Ingénierie Dirigée par les Modèles (IDM)

3.1 Introduction

La complexité, la variété des technologies existantes et l'accroissement des besoins font aujourd'hui dans le domaine du développement logiciel un véritable défi. Pour affronter ce défi, il est nécessaire d'offrir une approche permettant d'abstraire toutes les complexités technologiques.

L'approche que nous allons étudiée dans ce chapitre est L'Ingénierie Dirigée par les Modèles (IDM). Nous le consacrerons aux concepts de base de l'IDM qui couvrent les disciplines dans lesquelles les modèles jouent un rôle principal. En suit nous allons discuter les différents outils que nous avons utilisés pour la réalisation de notre projet.

3.2 Ingénierie Dirigée par les Modèles (IDM)

3.2.1 Définition d'Ingénierie Dirigée par les Modèles (IDM)

Le principe de « tout est objet » de l'approche objet des années 80, est devenu très utile pour que l'ingénierie du logiciel s'oriente aujourd'hui vers l'IDM et le principe du tout est modèle [49].

IDM est une technique de développement mettant à disposition des outils, des concepts et des langages afin de simplifier et de mieux maîtriser le processus de développement de systèmes qui ne cessent de croitre en complexité [50]. Ces outils sont construits autour du concept de méta-modèle et le concept de transformation de modèles.

Cette technique est une forme d'ingénierie générative dans laquelle tout ou partie d'une

application est générée automatiquement ou semi-automatiquement à partir de modèles, en utilisant notamment des transformations successives de ces modèles.

3.2.2 Notion de base en Ingénierie Dirigée par les Modèles

Les éléments de base de l'IDM sont [51]:

3.2.2.1 Système

Un système est une entité complexe formée d'un ensemble ordonné d'éléments liés les uns aux autres et qui interagissent entre eux.

3.2.2.2 Modèle

Un modèle est une abstraction et une simplification du système qu'il représente. Pour mieux comprendre son processus de développement. Il offre donc une vision schématique d'un certains nombre d'éléments décrits sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions que l'on se pose sur lui.

3.2.2.3 Meta-modèle

Un méta-modèle est un modèle qui permet de définir le langage utilisé pour exprimer le modèle. Il représente donc les entités d'un langage, leurs relations ainsi que leurs contraintes, Autrement dit, le méta-modèle est une spécification de la syntaxe du langage.

3.2.2.4 Méta-méta-modèle

Le méta-modèle MOF (Meta-Object Facility) [52] est un langage standard de définition « de méta-modèles pour les méta-modèles » désigné par le terme de méta-méta-modèle. Ce dernier est un modèle qui décrit Les éléments de modélisation nécessaires à la définition des langages de méta modélisation pour pouvoir interpréter un méta-modèle. Les méta-méta-modèles sont unique et méta circulaire c'est-à-dire la capacité de se définir eux-mêmes, ils sont alors auto-descriptifs.

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale à quatre niveaux.

- Niveau M0 : Le monde réel ou se trouve le système à étudier.

- Niveau M1 : Les modèles représentant cette réalité défini dans un certain langage.
- Niveau M2 : Les méta-modèles permettant la définition de ces modèles (méta-modèle définissant ce langage).
- Niveau M3 : Le méta-méta-modèle. C'est le dernier niveau, il représente le MOF (langage de définition de méta-modèles).

La figure suivant représente la hiérarchie de déférents niveaux d'OMG :

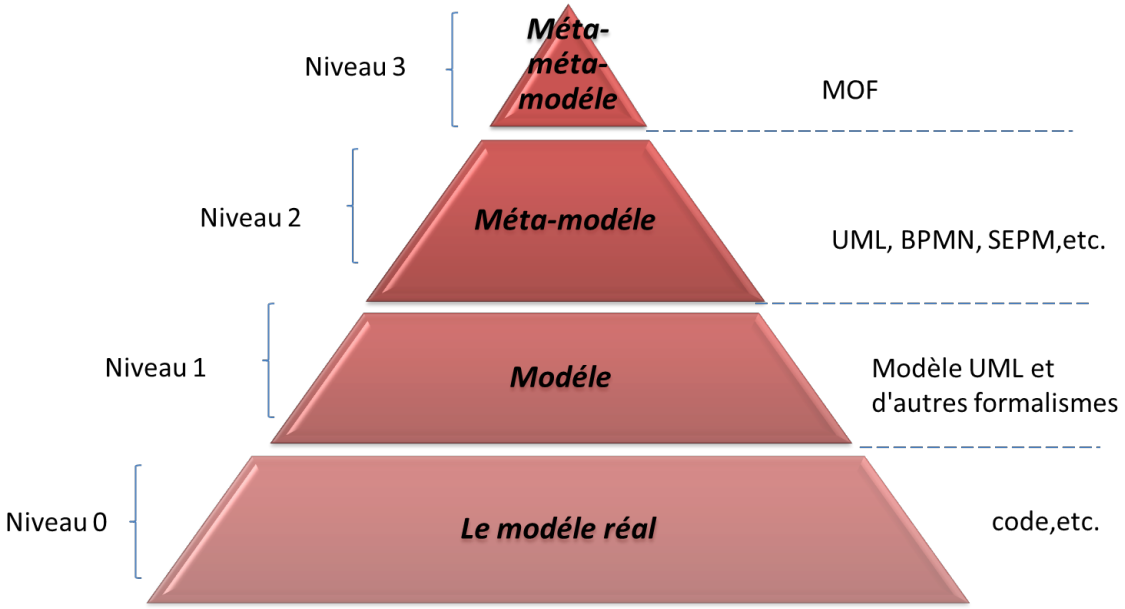

Figure 3.1: Pyramide de l'OMG.

