MA-004-451-1

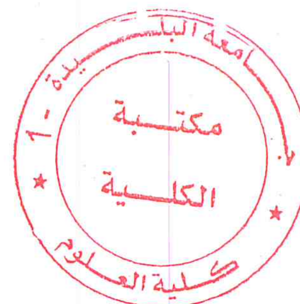
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saad DAHLEB de Blida

Faculté des sciences

Département d'informatique



Mémoire

MASTER ACADEMIQUE

Domaine : Mathématiques et d'Informatique

Filière : Informatique

Spécialité: Ingénierie Logiciel

THEME

Transformation de modèle : D'une architecture concrète en SysML vers un modèle de simulation en DEVS

Rapport présenter par : Abdesselam Ahmed

Zarouri Mahfoud

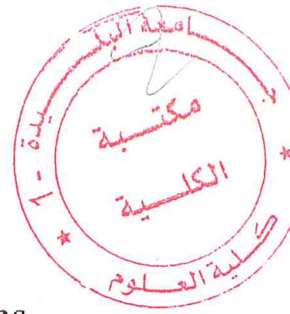
Directrice de mémoire : Mme Cherfa Imène

Devant le président de jury : Mme Boutoumi Bachira

Examinatrice : Mr Baouya abdelhakim

MA-004-451-1

Année Universitaire : 2017/ 2018



Remerciements

EN premier lieu nous souhaitons manifester nos sincères remerciements à :

- الله tout puissant، de nous avoir permis d'arriver à ce niveau d'étude، et aussi pour nous avoir donné beaucoup de patience et le chance et surtout le courage pour réaliser ce mémoire.
- Mme Imène Cherfa, notre promotrice, qui nous a suivis et prodigués de précieux conseils et ses dirigés du début jusqu'à la fin pour bien réaliser ce travail.
- Nos deux familles pour leur compréhension et leur présence tout au long de cette dure d'étude.
- Nos remerciements s'adressent aux membres du jury qui ont bien accepté de juger et d'´évaluer ce travail.
- Nos remerciements s'adressent à tous les enseignants qui ont contribué à notre formation durant les cinq ans d'études à cette université.
- A tous ceux qui de près ou de loin, ont contribué par leurs conseils, leurs encouragements et leurs amitiés, à l'édification de ce projet.

A toutes ces personnes, sincère remerciement du cœur.

Résumé

Le concept SoS présente un point de vue de haut niveau et explique les interactions entre chacun des systèmes constituants. Ce dernier possède ses propres caractéristiques pour atteindre ses propres objectifs. Le concept SoS est toujours à son stade de développement.

L'objectif de ce travail est de proposer une transformation de modèles de l'architecture concrète d'un SoS vers un langage de simulation. En effet, vue la dynamicité et l'interopérabilité des systèmes constituants d'un SoS, la simulation s'avère très importante.

Nous avons adapté des métamodèles existants, et réaliser la transformation vers le langage DEVS en utilisant le langage ATL.

Table des matières

Introduction Générale 01

Contexte 01
Problématique 01
Objective 01
Organisation du mémoire 01

PARTIE I: ETAT DE L'ART

CHAPITRE I: Ingénierie Systèmes De Systèmes

I.1. INTRODUCTION 02
I.2. L'ingénierie système 02
I.2.1 Définition du terme « système » 02
I.2.2 Qu'est-ce que l' « Ingénierie système » 02
I.3. Systèmes de systèmes (SoS) 03
I.4. Langages de simulation 04
I.4.1 Le langage de simulation Situation /réaction 05
I.4.1.1 Environnement du SoS 05
I.4.1.2 Définition de la mission 05
I.4.1.3 Définition du sous-système 06
I.4.1.4 Processus de simulation 07
I.4.2 DEVS 08
I. 4.2.1 Modélisation et simulation DEVS 08
I.4.2.2 XML et DEVS 09
I.5. Architecture des systèmes des systèmes 10
I.5.1 Principes et pratiques de l'architecture 11
I.5.2 Cadres d'architecture (DODAF & MODAF) 11
I.6. Méthodologies pour SOS 12
I.6.1 Cadre fondamental de M&S (Modeling and Simulation) 12
I.6.2 Ingénierie basée sur un modèle 13
I.7. Conclusion 14

CHAPITRE II: Transformation De Modèle

II.1. Introduction 16
II.2. Ingénierie dirigée par les modèles 16
II.3. Métamodélisation 17
II.3.1 Définition 17
II.3.2 Notion de modèles et méta-modèles........... 18
II.3.2.1 Modelés 19
II.3.2.2 Méta-modèles 20
II.3.2.3 Méta-méta-modèles 21
II.4. Transformation de modèles 23
II.4.1 Définition et principe de base 23
II.4.2 Taxonomie des transformations 24
II.4.2.1 Transformation horizontale de modèles 24
II.4.2.2 Transformation vertical de modèles 24
II.4.2.3 Transformation endogène de modèles 25
II.4.2.4 Transformation exogène de modèles 25
II.4.3 Les outils de transformation 26
II.4.3.1 Les langages et outil dédiés à la transformation de modèles.. 26
II.4.3.1.1 (Query ,Views ,transformation) 26
II.4.3.1.2 ATL.. 27
II.4.3.1.2.1 Présentation d'ATL 27
II.4.3.1.2.2 Règles de transformation 28
II.4.3.2 Outils de métamodélisation 33
II.5. Conclusion 34

CHAPITRE III : Les Langages de Modélisation

III.1. Introduction 36
III.2. SySML (System Modeling Language) 36
III.2.1 Diagramme de définition de blocs (BDD) 36
III.2.1.1Value Type 39
III.2.1.2 Partie 39
III.2.1.3 Composition 40
III.2.1.4 Agrégation 40
III.2.1.5 Association 40
III.2.1.6 généralisation 40
III.2.1.7 Opération 40
III.2.2 Diagramme de bloc interne (IBD) 42

III.2.2.1 Partie et connecteurs 43
III.2.2.2 Ports et interfaces 44
III.2.2.2.1 Type de ports 44
III.2.2.2.2 Interface 45
III.3. Le formalisme DEVS 46
III.3.2 Modèle atomique 46
III.3.3 Modèle couplé 47
III.4. Conclusion 49

PARTIE II : Contribution

CHAPITRE IV : Le Développement Des Méta modèles DEVS Et SYSML

IV.1. Introduction 52
IV.2. La Conception 52
IV.3. Le métamodèle SysML 52
IV.3.1 La structure Métamodèle de diagramme de BDD SysML 52
IV.3.2 Métamodèle de base DEVS 54
IV.3.2.1 Modèle couplé DEVS 56
IV.3.2.2 DEVS Atomique modèle 56
IV.4. La création de Métamodéle 58
IV.5. Les métamodeles proposés 58
IV.5.1 Métamodèle SysML des modèles (bloc et bloc interne) 58
IV.5.2 Métamodéle DEVS 60
IV.6. Les règles de Transformation de modèle SysML vers DEVS-XML.................................... 60
IV.7. Conclusion 64

CHAPITRE V : Implémentation

V.1. Introduction 66
V.2. Environnement de développement 66
V.2.1. Eclipse 66
V.2.2. Le plugin graphical modeling framework 67
V.2.3. GMF metamodele 67
V.2.4. Modèle de définition graphique 67

V.2.5. Editeur de diagramme de modèle Ecore........................ 67
V.3. La transformation avec Atlas Transformation Langage.................. 67
V.4. Echange de métadonnées XML 69
V.5. Présentation de l'application 69
V.5.1. Instance de création les modèles 70
V.5.2. Modèle de test et vérification de la transformation................ 71
V.5.3. Résultat de l'application 72
V.6. Evaluation de la complexité 74
V.7. Conclusion.. 75

Conclusion Générale 77

Liste des figures

CHAPITRE I: L'ingénierie Système de Système

Figure I-1 : Modèle conceptuel de SoS pour l'approche basée sur la situation / réaction... 07
Figure I-2 : Processus de simulation SoS. 08
Figure I-3 : Modèle DEVS représentant le système et les sous-systèmes.. 09
Figure I-4 : Exemple de simulation SoS avec trois systèmes et message de type XMIL 10
Figure I-5 : Processus graphique fondée sur un modèle 13

CHAPITRE II :

Figure II-1 : L'impact d'une approche IDM dans le processus de développement d'un système 17
Figure II-2 : Notions de base en technologie des objets 18
Figure II-3 : Notion de base en ingénierie des modèles 19
Figure II-4 : Relation entre système, modèle, méta-modèle et langage 21
Figure II-5 : la « pyramide » des « niveaux méta » de MDA.............. 22
Figure II-6 : Exemple d'organisation des modèles XML. 22
Figure II-7 : Schéma de base d'une transformation de modèles 24
Figure II-8 : Taxonomie des transformations de modèles................. 25
Figure II-9 : Architecture du standard QVT 27

Figure II-10 : Exemple de règle déclarative 30
Figure II-11 : matched standard rule 30
Figure II-12 : lazy rule 31
Figure II-13 : Unique lazy rules 32
Figure II-14 : Called rules ans action block 33
Figure II-15: Régle déclarative avec bloc d'action 33

CHAPITRE III :

Figure III-1 : Récapitulatif des diagrammes SysML [5] 36
Figure III-2 : Diagramme BDD simple 37
Figure III-3 : Syntaxe du diagramme BDD 37
Figure III-4 : Exemple de diagramme de définition de bloc............. 39
Figure III-5 : BDD Modèle RaddioReveil 41
Figure III-6 : Diagramme ibd simple [5] 42
Figure III-7 : Syntaxe du diagramme IBD 42
Figure III-8 : Les éléments d'un diagramme de bloc interne 43
Figure III-9 : Exemple de diagramme de bloc interne 45
Figure III-10 : Modèle atomique DEVS 47
Figure III-11 : Exemple de modèle couplé DEVS 48

CHAPITRE IV :

Figure IV-1 : Méta-modèle partiel montrant le bloc de BDD.............. 52
Figure IV-2 : Paradigme du modèle DEVS 54
Figure IV-3 : basic métamodéle DEVS.................................... 56
Figure IV-4 : Métamodéle diagramme de (bloc et bloc interne)........... 58
Figure IV-5 : Métamodéles DEVS 60

CHAPITRE V : Implémentation

Figure V-1 : Schéma de transformation dans ecore 68

Figure V-2 : Meta-formalisme ecore...................................... 68

Figure V-3 : Création d'une instance dynamique. 70

Figure V-4 : Modèle SysML a transformé en DEVS 71

Figure V-5 : Assessing Risks Concrete Architecture Modeling............ 71

Figure V-6 : représentation de port de communication entre les blocs.... 72

Figure V-7 : Choisir les règles de transformation 73

Figure V-8 : Les règles de transformation.............................. 73

Figure V-9 : Modèle de simulation en DEVS 74

Liste des tableaux

Tableau IV-1 : tableau représente la similarité entre l'entité SysML et les stéréotypes DEVS. 61

Tableau IV-2 : Le tableau de mapping entre SysML et modèle DEVS.. 62

Liste des Acronymes

IDM : Ingénierie Dirigée par les Modèles

OMG : Object Management Group

MDA : Model Driven Architecture

SoS : Systems of Systems

SysML : Systems Modeling Langage

DEVS : Discrete EVent system Specification

BDD : Block Definition Diagrams

IBD : Internal Block Diagrams

ATL : ATLas Transformation Language

C4I : Command, Control, Computers, Communications, and Information

ISR : Intelligence,Surveillance, and Reconnaissance

XMI : XML Metadata Interchange

XML : eXtensible Markup Language

DoDAF : Department of Defense Architecture Framework

MoDAF : Ministry of Defence Architecture Framework

FEA : Federal Enterprise Architecture

RUP : Rational Unified Process

TOGAF : The Open Group Architecture Framework

M&S : Modeling and Simulation (Modélisation et Simulation)

PIM : platform-independent model

PDM : Platform Definition Model

PSM : platform-specific model

UML : Unified Modeling Language

MDE : Model Driven Engineering (equivalent anglais de l'IDM)

MDD : Model Driven Development

CIM : Computation Independent Model

DTD : Document Type Definition

MM : MetaModel

QVT : Query, Views, Transformation

MOF, EMOF, CMOF : Essential/Complete Meta-Object Facility

OCL : Object Constraint Language

KM3 : Kernel MetaMetaModel

ADT : ATL Development Tool

RFP : Request For Proposal

XSLT : Extensible Stylesheet Language Transformation

AM : Atomic Modelé (Modèle atomique)

CM : Coupled Model (Modèle couplé)

GMF : eclipse Graphical Modeling Framework

EMF : Eclipse Modeling Framework

GEF : Graphical Editing Framework

M2M : Model To Model

Organisation de mémoire

Ce manuscrit s'organise en deux parties. Dont la première composée de trois chapitres, et porte sur l'ingénierie des systèmes et les différents outils de modélisation et de transformation de modelés. La seconde composée également de deux chapitres, et comporte les contributions, cette dernière comprend les métamodèles et l'essentiels des langages et outils que nous avons utilisé pour la conception et la réalisation de notre application.

