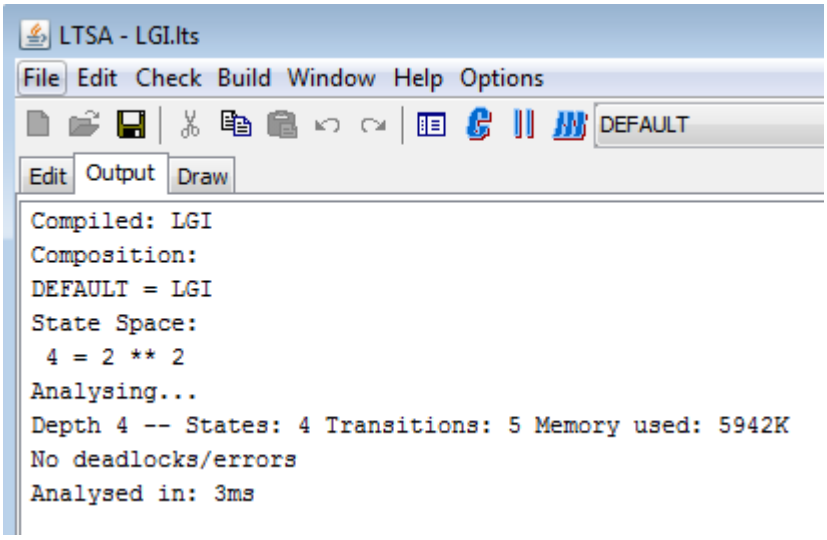


FIG 4.17 Résultat de vérification.

.

- Affiche résultat graphiquement :

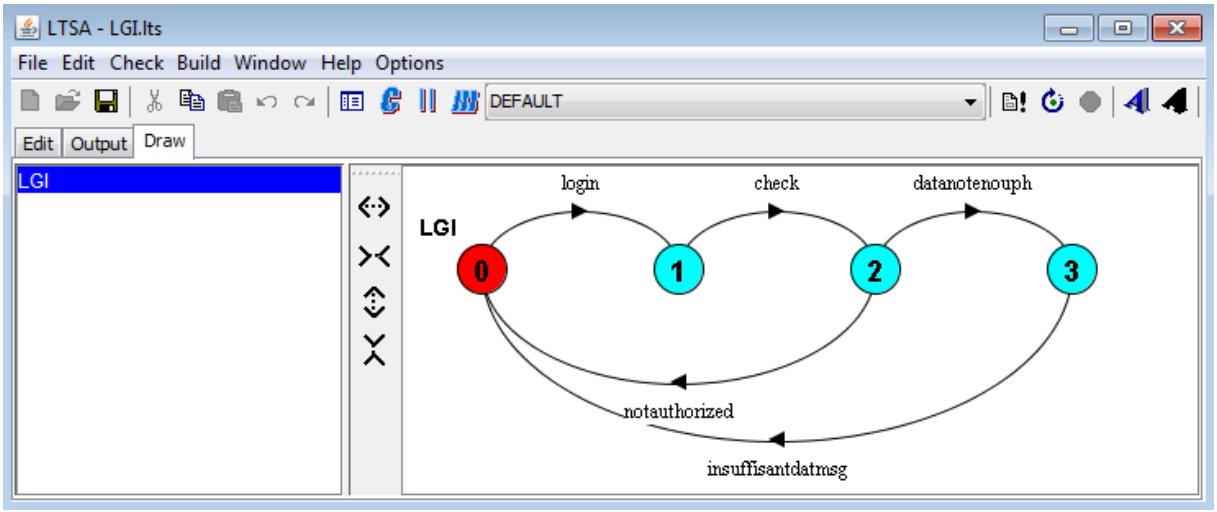


FIG 4.18 l'affichage graphique de vérification.

.

Conclusion :

Dans ce chapitre nous avons proposé une approche automatique pour transformer les AL vers XML. La méthode proposée se base sur langage de transformation ATL et utilise l'outil de modélisation et de méta-modélisation EMF.

Afin de réaliser cette méthode, nous avons proposé un méta-modèle (ecore EMF) qui nous a permis de générer un outil visuel pour la réalisation (écriture) des ALs.

Enfin, nous avons donné un exemple très riche pour illustrer notre démarche.

Conclusion générale

Le travail présenté dans ce mémoire s'inscrit dans le domaine de l'ingénierie dirigée par les modèles. Il se base essentiellement sur l'utilisation combinée de méta -modélisation et de transformation de modèle. Plus précisément, la méta-modélisation et transformation des architectures logicielles, à l'aide d'EMF et ATL.

Notre travail, repose sur deux axes : la définition des concepts et l'ensemble d'outils que de l'IDM supporte pour la manipulation des architectures logicielles et l'intégration des concepts d'architecture logicielle au sein de MDA.