3.2.3 Transformation de modèles

La notion de transformation de modèle est une opération et la génération d'un ou de plusieurs modèles cibles conformément à leur méta-modèle à partir d'un ou de plusieurs modèles sources conformément à leur méta-modèle.

Elle est nommé **d'endogène** si les modèles sources et cibles instancient le même méta modèle, sinon elle est dite **exogène** lorsque son modèle source et son modèle cible n'instancient pas le même méta modèle [51].

Une présentation générale des principaux concepts impliqués dans la transformation de modèles est illustrée dans la Figure 1.2.

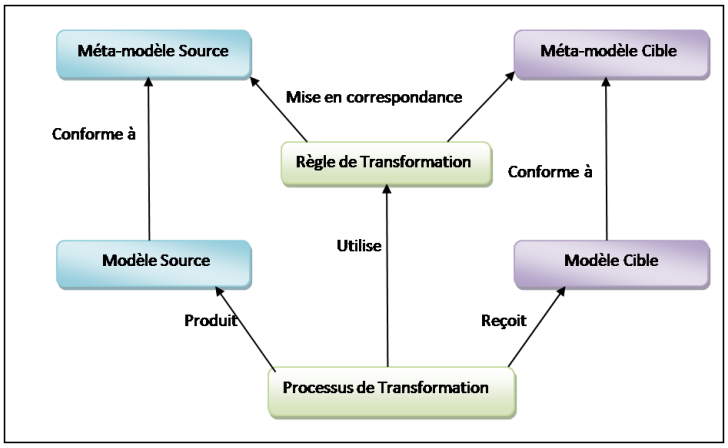

Figure 3.2: Schéma de base d'une transformation de modèles.

3.2.3.1 Types de transformation

Le niveau d'abstraction est un facteur important à prendre en considération. On basant sur ce dernier on peut distinguer les types de transformation suivants [53] :

3.2.3.2 Les transformations verticales

Une transformation est dite verticale lorsque la source et la cible sont définies à des niveaux d'abstraction différents. Une baisse du niveau d'abstraction d'une transformation est appelé raffinement. Par contre, une abstraction consiste en une élévation du niveau.

3.2.3.3 Les transformations horizontales

Une transformation est dite horizontale lorsque son modèle source et son modèle cible se trouvent au même niveau d'abstraction. On appliquant une modification à la représentation source. Cette modification peut être un ajout, une mise à jour, une suppression ou une restructuration d'informations.

3.2.3.4 Les transformations obliques

Une transformation oblique combine une transformation horizontale et une verticale. Les compilateurs peuvent utiliser ce type de transformation qui aide à effectuer des optimisations du code source avant de générer le code exécutable.

3.2.4 Les approches de transformation des Modèles

Plusieurs points de vue ont été la base de classification particulière de chacune des approches de transformation de modèle. Au plus haut niveau, La classification de **CZarnecki** [54] préconise deux types de transformation de modèles :

3.2.4.1 Les transformations Modèles vers Modèles

Ces transformations ont beaucoup évolué au cours de ces dernières années. Les transformations de modèle à modèle traduisent les modèles source en modèle cible, qui peuvent être des instances de méta-modèles identiques ou différents. Nous distinguons cinq techniques de transformation « Modèle vers Modèle » :

- **Approche par manipulation directe :**
  Cette approche de transformation se base sur une représentation interne des modèles source et cible manipuler par des APIs(Application Programming Interface),qui sont en général implémentées comme un framework orienté objet qui fournit une infrastructure pour organiser les transformations.

  L'utilisation de cette technique oblige le développeur de se préoccuper par l'implémentation ainsi que ordonnancement des règles. Il doit les faire à partir du zéro en utilisant un langage de programmation standard comme Java par exemple.

- **Approche relationnelle :**
  Dans cette approche les mathématiques jouent un rôle fondamental car cette manière de transformer est basée sur la logique déclarative qui exige l'utilisation des relations d'ordre mathématique pour permettre la spécification les relations entre les éléments du modèle source et ceux du modèle cible par l'adoption des contraints.

- **Approche basée sur les transformations de graphes :**
  Dans cette approche les modèles et les méta-modèles associés possèdent souvent une représentation graphique.

  Comme les approches relationnelles, les approches de transformation graphique sont capables d'exprimer la transformation du modèle en ce qui concerne la forme déclarative des règles de transformation. Les règles ne sont plus définies pour des éléments simples mais pour des fragments de modèles: on parle de filtrage de motif (pattern matching). Les motifs dans le modèle source, correspondant à certains critères, sont remplacés par d'autres motifs du modèle cible. Les motifs, ou fragments de modèles, sont exprimés soit dans les syntaxes concrètes respectives des modèles soit dans leur syntaxe abstraite.