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

- Dans le premier chapitre, nous dressons un état de l'art sur l'ingénierie des systèmes ainsi que les systèmes de systèmes (SoS) en présentant les concepts de base.
- Le deuxième chapitre présente les bases de l'IDM et la transformation de modèles avec un panorama d'outils et de langages dédiés ainsi que la métamodélisation.
- Le troisième chapitre détaille le contexte du langage source SysML et du langage cible qui est représenté par le formalisme DEVS.
- Le quatrième chapitre présente la phase conception de notre travail, cette dernière porté sur les métamodèles des diagrammes importants de SysML et le métamodèle de formalisme DEVS.
- Enfin le cinquième chapitre décrit en détail la phase implémentation, et porté sur la présentation de l'environnement de développement et les outils nécessaires afin de mettre en place un processus de vérification et la validation des transformations de modèles.

Chapitre I:
Ingénierie
Système De Système

Ici, nous énumérons seulement trois des nombreuses définitions potentielles [8] :

Définition 1: L'intégration du système de systèmes est une méthode pour poursuivre le développement, l'intégration, l'interopérabilité et l'optimisation des systèmes améliorer la performance dans les futurs scénarios de champs de bataille.

Définition 2: Les systèmes de systèmes existent lorsqu'il y a présence d'un la majorité des cinq caractéristiques suivantes: opérationnel et managérial indépendance, répartition géographique, comportement émergent, et le développement évolutif.

Définition 3: En ce qui concerne les combats en commun, le système de systèmes est concerné avec l'interopérabilité et la synergie de Command, Control, Ordinateurs, communications et information (C4I) et renseignement, Systèmes de surveillance et de reconnaissance (ISR).

I.4 Langages de simulation

Le principe général des systèmes de systèmes (SoS) est d'utiliser des systèmes existants pour construire un nouveau système. Dans l'ingénierie système, un SoS est un concept qui existe depuis un certain temps. Dans ce domaine, les systèmes existants sont principalement des systèmes physiques [7].

Par exemple, pour améliorer la qualité du traitement d'un patient, le suivi de sa vie quotidienne est le meilleur moyen de le maintenir à un niveau stable et d'éviter autant que possible des séquelles. Le suivi sera effectué grâce à la collaboration entre plusieurs systèmes d'information, composants de mesure, analyse de différents diagnostics. Chacun de ces éléments / personnes existe déjà et se comporte comme un système avec une mission bien définie. Mais pour l'accomplissement de la mission de garder un patient dans un état stable répond à un besoin au-delà de la capacité et le but de chacun d'eux. C'est rendue possible par les comportements émergents de la collaboration de ces différents systèmes / personnes [7].

Le principal intérêt des SoS réside dans les comportements émergents. En effet, c'est intéressant d'être en mesure d'exploiter un comportement qu'aucun système existant ne fournit seul, mais qui résulte de la collaboration de plusieurs d'entre eux. Cependant, la proximité des systèmes, qui n'étaient pas destinés à l'origine à travailler ensemble, peuvent produire des effets secondaires indésirables pour le SoS. Si pour le premier cas, nous peut parler de comportements émergents positifs, les indésirables les effets secondaires peuvent être considérés comme des

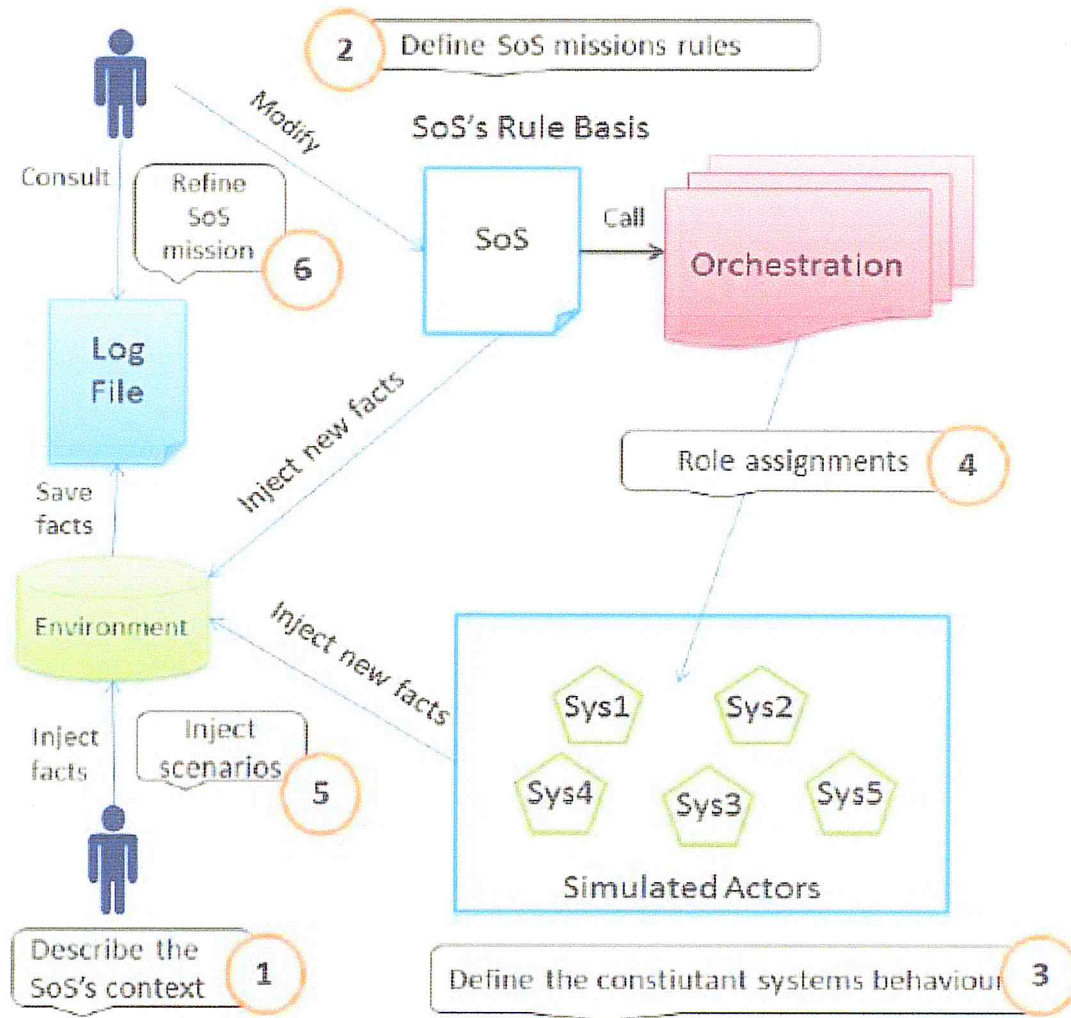


Figure I-2. Processus de simulation d'un SoS [7]

I.4.2 DEVS

I.4.2.1 Modélisation et simulation DEVS

Discrète Event System Spécification (DEVS) est un formalisme, qui fournit un moyen de spécifier les composants d'un système dans une simulation à événement discret. Dans le formalisme DEVS, il faut spécifier les modèles de base et comment ces modèles sont connectés ensemble. Ces modèles de base sont appelés modèles atomiques, et des modèles plus grands qui sont obtenus en connectant ces blocs atomiques sont appelés modèles couplés (voir la figure4) [8].

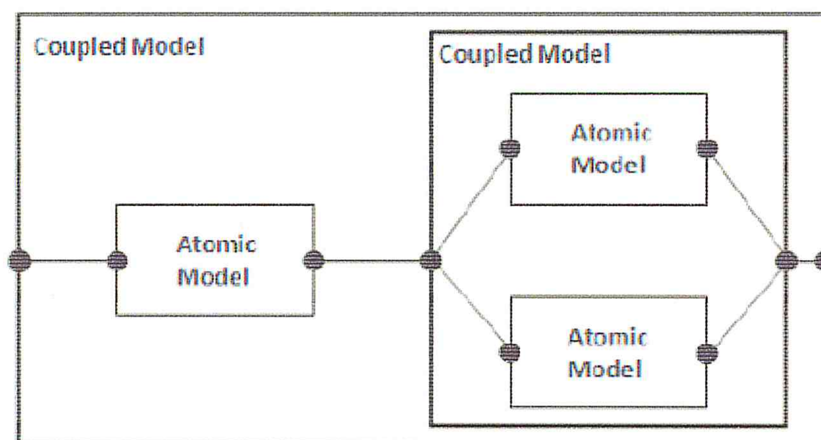


Figure I-3. Modèle DEVS représentant le système et les sous-systèmes [8]

L'environnement DEVS peut être utilisé avec succès pour simuler des problèmes plus complexes tels que systèmes autonomes dans un cadre de systèmes de systèmes.

Afin de tester l'architecture SoS dans l'environnement DEVS, le langage de base XML doit être intégré dans l'environnement de simulation. La phase suivante explore comment l'environnement XML et DEVS peuvent être combiné dans l'environnement de simulation.

I.4.2.2 XML et DEVS

Dans DEVS, les messages peuvent être transmis à partir d'un système (modèle couplé ou atomique) à l'autre en utilisant des formats de message prédéfinis ou définis par l'utilisateur. Depuis les systèmes au sein de SoS peuvent être différents dans le matériel et / ou le logiciel. Chaque système n'a pas nécessairement les connaissances (fonctionnement, mise en œuvre, timing, données problèmes, et ainsi de suite) d'un autre système dans un SoS. Par conséquent, il faut travailler à un niveau élevé (information ou niveau de données) afin de comprendre les conditions de travail du système. Un tel bon ajustement pour représenter les données d'une manière universelle est XML. La figure 5 décrit conceptuellement un exemple de simulation SoS pour démontrer l'utilisation de XML pour le passage de message en utilisant le formalisme DEVS [8].

Dans la figure 4, il existe trois systèmes dans une hiérarchie où les systèmes A et B peuvent envoyer et recevoir des données du système C. Le système C envoie et reçoit des données à partir d'un niveau supérieur comme décrit dans le message du système C.

Avec l'architecture d'intégration XML mentionnée ci-dessus et l'environnement DEVS, nous pouvons alors offrir des solutions au système de systèmes qui sont des systèmes complexes hétérogènes tels que des plates-formes de capteurs mobiles ou microdispositifs.