Le premier axe concerne la définition formelle des concepts de l'IDM (***système modèle, méta-modèle*** et ***méta-méta-modèle***), le principe de l'MDA (***PIM***, ***PSM***) et la transformation des modèles puis la présentation des outils supportés pour la modélisation et la transformation des modèles, ainsi l'ensemble de concepts de l'architecture logicielle (***Composant***, ***Interface***, ***connecteur***, ***configuration*** …).

Le deuxième axe concerne l'intégration des concepts architecturaux au sein de l'approche MDA.

Nous avons donc adopté l'approche MDA en utilisant différents standards (EMF, ATL et XML) afin de présenter un outil extensible comportant l'ensemble d'éléments de base pour la manipulation des systèmes dans un niveau plus haut.

Le résultat de notre travail est une approche automatique pour transformer les Architectures logicielle vers XML.

Dans un travail futur nous cherchons de profiter des résultats obtenus (les architectures logicielles XML) pour faire certain évaluation, analyse et génération de codes dans différents plateformes spécifiques.

Bibliographie

1- Benoit combemale , ingénierie dirigée par les modelés (idm) état de l'art, université de toulouse article publie le 12 aout 2008.

2- Jean-marie favre, jacky estublier, Mireille blay-fornarino , l'ingénierie dirigée par les modeles au-dela du mda , hermes - lavoisier novembre 2007.

3- Jacques barzic, model driven architecture (mda), conservatoire national des arts et métiers, fevrier 2007.

4- Jamal abd-ali, méta modélisation et transformation automatique de psm dans une approche mda , mai 2006.

5- Jean Bezivin & Xavier blanc, mda : vers un important changement De paradigme en génie logiciel, Juillet 2002.

6- Xavier Blanc. MDA en action : Ingénierie logicielle guidée par les modéles. Ey- rolles, 2005.

7- Mohamed HADJ KACEM , Modélisation des applications distribuées à architecture dynamique : Conception et Validation , Novembre 2008.

8- Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. RSTI-L'Objet, 10(4) :145–157, 2004.

9- Wikipidia.com consulté le 16.04.2013

10- http://www.eclipsetotale.com. , 19.04.2013

11- Jacques Barzic , Eclipse et ses plugins de modélisation (emf – gef – gmf) , *janvier 2008.*

12- Vanwormhoudt G., Vérification de modèles avec EMF et OCL, Présentation PPT, 2004. disponible ici : http://www.enic.fr/people/Vanwormhoudt/siteEMFOCL/documents/ EMF OCLpresentation.pdf.

13- BARZIC J., Model Driven Architecture (MDA), 2006. Disponible à l'adresse : http://www.289eme.fr/pdf/mda.pdf.

14- http://www.eclipse.org/proposals/emf-tiger/ consulté le 04.06.2013

15- http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part.04.06.2013

16- http://fr.wikipedia.org/wiki/Topcased . 04.06.2013

17- http://atom3.cs.mcgill.ca/ .

18- Adel Alti , Coexistence de la modélisation à base d'objets et de la modélisation à

base de composants architecturaux pour la description de l'architecture logicielle, Juin 2011.

19- Garlan D., Monroe R., Wile D., (1997). Acme: An Architecture Description Interchange Language, Proceedings of CASCON'97, Toronto, Canada .

20- clemens szyperski. Component software - beyond object-oriented programming. Addison-wesley, novembre 2002.

21- Oussalah C., Khammaci T., Smeda A., (2005). Les composants : définitions et concepts de base, dans Ingénierie des composants, Editions Vuibert, ISBN 2-7117-4836-7.

22- Medvidovic N., Taylor R., (2000). A Classification and Comparison Framework for oftware Architecture Description Languages, IEEE Transactions on Software Engineering, Vol.26, No1.

23- jeff magee, naranker dulay, and jeff kramer. A constructive development envi- ronment for parallel and distributed programs. In *iwccs'94 : proceedings of the ieee workshop on configurable distributed systems*, north falmouth, massachu- setts, usa, mars 1994.

24- Allen R., Garlan G., (1994). Formalizing Architectural Connection, Proceedings of the Sixteenth International Conference on Software Engineering (ICSE'94), Sorrento (Italy).

25- Magee J., Karmer J., (1996). Dynamic Structure in Software Architecture, Proceedings of ACM SIGSOFT'96 : Fourth Symposium Foundation of Software Engineering, San Francisco, CA.

26- ArchJava., (2004). Disponible sur http://archjava.fluid.cs.cmu.edu.

27- Bruneton E., (2004). Developing with Fractal, the ObjectWeb Consortium, disponible sur http://fractal.objectweb.org/tutoriel/2004.