- **Approche basée sur la structure :**
  Cette approche divise la transformation en deux étapes où la première est chargée de créer une structure hiérarchique du modèle cible, alors que la seconde est considérée

par l'ajustement de ses attributs et références dans la cible. Le cadre global détermine l'ordonnancement et la stratégie à appliquer, les utilisateurs ne se préoccupent que de fournir les règles de transformation.

- **Approche hybride :**
  Comme XDE et ATL, les approches hybrides sont la combinaison des différentes techniques ou alors des d'approches utilisant à la fois des règles à logique déclarative et impérative.

3.2.4.2 Les transformations modèles vers code

Nous pouvons considérer cette transformation comme un cas particulier de la transformation modèles vers modèles. En effet, il suffit de fournir un méta-modèle pour le langage de programmation cible.

En distingue deux approches liées à ce genre de transformation, les approches basées sur le principe du visiteur (Visitor-based approach). C'est une approche très basique de génération de code consiste à fournir un certain mécanisme de visiteur pour traverser la représentation interne d'un modèle et réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code obtenu est écrit dans un flux de texte. Ou celles basées sur le principe des patrons (Template-based approach) où Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source. FPL, XFramer et ANGIE sont des exemples de technologie liée à cette approche.

3.2.5 Avantage et inconvénient de l'IDM

L'approche IDM a des points forts et des points faibles selon [55] sont :

3.2.5.1 Avantages de l'IDM

- Elle met davantage l'accent sur l'automatisation du processus de développement des systèmes.
- Cette approche est basée principalement sur des représentations du système à un haut niveau d'abstraction qui sont les modèles.
- Le processus de développement dans cette approche revient à raffiner, maintenir, et éventuellement de transformer les modèles en d'autres modèles ou de générer le code exécutable. Il est important à noter que ces différentes activités se font d'une manière automatique.
- La méta-modélisation, qui est aussi un concept clé de l'approche IDM permet de donner à un langage de modélisation une notation abstraite ce qui permet de générer automa-

tiquement son éditeur. La méta modélisation des langages est de plus en plus adoptée pour des domaines spécifiques afin de réduire la complexité et d'exprimer efficacement les concepts du domaine.

3.2.5.2 Inconvénient de l'IDM :

- Malgré que la définition d'une syntaxe abstraite d'un langage par un méta modèle est bien maîtrisée et supportée par de nombreux environnements de méta modélisation, la définition de la sémantique de ces langages reste une question ouverte et cruciale. Actuellement, les environnements de méta modélisation sont en mesure de faire face à la plupart des problèmes de définition syntaxique, mais ils manquent de supports rigoureux permettant de fournir la sémantique des méta-modèles. La sémantique est généralement donnée en langage naturel, cela implique que les langages définis par méta modélisation ne sont pas encore aptes à l'analyse formelle de leurs modèles.
- L'absence de notations conviviale des différentes techniques formelles est un défi important pour les modèles formels. L'IDM permet, par le biais des notions de méta-modèles et de transformation de modèles, de concevoir des supports outillés pour les méthodes formelles. En ce sens, l'IDM n'apporte rien de nouveau sur le plan théorique aux concepts d'analyse formelle, mais elle peut permettre une meilleure utilisation afin de profiter pleinement de leurs avantages.

3.2.6 Les outils de l'IDM

Plusieurs outils d'IDM ont été développés, parmi celles-ci on peut citer :

- AToM3(A Tool for Multi-Formalism and Meta-Modelling).
- AGG (Attributed Graph Grammar citeagg)
- Eclipse Modeling Project
- Pogres PROgrammed Graph REwriting Systems
- VIATRA [VIATRA, 2010],VIsual Automated model TRAnsformations
- . . . etc

3.3 Outils de modélisation

3.3.1 Les origines de la fondation Eclipse

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 [56].

3.3.1.1 Les points forts d'Eclipse

Eclipse possède de nombreux points forts qui sont à l'origine de son énorme succès dont les principaux sont [57]:

- Une plate-forme ouverte pour le développement d'applications et extensible grâce à un mécanisme de plug-ins.
- Plusieurs versions d'un même plug-in peuvent cohabiter sur une même plateforme.
- Un support multi langage grâce à des plug-ins dédiés : Cobol, C, PHP, C#, ...
- Support de plusieurs plate-formes d'exécution : Windows, Linux, Mac OS X, ...
- Malgré son écriture en Java, Eclipse est très rapide à l'exécution grâce à l'utilisation de la bibliothèque SWT.
- Les nombreuses fonctionnalités de développement proposées par le JDT (refactoring très puissant, complétion de code, nombreux assistants, ...).
- Une ergonomie entièrement configurable qui propose selon les activités à réaliser différentes « perspectives »
- Un historique local des dernières modifications.
- La construction incrémentale des projets Java grâce à son propre compilateur qui permet en plus de compiler le code même avec des erreurs, de générer des messages d'erreurs personnalisés, de sélectionner la cible (java 1.3 ou 1.4) et de mettre en œuvre le scrapbook (permet des tests de code à la volée).
- Une exécution des applications dans une JVM dédiée sélectionnable avec possibilité d'utiliser un débogueur complet (points d'arrêts conditionnels, visualiser et modifier des variables, évaluation d'expression dans le contexte d'exécution, changement du code à chaud avec l'utilisation d'une JVM 1.4, ...).
- Propose le nécessaire pour développer de nouveaux plug-ins.