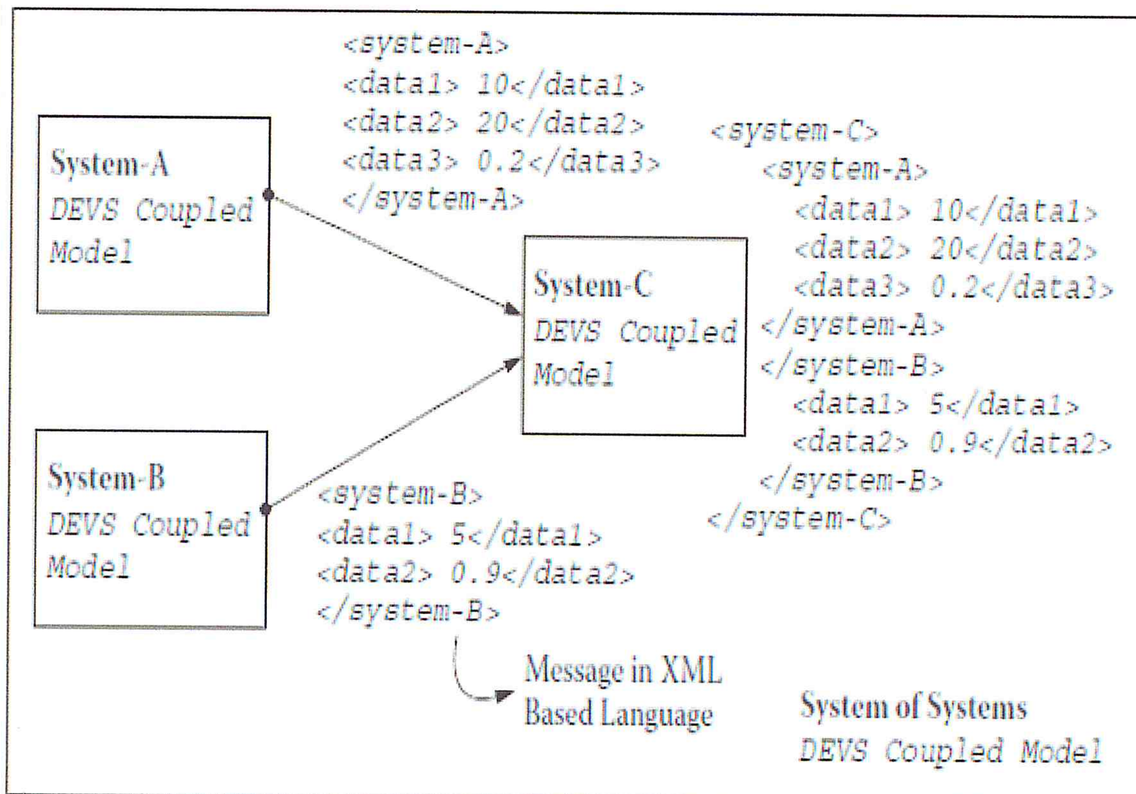


Figure I-4. Exemple de simulation SoS avec trois systèmes et message de type XML [8]

I.5 Architecture des systèmes de systèmes

La définition de l'architecture d'un système constitue l'étape la plus difficile à faire dans le développement d'un SoS. L'organisation fondamentale d'un système en termes de composants, leurs relations les uns aux autres, et à l'environnement, et les principes guidant sa conception et son évolution. Tandis qu'il est impossible de comprendre toutes les caractéristiques et les conséquences de l'architecture au moment où le système est conçu, il est possible de produire une architecture système qui maximise la capacité du système à répondre aux besoins des utilisateurs tout en minimisant les conséquences involontaire [8].

Partie I: Etat de l'art

L'architecture du système de systèmes concerne principalement l'architecture des systèmes créés à partir d'autres systèmes autonomes. Il y a deux disciplines d'architecture importantes qui sont étroitement liées à l'architecture des SoS.

Tout d'abord, il y a l'architecture du système, qui est principalement concernée avec les humains, les activités et les technologies, y compris la structure et le comportement qui constituent un système autonome au sein d'une entreprise. Alors qu'il va normalement interagir avec d'autres systèmes autonomes pour maximiser les capacités de l'entreprise, ses fonctions de base ne devrait pas dépendre de ces systèmes.

Ensuite, il y a l'architecture de l'entreprise, qui concerne principalement les ressources organisationnelles (personnes, information, capital et infrastructure physique) et activités [8].

I.5.1 Principes et pratiques de l'architecture

La conception de l'architecture commence par la reconnaissance d'un besoin, la déclaration de problème, et l'articulation d'une stratégie de solution. Elle continue avec la solution synthèse et analyse d'alternatives. Elle se termine par un modèle d'architecture du système à construire. Alors que le processus est assez simple, il est compliqué par le fait que la conception ne coule pas logiquement aux besoins. Il existe un processus bien défini pour l'architecture des systèmes et un ensemble de principes consacrés à la navigation dans l'espace de solution [8].

I.5.2 Cadres d'architecture (DODAF & MODAF)

Nous citons certains cadres d'architecture spécifiques. Chacun de ces cadres a été conçu pour répondre à un besoin spécifique. Tandis que certains d'entre eux sont plus adaptés à l'architecture SoS que d'autres, il peuvent tous être utilisés. Les cadres sont énumérés ci-dessous:

- Le cadre de Zachman
- Le Cadre d'architecture du Département de la défense (DoDAF)
- Le cadre d'architecture du ministère de la Défense (MoDAF)
- Le cadre de l'architecture d'entreprise fédérale (FEA)
- Le processus rationnel unifié (RUP)
- Le cadre d'architecture de groupe ouvert (TOGAF)

Les structures des trois cadres les plus significatifs-Zachman, DoDAF et MoDAF-sont discutés plus en détail ci-dessous [8] :

- Le cadre de Zachman, présenté en 1987 était le premier cadre d'architecture applicable aux SoS. Le cadre de Zachman a servi de fondation pour tous les autres cadres d'architecture abordés dans cette phase.

- Le Département de la Défense (DoD) maintient DoDAF, qui est utilisé par le Département de la Défense des États-Unis pour décrire les architectures. Sa structure est très bonne pour décrire l'architecture des SoS. Il a trois vues principales: vue opérationnelle, vue système et vue des normes techniques.
- Le ministère de la Défense du Royaume-Uni (U.K.) a modélisé le MoDAF après le modèle DoDAF. Notez que sa structure (montrée dans le tableau 2.4) [18] est très similaire à DoDAF. En plus des vues de base contenues dans DoDAF, MoDAF ajoute deux vues de base supplémentaires: la vue stratégique et la vue d'acquisition.

MoDAF est un autre excellent cadre pour représenter l'architecture des SoS. Outre que DoDAF et MoDAF ont tous été créés pour un usage gouvernemental les États-Unis et le Royaume-Uni, ils sont tous disponibles gratuitement.

I.6 Méthodologies pour SOS

I.6.1 Cadre fondamental de M&S (Modeling and Simulation)

La théorie de la modélisation et de la simulation fournit un cadre conceptuel et une approche computationnelle associée aux problèmes méthodologiques en M&S. Le cadre fournit un ensemble d'entités (système réel, modèle, simulateur, cadre expérimental) et les relations entre les entités (validité du modèle, exactitude du simulateur, entre autres) qui, en effet, présentent une ontologie du domaine M&S. L'approche computationnelle est basée sur la théorie mathématique des systèmes et travaille avec l'orientation de l'objet et d'autres paradigmes de calcul. Il est destiné à fournir un moyen sonore de manipuler les éléments du cadre et de dériver des relations logiques parmi ceux qui sont utilement appliqués aux problèmes du monde réel dans la modélisation et la simulation [9].

Dans sa réalisation computationnelle, le cadre est basé sur le système d'événements discrets.Le formalisme des spécifications (DEVS) est implémenté dans divers environnements orientés objet.

I.6.2 Ingénierie basée sur un modèle

Le processus d'ingénierie logicielle basé sur un modèle est communément appelé Model Driven Architecture (MDA) ou ingénierie dirigée par les modèles. L'idée de base derrière cette approche est de développer un modèle avant que l'artefact ou le produit soit conçu, puis transformer le modèle lui-même au produit réel. Le MDA est avancé par Object Management Group (OMG).

L'approche MDA définit la fonctionnalité du système à l'aide d'un modèle indépendant de la plate-forme (PIM) en utilisant un langage spécifique au domaine approprié. Puis donné un modèle de définition de plate-forme (PDM), le PIM est traduit en un ou plusieurs modèles spécifiques à la plate-forme (PSM). L'OMG documente le processus global dans un document appelé guide MDA.

L'outil MDA est utilisé pour développer, interpréter, comparer, aligner, etc. des modèles ou des méta-modèles [9].

Voici le processus de la technique d'essai fondée sur un modèle, comme le montre la figure 5:

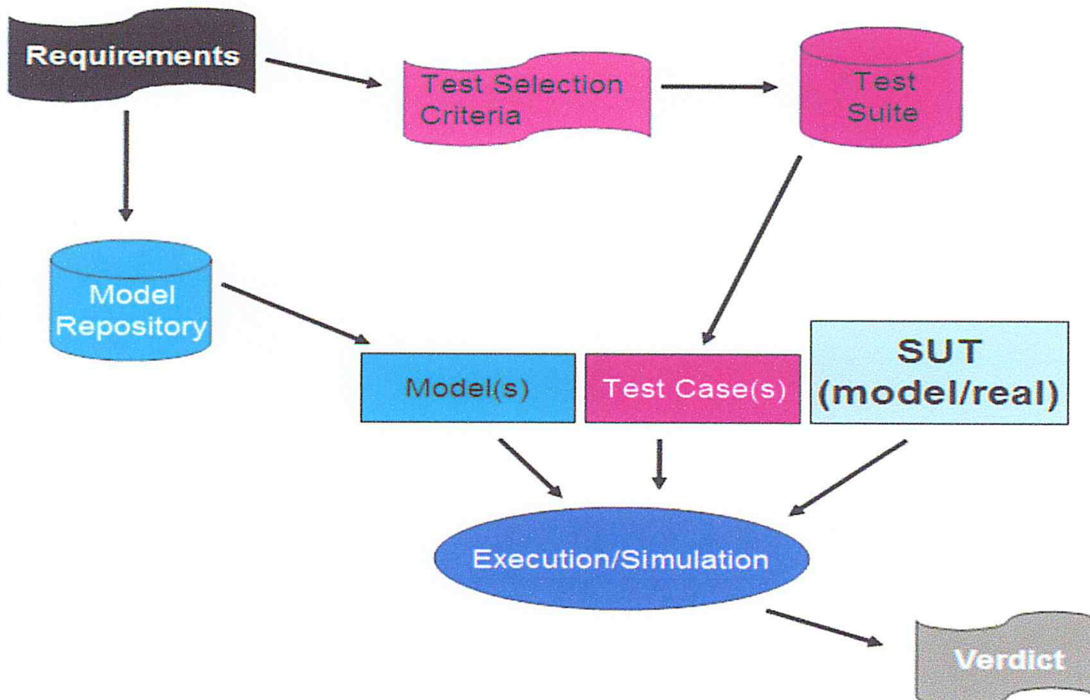


Figure I-5. Processus graphique fondée sur un modèle [9]

- Un modèle de la SUT est construit sur la spécification des exigences existantes avec désiré niveaux d'abstraction ;
- Les critères de sélection des tests sont définis avec un objectif de détection des défauts graves et probables à un coût acceptable. Ces critères décrivent de manière informelle les directives pour un test suite ;
- Les critères de sélection des tests sont ensuite traduits en spécifications de cas de test. C'est une activité où un document textuel est "opérationnel". Les générateurs automatiques de cas de test tombent dans cette étape d'exécution ;
- Une suite de tests est 'générée' basée sur le modèle sous-jacent et le cas de test Caractéristiques ;
- Les cas de test de la suite de tests générée sont exécutés sur le SUT après mécanisme de priorisation et de sélection. Chaque exécution entraîne un verdict de «réussite» ou ' Échoué ' ou ' non concluant '.

I.7 Conclusion

Nous avons définis dans ce chapitre les systèmes de systèmes (SoS), les modèles d'architecture, ainsi que les méthodologies pour les SoS.

L'extension d'UML pour mieux représenter les problèmes du système a conduit à création de SysML. La création de SysML est beaucoup plus qu'une simple adaptation d'UML; c'est un langage de modélisation beaucoup plus complet pour décrire les systèmes. Bien que ni UML ni SysML ne soient des frameworks d'architecture, ils spécifient tous deux un langage de modélisation qui peut être utilisé dans les cadres l'architecture pour créer des produits ou des modèles de vue spécifiques.

Dans le prochain chapitre nous allons voir la transformation de modèle, et dans le chapitre trois nous allons présenter en détail les deux langages de modélisation SysML et DEVS.

CHAPITRE II : Transformation De Modèle

II.1 Introduction

L'IDM est née d'un besoin simple : répondre aux exigences grandissantes du génie logiciel en termes de qualité des systèmes, de portabilité et d'interopérabilité. Pour cela, l'IDM propose comme solution de regrouper des concepts au sein d'abstractions de systèmes : les modèles. Nous allons présenter dans ce chapitre les concepts de base de l'ingénierie dirigée par les modèles. Nous illustrons aussi la métamodélisation, et la transformation de modèles.

II.2 Ingénierie dirigée par les modèles

Après avoir abordé le monde de la modélisation et simulation avec des concepts, des formalismes de bas niveau d'abstraction, dont le formalisme DEVS, et introduit la notion d'interopérabilité entre ces formalismes, abordons maintenant un autre aspect des modèles à travers ce qui est devenu en quelques années une discipline incontournable du génie logiciel : l'Ingénierie Dirigée par les Modèles (ou IDM).
L'IDM (en anglais *Model Driven Engineering*, abrégé en MDE, parfois *Model Driven Development*, abrégé en MDD) est l'appellation générique pour une discipline particulière du génie logiciel qui regroupe plusieurs familles d'approches partageant des pratiques, des méthodes, des techniques communes, souvent implémentées au sein d'outils de développement. La finalité de l'IDM est double : d'une part améliorer la qualité des systèmes, et d'autre part simplifier leur mise en œuvre et leur maintien sur une ou plusieurs plateformes [1].

L'Ingénierie Dirigée par les Modèles (partie droite de la figure 6) a été définie pour pallier les problèmes présents dans les approches classiques de développement. Dans une approche IDM, les niveaux d'analyse et de conception d'un système font partie intégrante du processus automatisé de développement. La figure7 montre l'impact d'une approche IDM dans le processus de développement d'un système [2].

Dans le cas où les nouvelles exigences sont remontées au niveau de la spécification des besoins, cela n'aura pas pour finalité la production d'un système entièrement en adéquation avec les nouveaux besoins. En effet, dans une approche classique de développement, les nouveaux besoins doivent être traduits manuellement de la plus haute couche d'abstraction jusqu'à la couche de code [2].

C'est pour cette raison que dans les cycles classiques de développement les nouveaux besoins sont directement pris en compte dans les couches correspondantes au code du système. Avec une telle pratique nous obtenons une désynchronisation entre les hautes couches d'abstraction correspondant à la spécification d'un système et les basses couches correspondant au code du système [2].

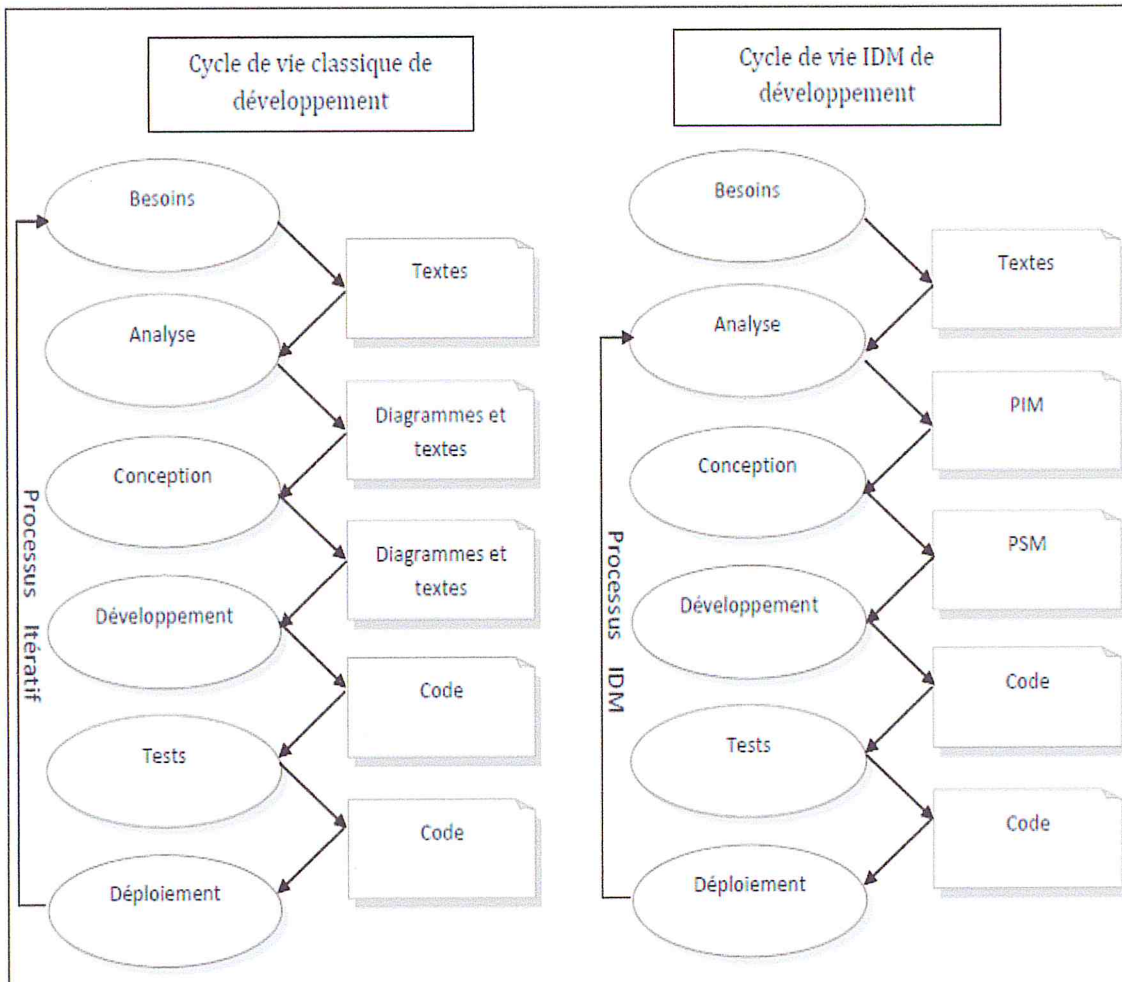


Figure II-1. L'impact d'une approche IDM dans le processus de développement d'un système [2]

II.3 Métamodelisation

II.3.1 Définition

La métamodélisation [2] est l'activité consistant à définir le méta modèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser.

L'Ingénierie Dirigée par les Modèles repose sur les trois formalisations suivantes :

(i) Modèles,

(ii) Méta-modèles,

(iii) Transformation de modèles.

Dans les phases suivantes nous présenterons ces trois concepts.

II.3.2 Notion de modèles et méta-modèles

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.).

L'IDM vise donc, de manière plus radicale que pouvaient l'être les approches des patterns et des aspects, à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc.

C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet. Alors que l'approche objet est fondée sur deux relations essentielles, « InstanceDe » et « HériteDe ». (Voir la figure 7)

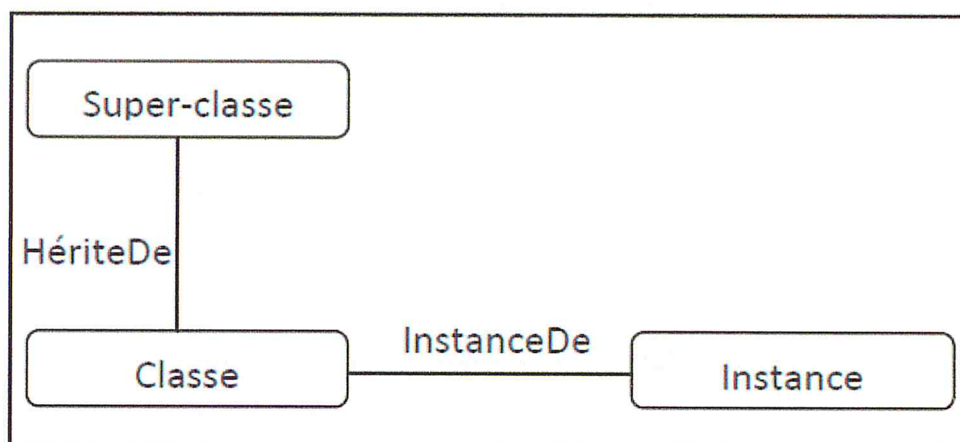


Figure II-2. Notions de base en technologie des objets [2]

- Les modèles dits PIM (Platform Independent Model). C'est un modèle informatique qui représente une vue partielle d'un CIM. Le PIM représente le logique métier spécifique au système ou le modèle de conception. Il représente le fonctionnement des entités et des services.

- Les modèles dits PSM (Platform Specific Model). Un modèle PSM est dépendant de la plateforme technique spécifiée par les architectes d'un système. Le PSM sert essentiellement de base à la génération de code exécutable vers la ou les plateformes techniques. Le PSM décrit comment le système utilisera cette ou ces plateformes.

Dans les sections suivantes nous présenterons l'ensemble de ces types de modèles.

II.3.2.2. Méta-modèles

Dans une approche IDM, les spécificités techniques sous-jacentes à un modèle sont écartées. L'IDM préconise l'utilisation d'un mécanisme standard et abstrait pour définir des modèles. Ce mécanisme abstrait est dénoté par le terme méta-modèle. Ainsi dans la pratique, tout modèle défini par l'intermédiaire d'une approche IDM doit posséder un méta-modèle.

Nous pouvons vulgariser la notion de méta-modèle en le définissant comme étant le modèle d'un 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 « Conformeà » (Voir la figure 8) et nommée X sur la figure 9. 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 (X). 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.

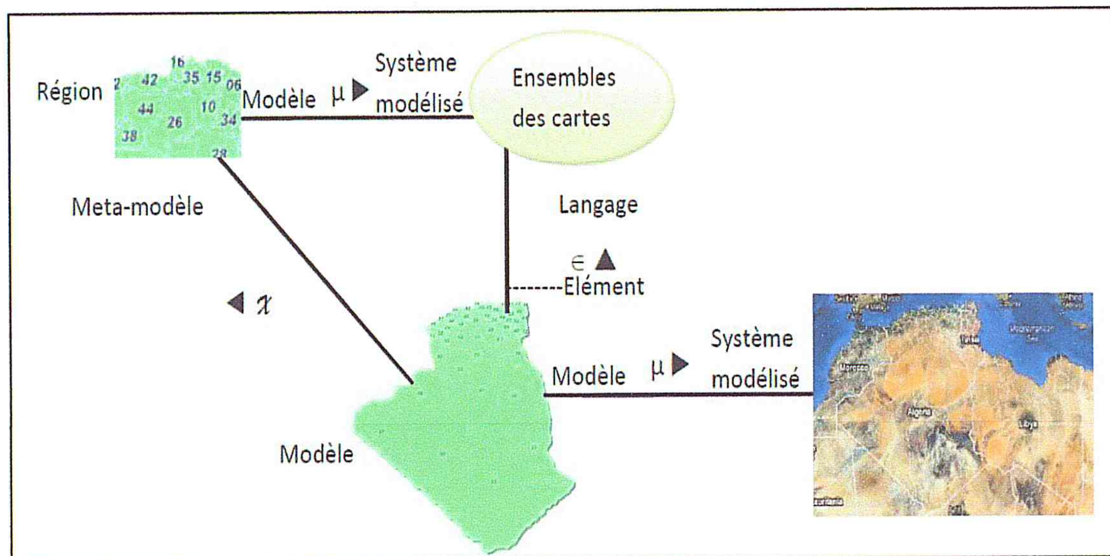


Figure II-4. Relation entre système, modèle, méta-modèle et langage [2]

II.3.2.3. Méta-méta-modèles

L'intérêt essentiel des méta–modèles est de faciliter la fragmentation des représentations selon les préoccupations. Lorsque l'on considère un système donné, on peut travailler avec différentes vues de ce système, chacune de celles–ci étant caractérisée de façon précise par un méta–modèle donné. [2]

Quand plusieurs modèles différents ont été extraits du même système à l'aide de méta–modèles différents, ces modèles restent liés et pourront être recomposés par la suite. Pour que ceci puisse être largement appliqué, il est nécessaire de disposer d'une organisation régulière des modèles composites.

Un postulat essentiel du MDE est de considérer que les méta–modèles sont des modèles et sont exprimés par un méta–modèle unique : le méta–méta–modèle.

La conformité des méta– modèles au méta–méta–modèle permet de comparer, transformer, regrouper, trouver les différences, etc. entre des modèles exprimés sur la base de formalismes différents.

Ce postulat essentiel permet d'éviter la fragmentation des modèles selon l'espace considéré. Pour supporter cette approche l'OMG a introduit une architecture 4 couches de méta-modélisation (Figure 10) :

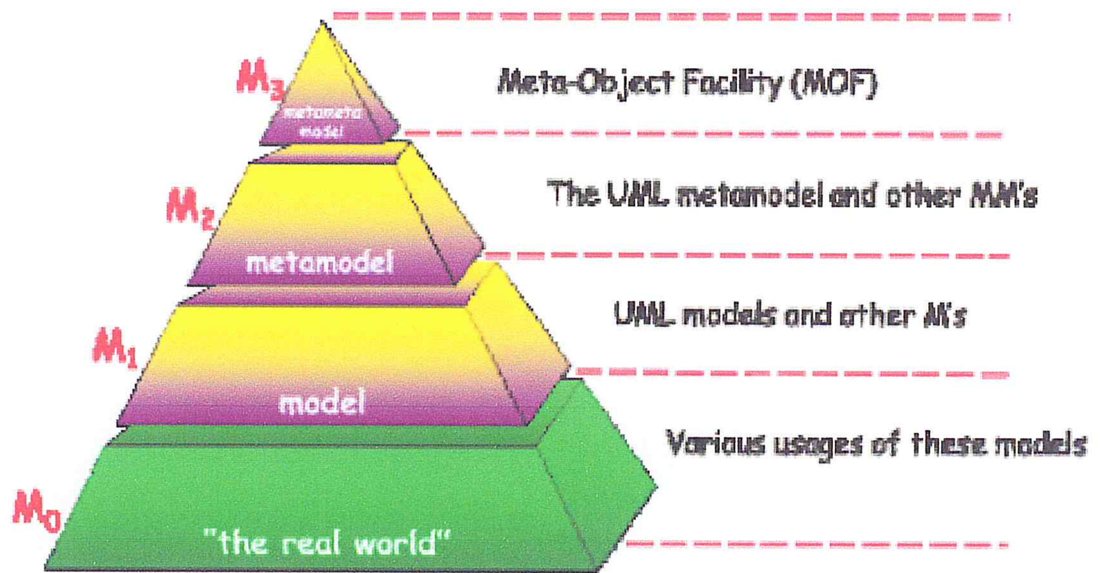


Figure II-5. La « pyramide » des « niveaux méta » de MDA

Généralement, on utilise la notation suivante pour représenter les différents niveaux de modélisation :
M0 : pour décrire le niveau des systèmes réels,
M1 : correspond au niveau des modèles de systèmes,
M2 : pour le niveau des méta–modèles,
M3 : le méta–méta–modèle.