- Possibilité d'utiliser des outils open source : CVS, Ant, Junit.
- La plate-forme est entièrement internationalisée dans une dizaine de langue sous la forme d'un plug-in téléchargeable séparément.
- Le gestionnaire de mise à jour permet de télécharger de nouveaux plug-ins ou nouvelles versions d'un plug-in déjà installées à partir de sites web dédiés. (Eclipse 2.0).

3.3.1.2 Eclipse Modeling Tools

Le package modeling fournit des outils et des temps d'exécution pour construire des applications basées sur des modèles. On peut l'utiliser pour concevoir graphiquement des modèles de domaine, pour tirer parti de ces modèles au moment de la conception en créant et en éditant des instances dynamiques, pour collaborer via le support d'équipe d'Eclipse avec des facilités pour comparer et fusionner des modèles et des instances de modèle structurellement, et enfin générer du code Java à partir de ces modèles pour produire des applications complètes [4].

3.3.2 La plate-forme de modélisation sous Eclipse Modeling Tools

Dans cette section nous présentons brièvement les outils de modélisation de la plateforme Eclipse utilisés au cours de notre travail. Cette présentation concerne EMF.

3.3.2.1 Eclipse Modeling Framework (EMF) :

Eclipse Modeling Framework (EMF) [11] est une plate-forme de modélisation et de génération de code qui facilite la construction d'outils et d'autres applications basées sur des modèles de données structurées.

Il s'agit donc d'un ensemble d'outils de développement intégré à l'environnement Eclipse sous forme de plugins parmi lesquels on cite : le méta-modèle Ecore, l'éditeur EMF. Edit, le modèle de génération GenModel, etc. EMF a été conçu pour ouvrir Eclipse au développement dirigé par les modèles, c'est une approche basée sur une simplification du MOF. Il permet de définir des méta-modèles puis d'en dériver une implantation en Java pour construire des modèles instances [58].

EMF se compose de trois parties fondamentales :

- **Ecore :** Ecore est le modèle de base (méta-) au cœur du EMF. Il permet d'exprimer d'autres modèles en tirant parti de ses constructions. Ecore est aussi son propre méta-modèle (c.-à-d., Ecore est défini en termes de lui-même). Ecore est considéré comme l'implémentation de référence de facto de l'EMOF d'OMG (Meta-Object Essential Installation).

Le méta-modèle Ecore contient les constructions suivantes :

- EClass : représente une classe, avec zéro ou plus d'attributs et zéro ou plus de références. Nous appelons ses instances des classes.
- EAttribute : représente un attribut qui a un nom et un type. Nous appelons ses instances des attributs.
- EReference : représente une extrémité d'une association entre deux classes. Il y a des drapeaux pour indiquer s'il s'agit d'un confinement et d'une classe de référence qu'il désigne. Nous appelons ses cas des références.
- EDataType : représente le type d'un attribut, par exemple, int, float ou java.util.Date. Nous appelons ses instances des types [59].

- **EMF.Edit :** comprend des classes réutilisables génériques pour la construction d'éditeurs pour les modèles EMF. Il fournit :
  - Classes de fournisseurs de contenu et d'étiquettes : support des sources de propriétés et autres classes de commodité qui permettent aux modèles EMF d'être affichés à l'aide de visualisateurs de bureau standard (JFace) et de fiches de propriétés.
  - Un framework de commandes : comprenant un ensemble de classes d'implémentation de commandes génériques pour la construction d'éditeurs qui supportent l'annulation et le rétablissement entièrement automatiques.
- **EMF.Codegen :** L'installation de génération de code EMF est capable de générer tout le nécessaire pour construire un éditeur complet pour un modèle EMF. Il comprend une interface graphique à partir de laquelle les options de génération peuvent être spécifiées, et les générateurs peuvent être invoqués. L'installation de production tire parti du composant JDT (Java DevelopmentTooling) d'Eclipse [58].

3.3.2.2 objective d'Eclipse Modeling Framework :

L'objectif général d'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, présentés sous diverses formes, en Core Model.
2. La gestion de la persistance du Core Model.
3. La transformation du Core Model en code Java.

EMF peut gérer en entrée des modèles présentés sous trois formats :

- UML.

- XMI.
- Code Java Annoté.

Dans EMF, il est possible de définir un méta-modèle et de générer les interfaces afin de pouvoir manipuler les instances du méta-modèle dans Eclipse [60].

Les problèmes résolus par EMF

L'EMF peut être utilisé pour décrire et construire un modèle. Basé sur cette définition, le code Java peut être généré et amélioré par l'ajout de code Java de plus haut niveau. Ce modèle implémenté peut être utilisé comme base pour tout développement d'applications Java [61].

3.3.2.3 Les formats d'entrée standard :

- **UML :**
  Pour cette option, il existe trois possibilités.
  - L'édition directe conformément au méta-modèle Ecore.
  - L'importation de modèles UML.
  - L'exportation de modèles UML.
- **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.
  - Comme nous l'avons vu ci-dessus, il tend à devenir un standard, du moins pour le développement orienté objet.
  - 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.
- **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).

3.3.3 Graphical Editing Framework (GEF)

Le framework GEF a été créé pour l'environnement de développement Eclipse. L'interaction direct avec le méta-modèle se fait a travers la "couche" GEF, offrant ainsi a l'utilisateur tous les outils d'édition graphique du méta-modèle en question [62].

3.3.4 Graphical Modeling Framework GMF :

GMF pour « Graphical Modeling Framework » est un plugin qui permet de créer un éditeur graphique conforme à un modèle. Autrement dit, grâce à GMF, vous allez pouvoir vous fabriquer votre propre outil qui vous permettra d'instancier facilement votre propre modèle [63].

3.3.4.1 Composants de GMF :

GMF se compose principalement de deux parties [64] :

- **Runtime :** Attache EMF et GEF et représente le pont entre la notation et le domaine du modèle. Le « runtime » fournis aussi la persistance et la synchronisation entre les deux et permet de faciliter l'intégration entre EMF et GEF.
- **Génération (tooling) :** La génération de GMF permet aux utilisateurs de personnaliser leur structure de diagramme.

3.3.4.2 Editeur de diagramme avec GMF

La création d'un éditeur de diagramme à l'aide du GMF nécessite l'élaboration de six modèles qui sont : modèle de domaine (modèle EMF Ecore), modèle de générateur EMF, modèle de définition graphique, modèle de définition d'outillage, modèle de cartographie et mode générateur GMF.

- Le modèle de domaine (Domain Model) : portant l'extension. ecore, est développé en définissant un modèle Ecore avec EMF Editor ou en important le modèle d'autres fournisseurs Ecore. La définition du diagramme (.ecore_ diagram) est maintenue séparée du modèle de domaine dans GMF. De plus, le modèle de générateur EMF (.genmodel) doit être créé à partir du modèle Ecore à l'aide de l'assistant de modèle EMF [65].

- Le modèle de génération du code de l'éditeur, portant l'extension .gmfgen. Il est généré à partir du modèle de définition de mapping. Il permet de paramétrer les détails d'implémentation qui seront utilisés lors de la génération du code final de l'éditeur [66].
- Le modèle de définition graphique (Graphical Definition Model), portant l'extension .gmfgraph. Il est utilisé pour définir les différents éléments graphiques (figures, nœuds, compartiments, liens,etc.) dans l'éditeur [66].
- Le modèle de définition d'outillage (Tooling Definition Model) portant l'extension .gmftool. Il est utilisé pour la conception de la palette d'outils et d'autres outils périphériques comme les menus et les barres d'outils [66].
- Le modèle de définition du Mapping (Mapping Definition Model, portant l'extension .gmfmap. Il définit en utilisant l'assistant GMFMap les liaisons entre le modèle du domaine, le modèle de la définition graphique et le modèle de la définition d'outillage. Le GMF vise à permettre de lier un concept du domaine (défini dans le modèle de domaine) avec sa représentation graphique (définis dans le modèle de définition graphique) et les outils qui le gèrent (définis dans le modèle de définition de l'outillage) [66].
- Le modèle de générateur GMF permet de charger le modèle de cartographie et le modèle de générateur EMF afin de générer le code pour le plug-in d'éditeur de diagramme. Ainsi, il est analogue au modèle de générateur EMF en ce qu'il permet de définir et d'ajouter des paramètres de génération de code [64] [67].

3.3.5 Application Rich Client Plateform (RCP)

Pour générer l'application, deux options s'offraient à nous. Nous avions le choix de créer l'application sous forme d'une application RCP ou sous forme d'un greffon Eclipse. La seconde solution dépendant fortement d'Eclipse et donc de sa version, ce qui ne nous paraissait pas assez sûr c'est pourquoi nous avons fait le choix de l'application RCP, moins dépendante des versions et donc plus stable dans le temps [62].

3.4 Conclusion

l'IDM a atteint un bon niveau de maturité et est devenue une nouvelle démarche en génie logiciel qui conçoit l'intégralité du cycle de développement en se basant sur la méta-modélisation et transformation de modèles.

Les outils de modélisation offerts par Eclipse sont très puissants et facilitent énormément la tache de modélisation durant le développement orienté modèle des applications. L'utilisation d'EMF et GMF apporte beaucoup d'avantages car chacun joue le rôle d'un complément de l'autre : EMF est responsable de construire un modèle persistant et accessible à partir d'un

code Java. D'un autre coté GMF permet à l'utilisateur de définir comment l'éditeur doit apparaitre.

Chapitre 4

Développement d'un Environnement de Modélisation Graphique pour les Applications IoT

4.1 Introduction

Dans ce chapitre nous avons choisi la Norme BPMN 2.0, abordée dans le deuxième chapitre de ce mémoire, pour proposer un Environnements Graphique de modélisation du comportement des applications IoT en se concentrant sur les interactions entre le processus central et les dispositifs IoT. Notre démarche est basée sur la méta-modélisation et l'utilisation des technologies (EMF et GMF) de la plate-forme Eclipse Modeling Tools.