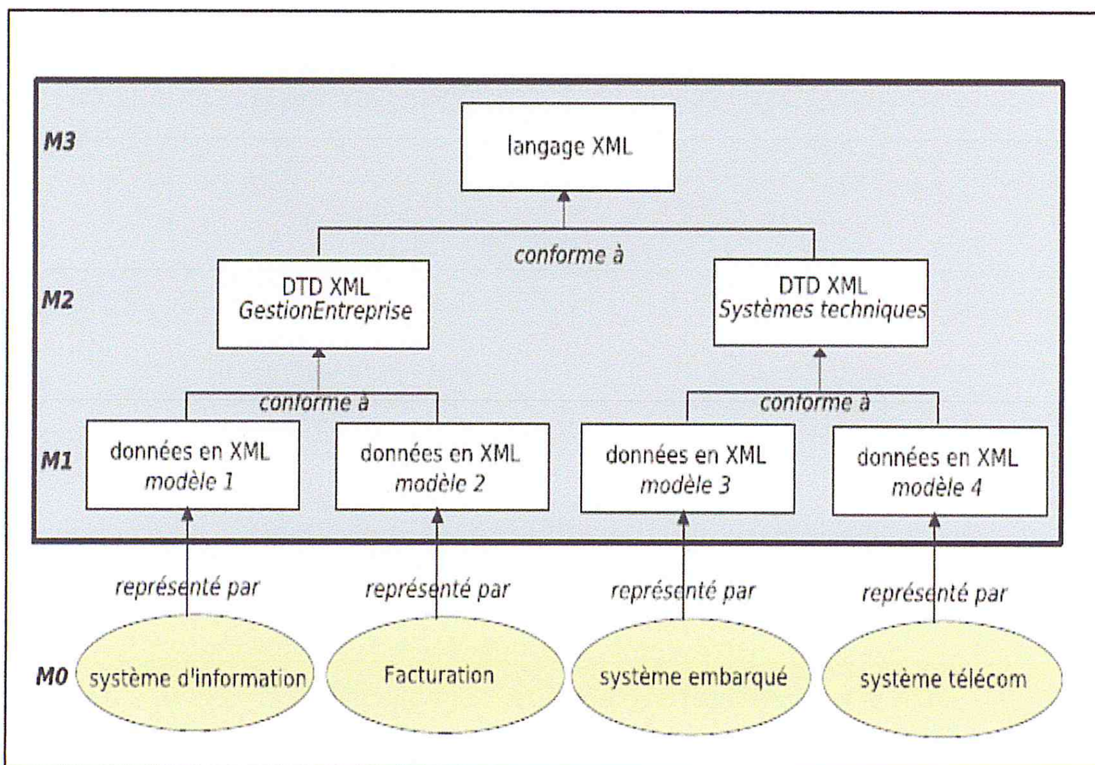


Figure II-6. Exemple d'organisation des modèles XML. [2]

Cette hiérarchie (Voir figure11) se retrouve quels que soient les espaces technologiques considérés (Ecore dans le domaine technologique d'Eclipse, les grammaires, les schémas XML et le MDA de l'OMG). Par exemple, dans le cas du langage XML, on a, au niveau M0, les données du système, au niveau M1 les données modélisées en XML, au niveau M2 les DTD XML et au niveau M3 le langage XML lui–même. La figure 7 donne un exemple d'organisation des modèles sur la base du langage XML. [2]

II.4 Transformation de modèles

II.4.1 Définition et principe de base

La définition la plus générale et qui fait l'unanimité au sein de la communauté IDM, consiste à dire qu'une transformation de modèles est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources. Dans l'approche par modélisation, cette transformation se fait par l'intermédiaire de règles de transformations qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible.
En réalité, la transformation se situe entre les méta-modèles source et cible qui décrivent la structure des modèles cible et source.

Etant donné un modèle source dans un langage L1, (tel que UML) et un modèle cible dans un langage L2 (tel que Java), il s'agit dans cette étape d'élaborer une mise en correspondance des concepts de L1 à ceux de L2 (ex. une classe UML correspond à une ou plusieurs classes Java). Dès lors, on a recours à la technique de métamodélisation pour mettre en place une base de règles exhaustive et générique.
Le moteur de transformation de modèles prend en entrée un ou plusieurs modèles sources et crée en sortie un ou plusieurs modèles cibles. Une transformation des entités du modèle source met en jeu deux étapes. La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs méta-modèles, ce qui induit l'existence d'une fonction de transformation applicable à toutes les instances du méta-modèle source.

La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé moteur de transformation ou d'exécution. La figure 12 illustre ces deux étapes d'une transformation de modèles.

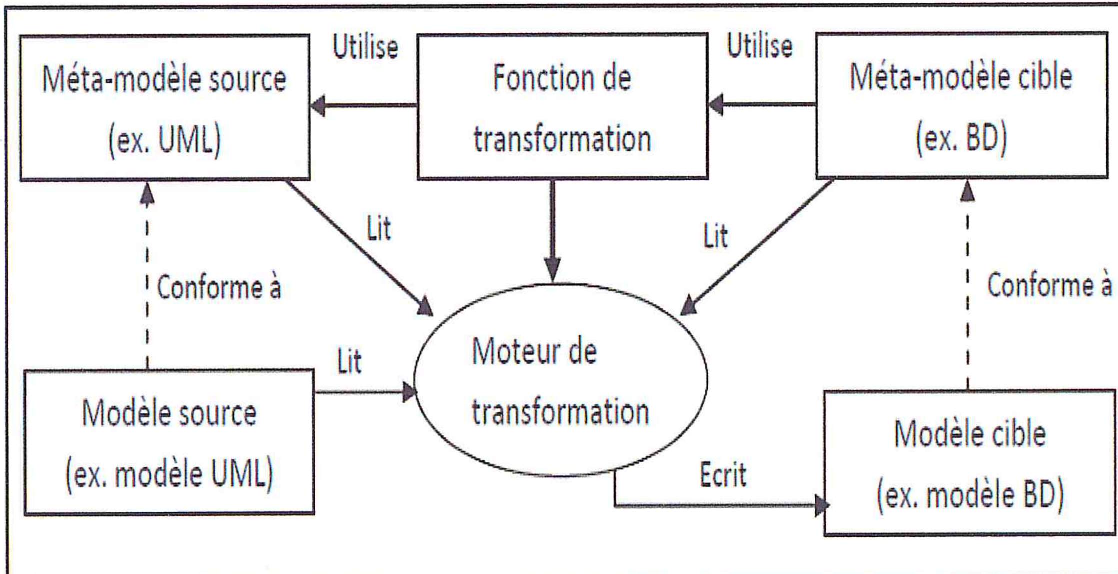


Figure II-7. Schéma de base d'une transformation de modèles [2]

II.4.2 Taxonomie des transformations

Partant de la nature des méta-modèles source et cible, on distingue des transformations dites endogènes et exogènes combinées à des transformations dites verticales et horizontales. [2]

II.4.2.1 Transformation horizontale de modèle

Une transformation de modèle horizontal est une transformation, où le modèle source et le modèle cible appartiennent au même niveau d'abstraction. Un exemple pour ce type particulier de transformation est le refactoring, où le modèle cible, par rapport au modèle source, change dans sa structure interne sans changer le comportement.

II.4.2.2 Transformation verticale de modèle :

Contrairement à la transformation de modèle horizontale, différents niveaux d'abstraction sont utilisés dans le modèle source et cible. Cette transformation est appelé aussi raffinement parce que des informations supplémentaires sont ajoutées.

II.4.2.3 Transformation endogène de modèle :

C'est la transformation qui affecte les modèles exprimés dans le même langage. Les deux modèles source et cible sont conformes au même méta-modèle.

II.4.2.4 Transformation exogène de modèle :

Contrairement aux transformations endogènes, les transformations exogènes sont exprimées entre les modèles conformes à des différents méta-modèles. Les transformations exogènes peuvent également être appelées translation. Un exemple pour les transformations exogènes est la génération de code ou de documentation ou l'ingénierie inverse par exemple la décompilation. Par exemple, un diagramme de classes UML peut être traduit en code Java. La traduction d'un code Java en diagramme de classe UML est un exemple pour le reverse engineering. La figure13 résume les combinaisons possibles entre transformations de modèles.

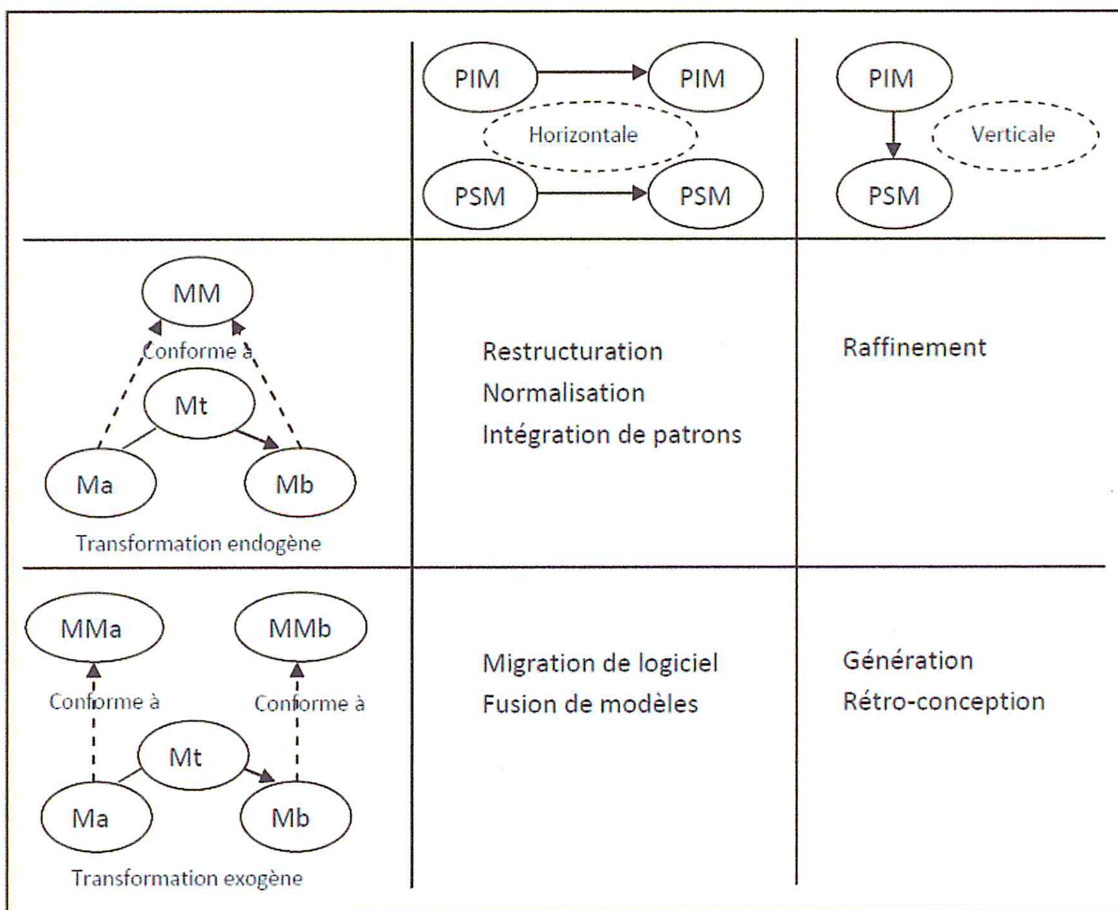


Figure II-8. Taxonomie des transformations de modèles

II.4.3 Les outils de transformations

II.4.3.1 Les langages et outils dédiés à la transformation de modèles

La définition de transformations a pour objectif de rendre les modèles opérationnels dans une approche IDM, ce qui augmente considérablement la productivité des applications. La transformation de modèles est une opération très importante dans toute approche orientée modèle. En effet, les transformations assurent les opérations de passage d'un ou plusieurs modèles d'un niveau d'abstraction donné vers un ou plusieurs autres modèles du même niveau (transformation horizontale) ou d'un niveau différent (transformation verticale). Le modèle transformé est appelé *modèle source* et le modèle résultant de la transformation est appelé *modèle cible.* [4]

Dans cette catégorie, on retrouve les outils et langages conçus spécifiquement pour faire de la transformation de modèles et prévus généralement pour être intégrables dans les environnements de développement standard (Eclipse par exemple). Parmi ces langages, nous pouvons citer les deux langages QVT (Query, Views, Transformation) et ATL (ATLAS Transformation Langage pour langage de transformation.

Dans la suite de cette section, nous présentons essentiellement le langage ATL qui est représentatif de cette catégorie des langages dédiés aux transformations, et fait l'objet de nombreuses expérimentations dans la communauté IDM. Outre de tout ça, nous l'avons utilisé comme un langage de transformation dans la partie contribution, en particulier dans le chapitre implémentation.

II.4.3.1.1 QVT (Query, Views, Transformation)

Les transformations de modèles étant au coeur de l'IDM, un standard dénommé QVT (Query, Views, Transformation) a été établi pour modéliser ces transformations. Ce standard définit le métamodèle permettant l'élaboration des modèles de transformation. La dernière version du standard QVT présente un caractère hybride dans le sens qu'elle est composée de trois langages de transformation différents (Figure 14 ci-dessous). La partie déclarative de QVT est définie par les langages Relations et Core ayant des niveaux d'abstraction différents. Relations est un langage orienté utilisateur permettant de définir des transformations à un niveau d'abstraction élevé. Il a une syntaxe textuelle et graphique. Le langage Core forme l'infrastructure de base pour la partie déclarative ; c'est un langage technique de bas niveau

défini par une syntaxe textuelle. Il sert à spécifier la sémantique du langage Relations, sous la forme d'une transformation Relations2Core. La vision déclarative passe par une association de patterns, côté source et cible pour exprimer la transformation, en laissant l'outil se débrouiller seul.

Manifestement, elle permet une expression plus simple des transformations de type mappings.

La composante impérative de QVT est supportée par le langage Operational Mappings. La vision impérative impose une navigation explicite et une création explicite des éléments du modèle cible. Le langage Operational Mappings étend les deux langages déclaratifs de QVT en ajoutant des constructions impératives (séquence, sélection, répétition, etc.) ainsi que des constructions OCL à effet de bord. Les langages de style impératif sont mieux adaptés pour des transformations complexes qui comprennent une composante algorithmique importante. Par rapport au style déclaratif, ils ont l'avantage de gérer les cas optionnels dans une transformation. Enfin, QVT propose un deuxième mécanisme d'extension pour spécifier des transformations, en permettant d'invoquer des fonctionnalités de transformations implémentées dans un langage externe (*Black Box*). [4]

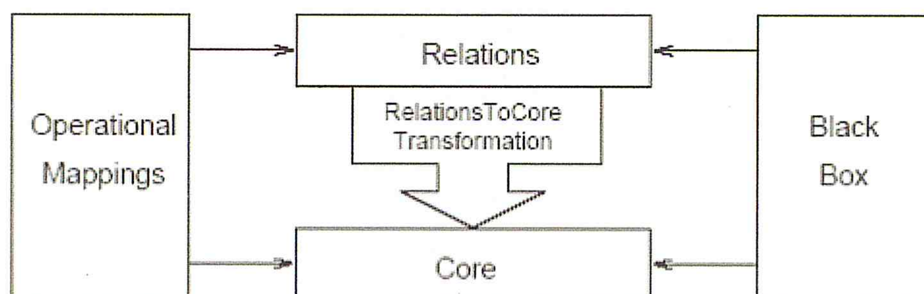


Figure II-9. Architecture du standard QVT

II.4.3.1.2 ATL

II.4.3.1.2.1 Présentation d'ATL

ATL est l'acronyme d'ATLAS *Transformation Language* ; c'est un langage à vocation déclarative, mais qui est en réalité hybride et permet de faire des transformations de modèles aussi bien endogènes qu'exogènes. Les outils de transformation liés à ATL sont intégrés sous forme de plug-in ADT (ATL Development Tool) pour l'environnement de développement

- La correspondance entre les modèles définis en MOF.
- L'interrogation d'un modèle MOF afin de filtrer et sélectionner des éléments du modèle source à transformer.
- La création des vues sur des méta-modèles MOF, une vue sur un système modélisé est un modèle dérivé du modèle du système, ne révèle que certains aspects de ce dernier.

Un langage de transformation peut être déclaratif, impératif ou hybride.

Dans la programmation déclarative, on décrit d'une part les données du problème à traiter et d'autre part les contraintes sur ces données. Le programme s'exécute à partir de la situation courante décrite par les données tout en respectant les contraintes.
Le langage déclaratif décrit ce qu'on devrait avoir à l'issue d'un certain nombre de données initiales. XSLT (Extensible Stylesheet Language Transformation), est un exemple de langage déclaratif de transformation.

Par opposition à un programme déclaratif, un programme impératif décrit comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer.

Un langage hybride regroupe à la fois les paradigmes de programmation déclarative et impérative : l'ordre d'exécution des modules doit être spécifié tandis qu'au sein d'un même module, la détermination de l'ordre d'exécution des règles n'est pas à la charge de l'utilisateur.

La règle de transformation est la construction élémentaire en ATL pour exprimer la logique de transformation. Les règles ATL peuvent être soit *déclaratives* soit *impératives*. [14]

- **Règles déclaratives : matched rules**

Une matched rule est composée d'un motif source et d'un motif cible. Le motif source d'une règle définit un ensemble de types source (venant des métamodèles source) et une garde sous la forme d'une expression OCL booléenne. Un motif source est évalué en un ensemble de tuples dans les modèles source.
Le motif cible est composé d'un ensemble d'éléments. Chacun de ces éléments définit un type cible (venant d'un métamodèle cible) et un ensemble d'affectations appelées bindings. Un binding fait référence à une propriété (attribut ou référence) du type et spécifie une expression dont la valeur est utilisée pour initialiser la propriété. [14]

➢ Le morceau de code suivant montre une règle déclarative simple :

```
1  rule PersistentClass2Table{
2    from
3      c : SimpleClass!Class (
4        c.is_persistent and c.parent.oclIsUndefined()
5      )
6    to
7      t : SimpleRDBMS!Table (
8        name <- c.name
9      )
10 }
```

Figure II-10. Exemple de règle déclarative

Le nom de la règle (PersistentClass2Table) est donné après le mot-clé rule (ligne 1). Le motif d'entrée dé_nit une variable de type SimpleClass!Class (ligne 3). La garde (ligne 4) spéci_e que seules les classes persistantes sans class mère sont retenues.

Le motif cible contient un élément de type SimpleRDBMS!Table (lignes 7-9). Cet élément a un binding (ligne 8) qui dé_nit une expression utilisée pour initialiser l'attribut name (le nom de la table).

Le symbole <- est utilisé pour délimiter la propriété à initialiser à gauche de l'expression servant à l'initialiser à droite. Il y a plusieurs types de règles déclaratives qui diffèrent par la manière dont elles sont déclenchées :

- ***Les règles standard (standard rules)*** sont appliquées une seule fois pour chaque tuple correspondant à leur motif d'entrée trouvé dans les modèles source.

➢ Le morceau de code suivant montre matched standard rule :

```
rule PersistentClass2Table{
                 from
Source            c : SimpleClass!Class (
pattern           c.is_persistent and
                 c.parent.oclIsUndefined()
                  )
                 to
Target            t : SimpleRDBMS!Table (
pattern           name <- c.name
                  )
                 }
```

Figure II-11. matched standard rule

- ***Les règles paresseuses (lazy rules)*** sont déclenchées par les autres règles. Elles sont appliquées sur chaque tuple autant de fois qu'il est référencé par les autres règles. Ceci signifie qu'une règle paresseuse peut être appliquée plusieurs fois sur un même tuple en produisant à chaque fois un nouvel ensemble d'éléments cible.

➢ Le morceau de code suivant montre *lazy rule* :

```
lazy rule R1 {
        from
                s : Element
        to
                t : Element (
                   value <- [R2.t]s
                )
}
```

Figure II-12. *lazy rule*

- ***Les règles paresseuses uniques (unique lazy rules)*** sont aussi déclenchées par les autres règles. Elles sont en revanche appliquées une unique fois pour chaque tuple. Si une règle paresseuse unique est déclenchée plus tard sur le même tuple, les mêmes éléments cible, créés la première fois, sont utilisés.

➢ Le morceau de code suivant montre *unique lazy rules* :

```
unique lazy rule Feature2Column {
    from
        trace : Sequence(OclAny)
    to
        col : SimpleRDBMS!Column (
            name <- trace->iterate(e; acc : String = '' |
                acc + if acc = ''
                    then ''
                    else '_' endif + f.name),
            type <- trace->last().type
        )
}
```

Figure II-13. *Unique lazy rules*

- **Règles impératives**

ATL contient une partie impérative. Les deux principales constructions impératives d'ATL sont : [14]

- ***Les règles appelées (called rules)*** qui sont similaires à des procédures. Elles sont invoquées par leur nom et peuvent prendre des arguments. Leur implémentation peut être native ou spécifiée en ATL. Dans ce dernier cas, la définition ressemble à une règle déclarative sans motif d'entrée. En effet, la règle n'est pas exécutée sur des tuples reconnus dans les modèles d'entrée mais appelée.
- ***Le bloc impératif (action block)*** qui contient une séquence d'instructions impératives. Il peut être utilisé à la place de ou en combinaison avec un motif cible, dans des règles déclaratives ou impératives. Les instructions disponibles en ATL sont celles communément offertes par les langages impératifs. Ainsi, il y a, pour le contrôle de flux, une instruction conditionnelle, des boucles, mais aussi des affectations, etc.
  Si soit une règle appelée soit un bloc impératif est utilisé dans un programme ATL, ce programme n'est plus totalement déclaratif. Dans certains cas, il peut être utile d'interdire l'utilisation de ces constructions.

- Le morceau de code suivant montre une règle appelée et bloc d'action (Called rules and action block) :

```
helper def: id : Integer = 0;
   rule getId() {                              -- : Integer
      do { thisModule.id <- thisModule.id + 1; -- increment id
         thisModule.id;                        -- returns id
      }
   }
```

Figure II-14. Called rules and action block

- Le morceau de code suivant montre une règle hybride (règle déclarative avec bloc d'action) :

```
rule Test {
   from s : S!Test
   to t : T!Test
   do { t.id <- thisModule.getId();
   }
}
```

FigureII- 15. Règle déclarative avec bloc d'action

II.4.3.2 Outils de métamodélisation

La dernière catégorie des outils de transformation de modèles est celle des outils de méta-modélisation dans lesquels la transformation de modèles revient à l'exécution d'un méta-programme. Parmi ces outils nous présentons essentiellement Kermeta de l'IRISA-INRIA Rennes et EMF/Ecore (Eclipse-EMF-website) de la fondation Eclipse, qui sont représentatifs de cette catégorie d'outils demétamodélisation, et qui font l'objet de nombreuses expérimentations dans la communauté IDM.

Dans l'approche par programmation classique, on dit qu'un programme est composé d'un ensemble de structures de données combinées à des algorithmes. C'est sur la base de cette assertion que le langage de métamodélisation Kermeta a été élaboré. Une description en Kermeta est assimilable à un programme issu de la fusion d'un ensemble de métadonnées (EMOF) et du métamodèle d'action AS (Action Semantics) qui est maintenant intégré dans UML Superstructure.

Le langage Kermeta est donc une sorte de dénominateur commun des langages qui coexistent actuellement dans le paysage de l'IDM. Ces langages sont les langages de métadonnées (EMOF, ECORE,...), de transformation de modèles (QVT, ATL,...), de contraintes et de requêtes (OCL), et d'actions (Action Semantics, Xion).