Notre projet se divise en trois étapes:

La méta-modélisation: cette étape consiste à proposer un méta modèle du langage BPMN adapté à la modélisation des applications IoT.

Génération d'un éditeur arborescent avec EMF: dans la deuxième étape, la technologie EMF est utilisée afin de pouvoir générer un éditeur arborescent du méta-modèle proposé.

Génération d'un éditeur graphique avec GMF: dans la dernière étape, la technologie GMF est utilisée pour développer l'éditeur graphique.

4.2 La Méta-Modélisation de la notation BPMN

Pour définir le méta-modèle, nous avons choisis l'outil Eclipse Modeling Tools comme un formalisme de méta-modélisation qui permet de créer des classes, des associations et des rela-

tions d'héritage sur le diagramme de classe disponible sur ce dernier.

4.2.1 Méta-modèle

Le méta-modèle que nous avons proposé (la figure 4.1) est une adaptation de méta-modèle BPMN 2.0 pour pouvoir modéliser les applications IoT. Il est composé de 35 classes reliées entre elles par 10 associations.

Nous avons choisi de représenter les dispositifs IoT comme des participants ou le comportement de chaque participant est modélisé dans des bassins (Pool) séparés. Pour cela nous avons crée deux classe (CentralProcess, IotDevice) héritées de la classe Pool (voir la figure 4.2):

- La classe **CentralProcess** représente le processus central de l'application IoT. Une seule instance de cette classe est possible dans les modèles.
- La classe **IotDevice** permet d'avoir un ou plusieurs dispositifs IoT.

Nous avons aussi respecté les conditions d'utilisation des objets connexion de BPMN 2.0 comme suit:

L'arc **séquenceFlow** Permet d'aller (voir la figure 4.3):

- D'une activité(ReceiveTask, SendTask, ManualTask, ScriptTask) vers une passerelle (OR, AND, XOR) et vise vers ça.
- D'une activité(ReceiveTask, SendTask, ManualTask, ScriptTask) vers un événement (StartEvent
  (Startnone, StartMsg, StartTimer)), IntermediateEvent(InSendMsg, InReceiveMsg, InTimer) et EndEvent(EndMsg, Terminate)et vise vers ça.
- D'une passerelle (OR, AND, XOR) vers un événement(StartEvnt(Startnone, StartMsg, StartTimer)), IntermediateEvent(InSendMsg, InReceiveMsg, InTimer) et EndEvent(EndMsg, Terminate) et vise vers ça.
- D'une activité vers une autre.
- D'un événement vers un autre.
- D'une passerelle vers une autre.