Kermeta est un véritable langage de métamodélisation exécutable. En effet, le métamodèle de Kermeta est composé de deux packages, Core et Behavior correspondant respectivement à EMOF et à une hiérarchie de métaclasses représentant des expressions impératives. Ces expressions seront utilisées pour programmer la sémantique comportementale des métamodèles. En effet, chaque concept d'un métamodèle Kermeta peut contenir des opérations, le corps de ces opérations étant composé de ces expressions impératives (package Behavior). On peut ainsi définir la structure aussi bien que la sémantique opérationnelle d'un métamodèle, rendant ainsi, exécutables les modèles qui s'y conforment.

II.5 Conclusion

Nous avons définis dans ce chapitre l'ingénierie des modèles ainsi la métamodélisation et dans la dernière phase nous avons décrit les différents outils de transformation de modèles.

Dans le prochain chapitre, nous allons présenter les langages source et cible de la transformation.

CHAPITRE III : Les Langages de Modélisation

III.1 Introduction

Nous avons utilisé dans notre mémoire le langage SysML comme un langage source et le formalisme DEVS comme un langage cible (destination). Dans cette section, nous présentons essentiellement les deux langages SysML et DEVS.

III.2 SysML (System Modeling Langage)

SysML est un langage de modélisation des systèmes complexes. C'est un ensemble de diagrammes que l'on peut catégoriser de cette façon (Voir figure 21) [5] :

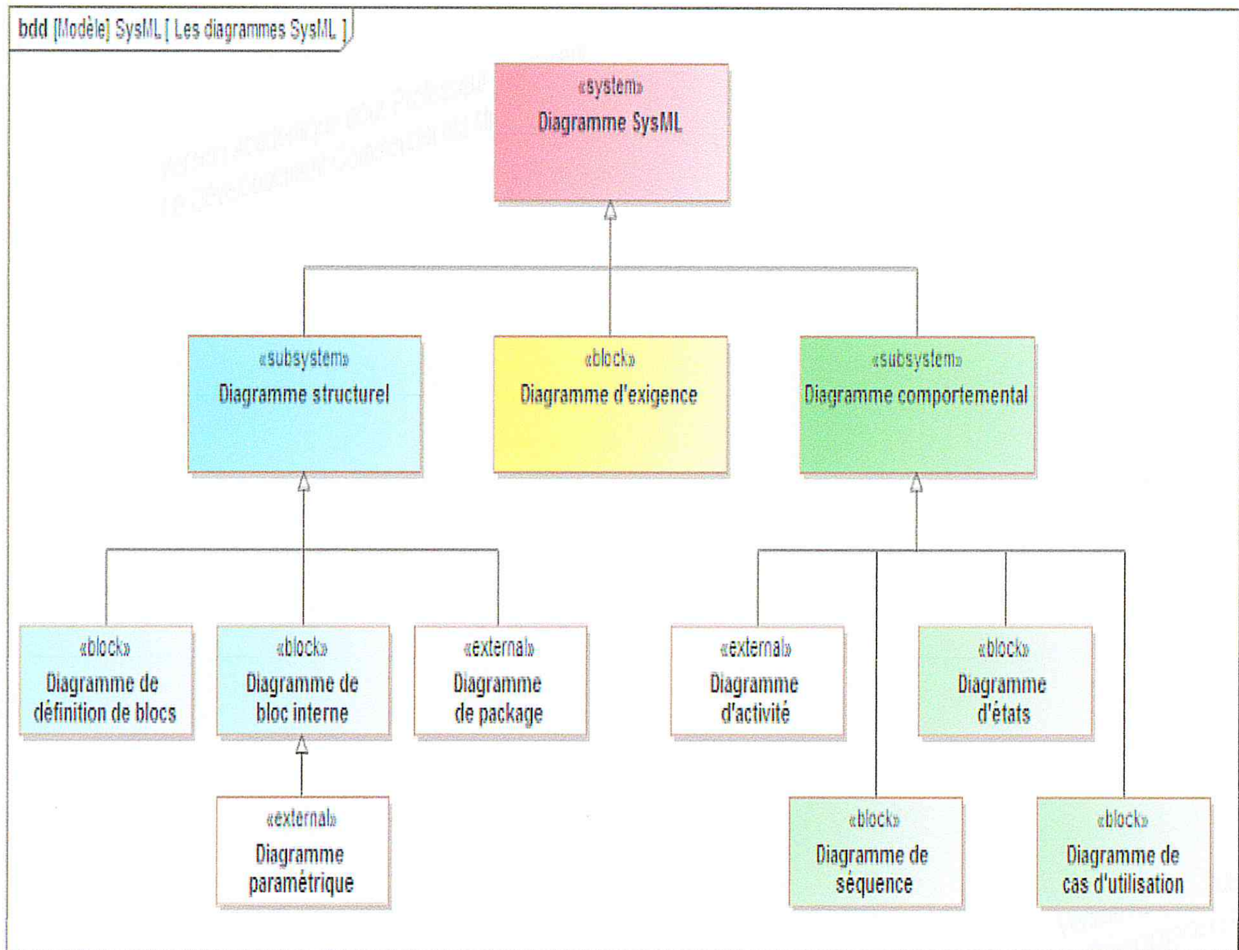


Figure III-1. Récapitulatif des diagrammes SysML [5]

III.2.1 Diagramme de définition de blocs (bdd)

Le diagramme de définition de blocs permet de définir une arborescence de blocs. Ces blocs pouvant modéliser un élément du contexte, une fonction ou un composant, le diagramme bdd aura une signification différente [5].

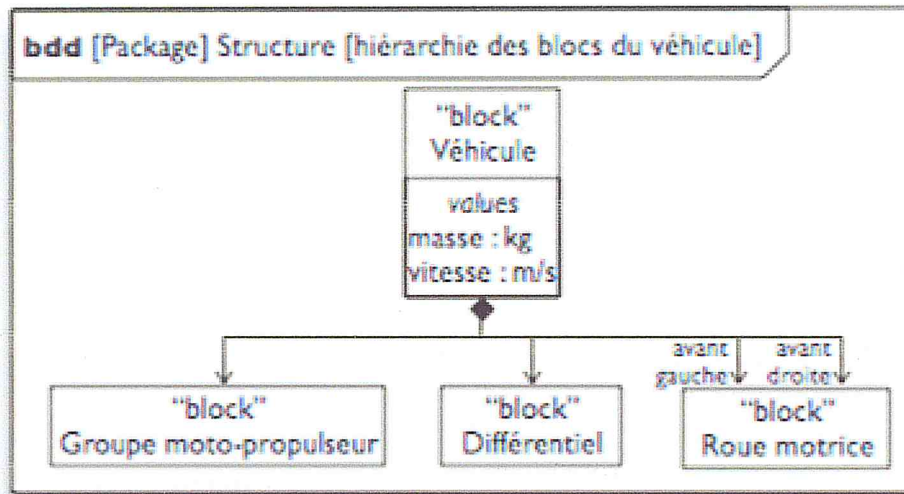


Figure III-2. Diagramme bdd simple [5]

La description de la syntaxe est fournie ci-dessous :

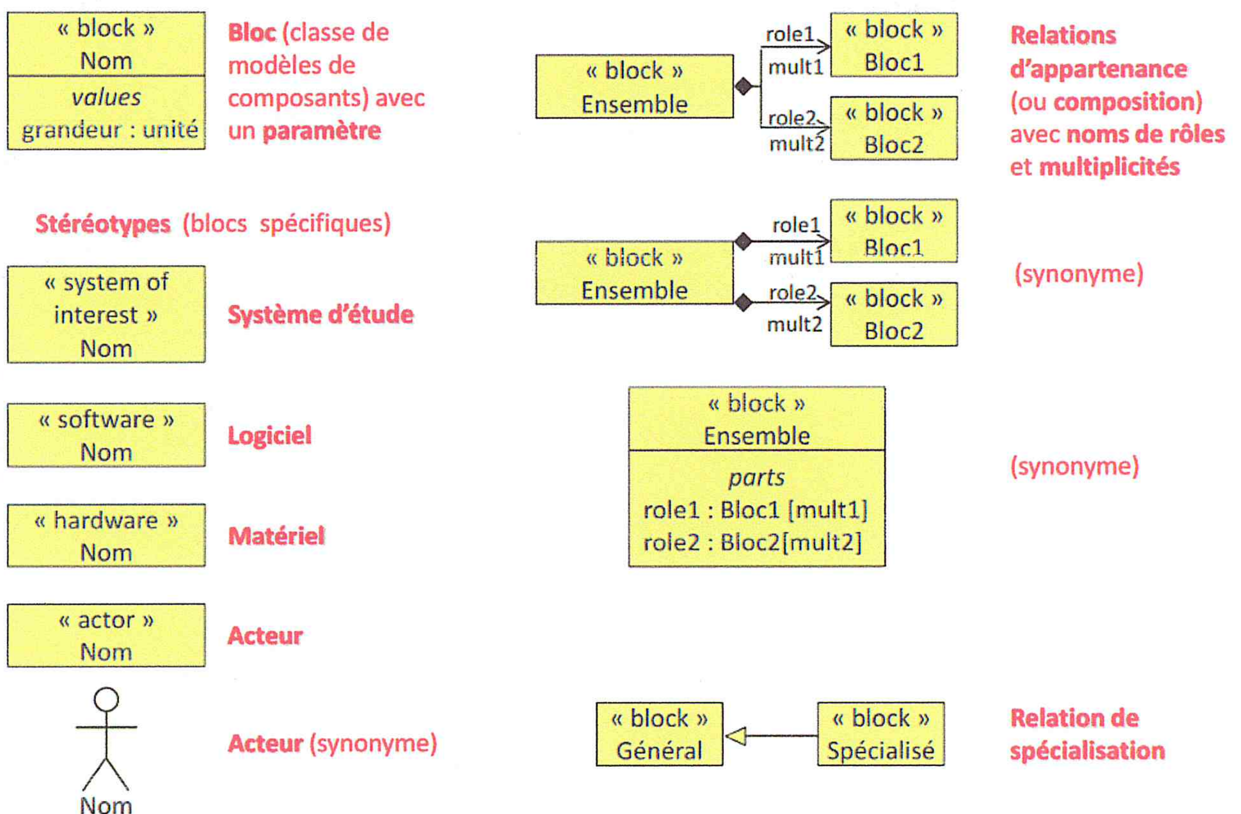


Figure III-3. Syntaxe du diagramme bdd

Il existe également la relation d'agrégation simple. Par rapport à la relation de composition (ou agrégation forte), elle est modélisée par un losange vide. La nuance principale concerne le lien entre les deux blocs. Pour une composition, la destruction de l'élément supérieur entraîne la destruction de l'élément inférieur, ce qui n'est pas le cas pour une agrégation simple [12].

SysML est un diagramme statique. Il montre les briques statiques : blocs, composition, associations, Il est utilisé pour décrire l'architecture matérielle du système.
Un bloc est une entité bien délimitée qui encapsule principalement des attributs (Variables d'état), des opérations (procédures comportementales), des contraintes, des ports (échange de flux avec l'extérieur) et des parts (sous-blocs internes) [12].

Le bloc SysML (block) constitue la brique de base pour la modélisation de la structure d'un système. Il peut représenter un système complet, un sous-système ou un composant élémentaire. Le bloc permet de décrire également les flots qui circulent à travers un système. Les blocs sont décomposables et peuvent posséder un comportement. On peut s'en servir pour représenter des entités physiques, mais aussi des entités logiques ou conceptuelles. Les propriétés sont les caractéristiques structurelles de base des blocs. Elles peuvent être de deux types principaux : les valeurs (value properties) décrivent des caractéristiques quantifiables en terme de value types (domaine de valeur, dimension et unité optionnelles).
Les parties (part properties) décrivent la hiérarchie de décomposition du bloc en termes d'autres blocs. Chaque bloc (ou type) définit un ensemble d'instances partageant les propriétés du bloc, mais possédant chacune une identité unique [12].

Dans un bdd, un bloc est représenté graphiquement par un rectangle découpé en compartiments. Le nom du bloc apparaît tout en haut, et constitue l'unique compartiment obligatoire. Tous les autres compartiments ont des labels indiquant ce qu'ils contiennent : valeurs, parties, etc.
Par la suite on à un exemple de diagramme de définition de bloc représente un vélo de promenade qui est composé de trois blocs, à chaque bloc a son propre caractéristique comme des propriétés ou opération contraint etc [12].