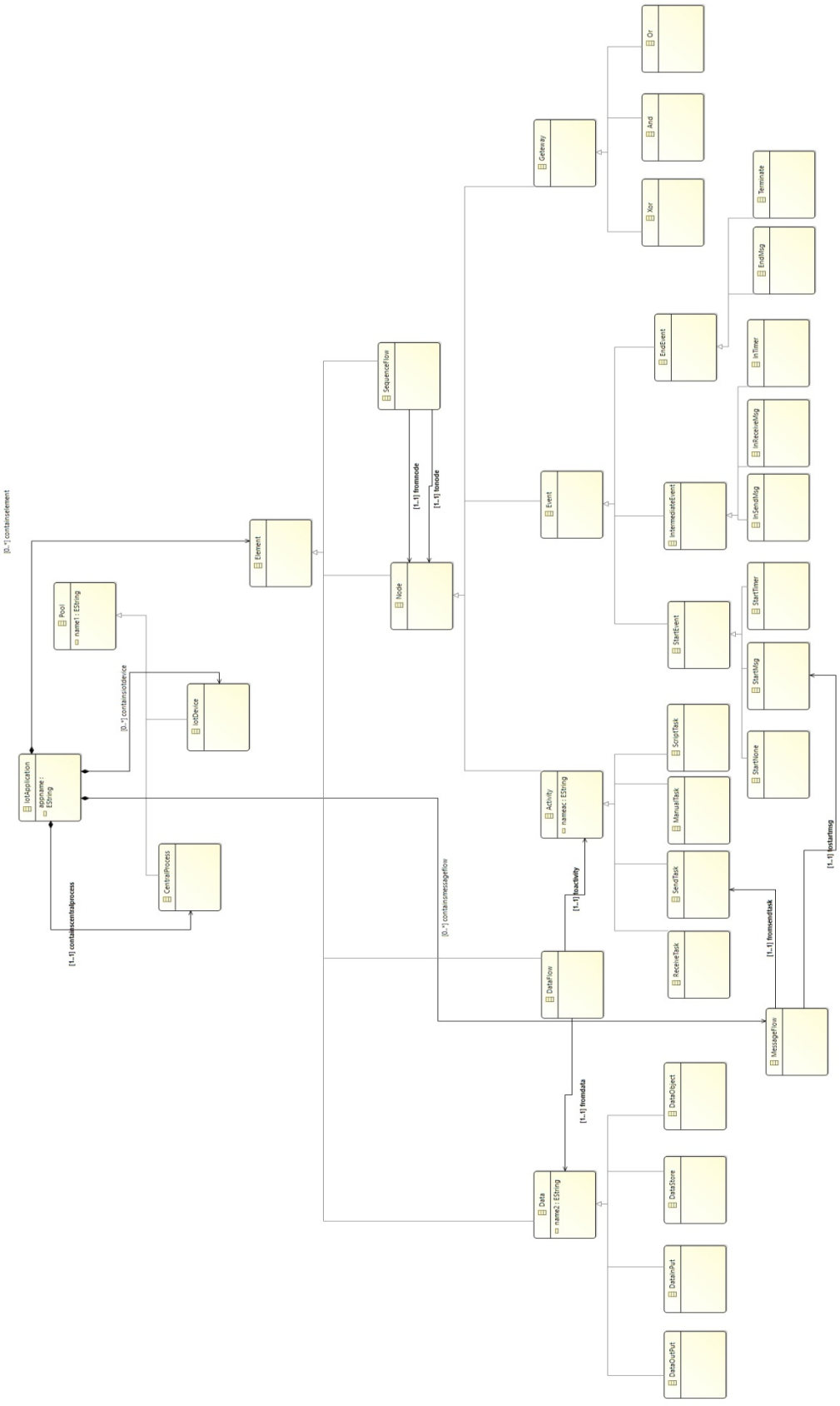

Figure 4.1: Extension du méta-modèle BPMN 2.0 sous Eclipse Modeling Tools

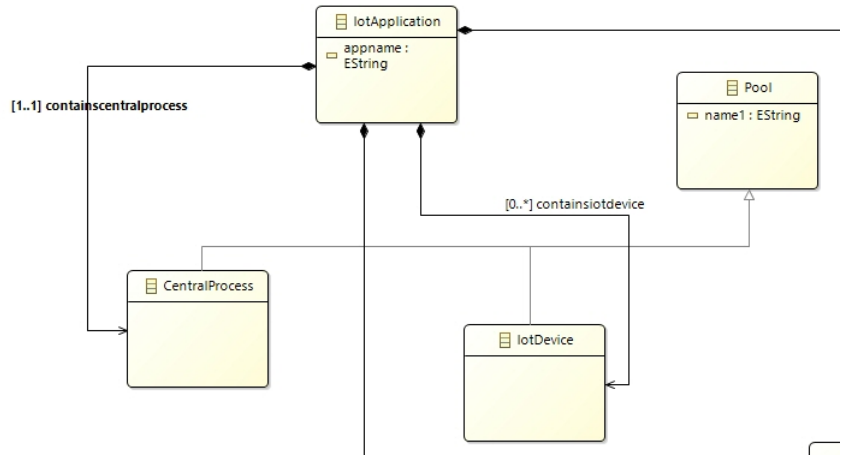

Figure 4.2: Représentation des dispositifs IoT dans le méta-modèle.

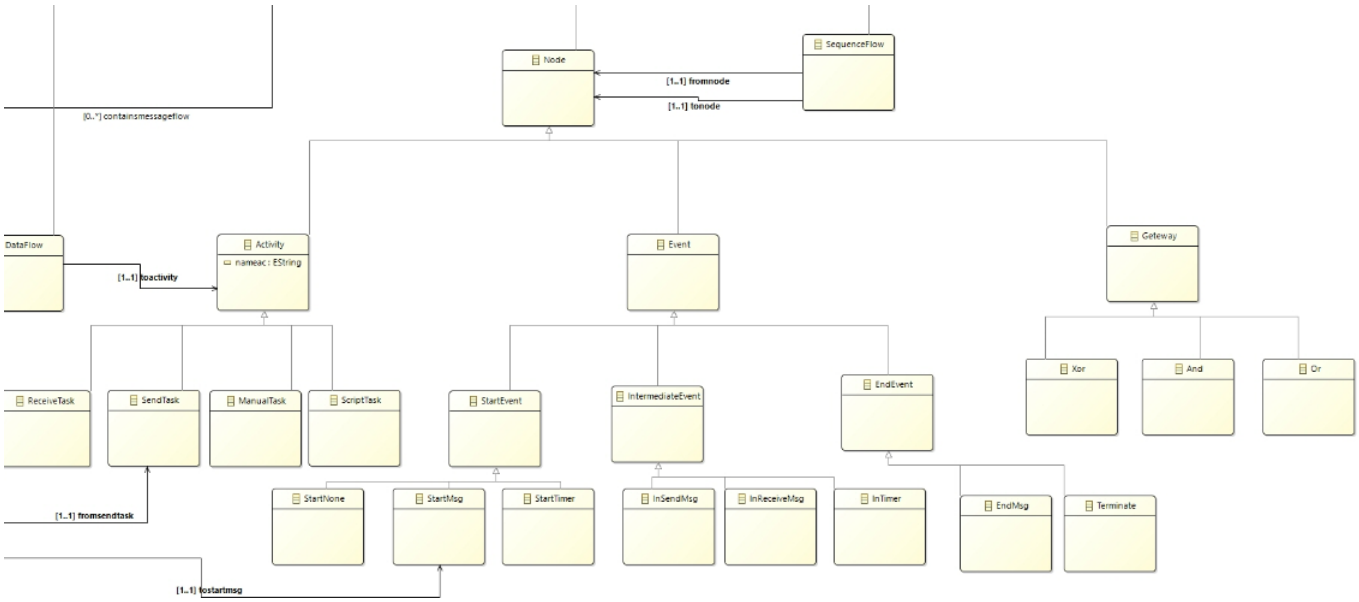

Figure 4.3: Représentation de flux de séquence dans le méta-modèle.

L'arc **MessageFlow** permet d'aller uniquement de l'activité SendTask vers l'événement start(StartMsg)(voir la figure 4.4).

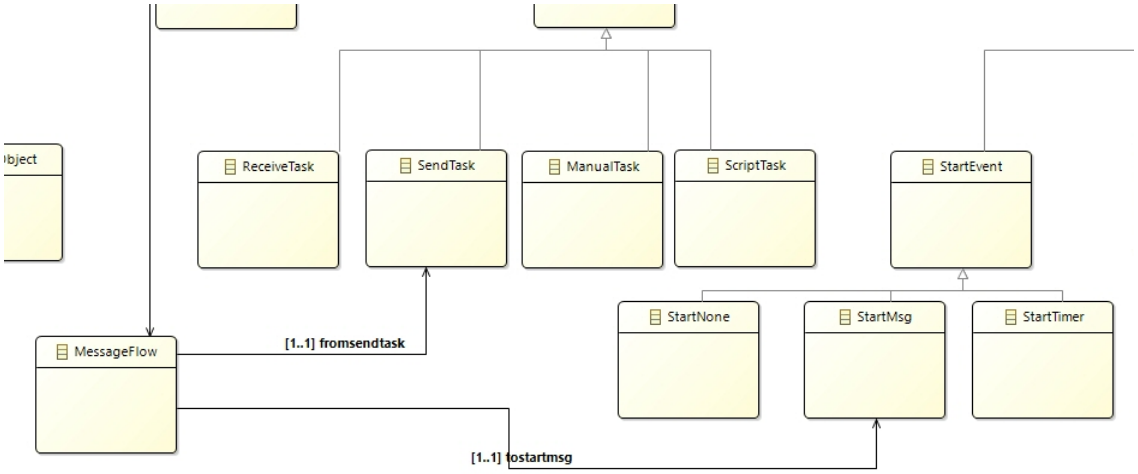

Figure 4.4: Représentation de flux de message dans le méta-modèle.

L'arc **DataFlow** permet d'aller uniquement de Data (DataOutput, DataInput, DataStore, DataObject)vers une activité(ReceiveTask, SendTask, ManualTask, ScriptTask)(voir la figure 4.5):

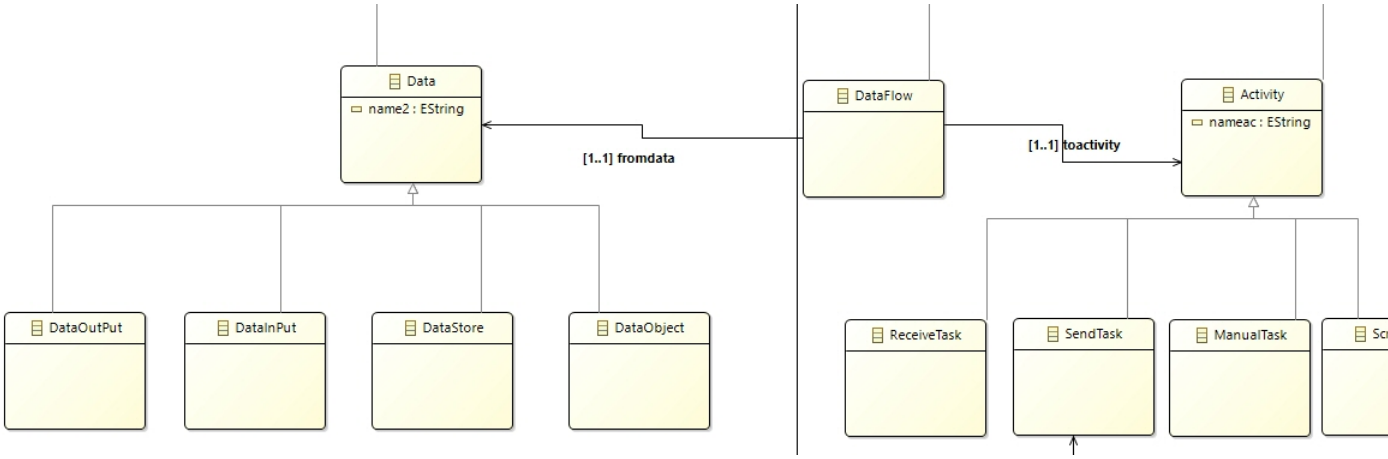

Figure 4.5: Représentation de flux Data dans le méta-modèle.

4.2.2 Les composants du Méta-modèle

4.2.2.1 Les classes

Le méta-modèle se compose de 35 classes. Le tableau ci-dessous représente une description de ces classes:

Class	Description