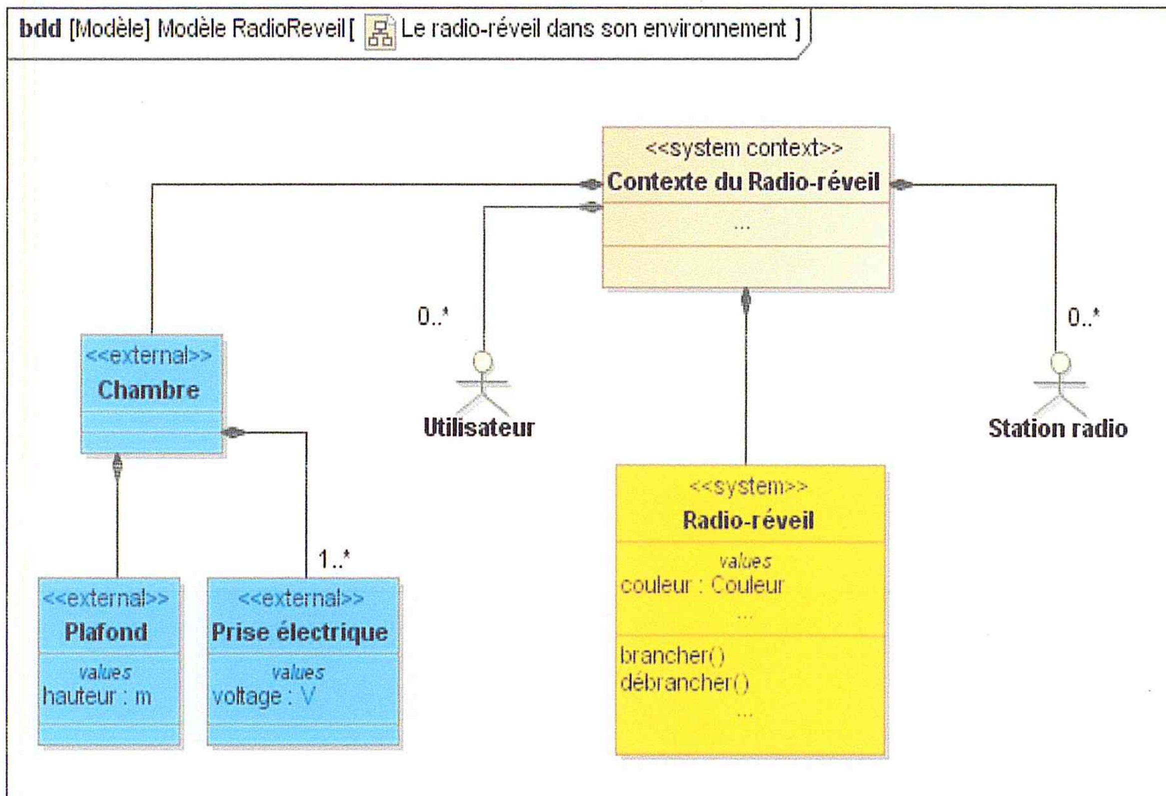


Figure III-4. Exemple de diagramme de définition de bloc

Il y a d'autre caractéristique qui compose le bloc, nous abordons sa définition [12] :

III.2.1.1 ValueType

Les valeurs (*value properties*) sont utilisées pour modéliser les caractéristiques quantitatives des blocs. Il est très important de bien définir les types de ces valeurs en termes de *value types* réutilisables.

Les *value types* sont basés sur les types de base proposés par SysML, à savoir :

- les types primitifs : *integer*, *string*, *boolean*, *real*, etc. ;
- les énumérations qui définissent un ensemble de valeurs nommées.
- les types structurés, permettant de définir plusieurs champs, chacun étant à son tour une valeur.

III.2.1.2 Partie

Revenons sur la notion de partie (*part property*) introduite au premier paragraphe.

Il s'agit d'une relation de composition entre blocs, aussi appelé relation « tout-partie », dans laquelle un bloc représente le tout, et les autres ses parties. Une instance du tout peut contenir plusieurs instances d'une partie, grâce à la notion de multiplicité, évoquée précédemment. Les

parties (*parts*) peuvent être listées dans un compartiment du bloc, avec le format suivant : nom partie : nom bloc [multiplicité].

III.2.1.3 Composition

La relation de composition entre blocs, dans laquelle un bloc représente le tout et les autres ses parties, peut également être représentée graphiquement. Le côté du tout est indiqué par un losange plein. La multiplicité de ce même côté ne peut être que 1 ou 0..1, car une instance de partie ne peut exister que dans une instance de tout au maximum à un moment donné.

III.2.1.4 Agrégation

La relation d'agrégation (losange vide) est beaucoup moins forte que la relation de composition (losange plein). En particulier, il n'y a pas de contrainte de multiplicité du côté du tout, et donc pas nécessité d'une structure stricte d'arbre. Mais il y a d'autres cas où l'agrégation est utile : pour représenter le fait que la contenance n'est pas vraiment structurelle et obligatoire, mais plus conjoncturelle.

III.2.1.5 Association

Un dernier type de relation entre blocs s'appelle l'association. L'association est une relation n'impliquant pas de contenance, comme la composition ou l'agrégation, mais une relation d'égal à égal.

III.2.1.6 Généralisation

Toutes les définitions qui apparaissent dans un bdd peuvent être organisées dans une hiérarchie de classification. Le but est souvent de factoriser des propriétés communes (valeurs, parties, etc.) à plusieurs blocs dans un bloc généralisé. Les blocs spécialisés « héritent » des propriétés du bloc généralisé et peuvent comporter des propriétés spécifiques supplémentaires. La généralisation en SysML se représente graphiquement par une flèche triangulaire pointant sur le bloc généralisé.

III.2.1.7 Opération

Tout bloc possède également des propriétés comportementales, les principales étant appelées opérations. Une opération représente soit :

- Une requête synchrone (l'émetteur est bloqué en attente d'une réponse), avec ses éventuels paramètres (ou arguments) en entrée, sortie, ou les deux ;
- Une requête asynchrone, aussi appelée réception (l'émetteur n'est pas bloqué en attente d'une réponse). Chaque réception est associée à un signal qui définit un message avec ses éventuels paramètres. Des réceptions définies dans des blocs différents peuvent

répondre au même type de signal, ce qui permet de réutiliser la définition de messages communs.

Les opérations sont montrées dans un compartiment supplémentaire avec leur signature (nom, paramètres et type de retour) :

nom_opération (liste de paramètres) : type_retour

La liste de paramètres est une liste d'éléments séparés par des virgules, chaque élément étant noté : direction nom_paramètre : type_paramètre

La direction pouvant prendre une des valeurs suivantes : in, out, inout.

Les réceptions sont souvent montrées dans un compartiment à part. Les signaux peuvent à leur tour être définis comme des blocs avec un mot-clé « signal ».

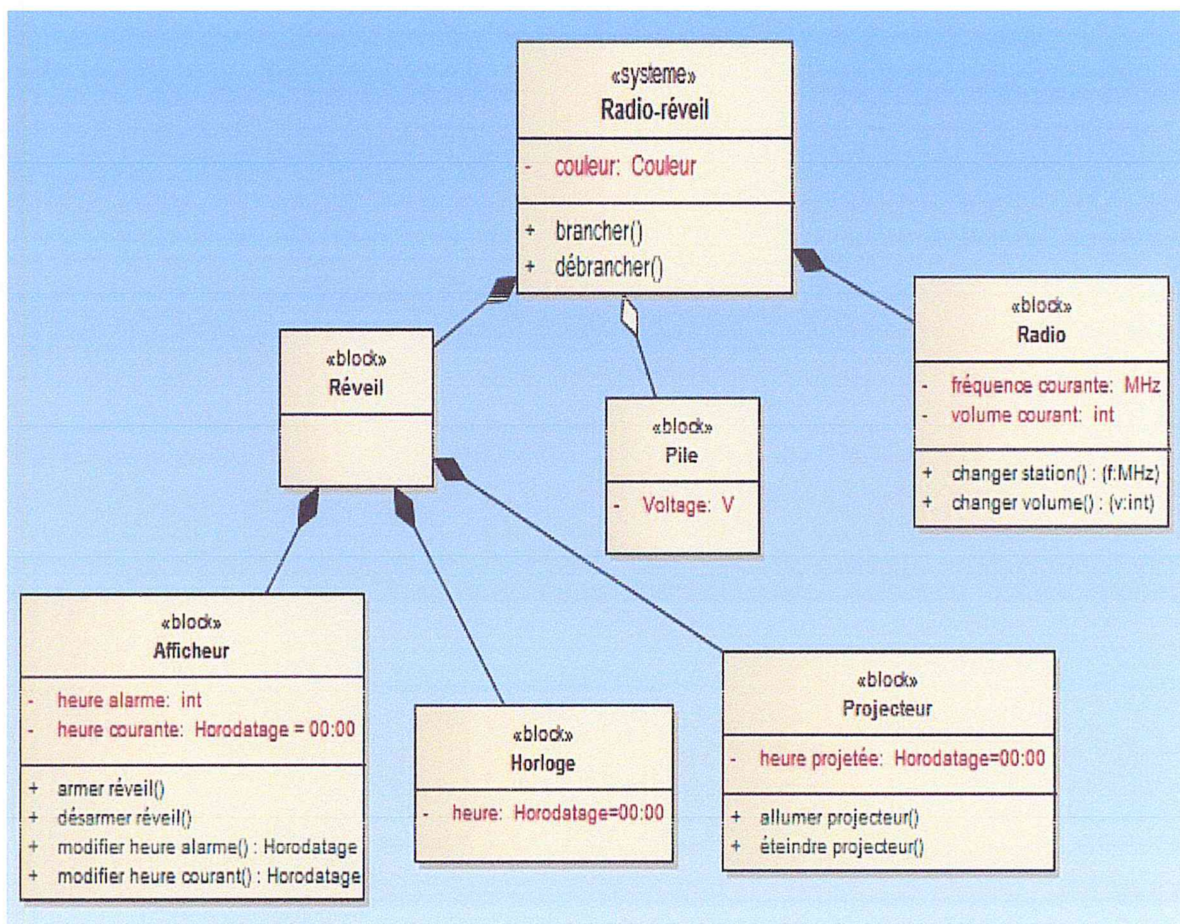


Figure III-5. Bdd Modèle RadioReveil

III.2.2 Le Diagramme de bloc interne (ibd)

Le diagramme ibd permet de montrer entre les différents sous-blocs d'un bloc et de spécifier les interfaces à l'aide de ports. Afin de distinguer l'utilisation de plusieurs sous-blocs d'une même catégorie, on utilise l'instance d'un bloc modélisée à l'aide des deux points [5] :

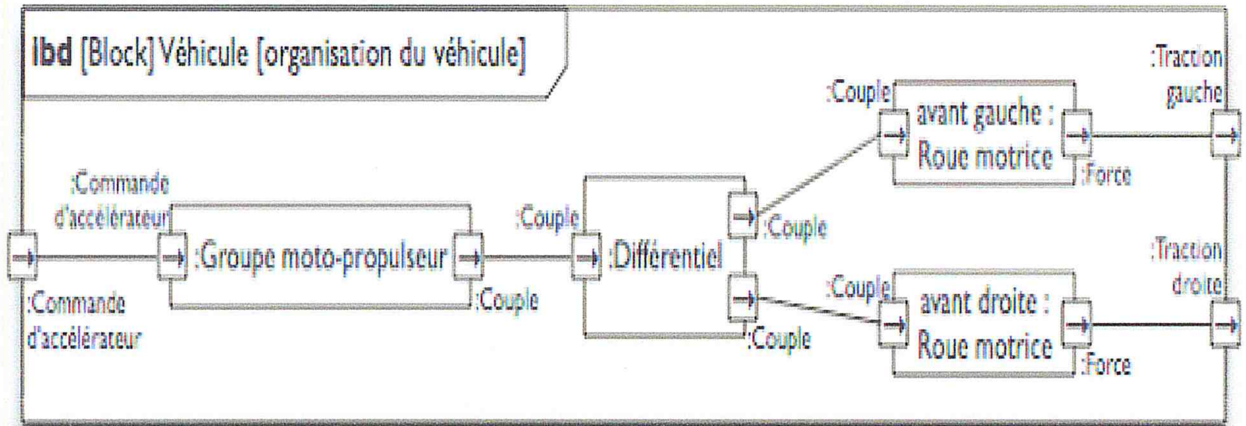


Figure III-6. Diagramme ibd simple [5]

La description de la syntaxe est fournie ci-dessous :

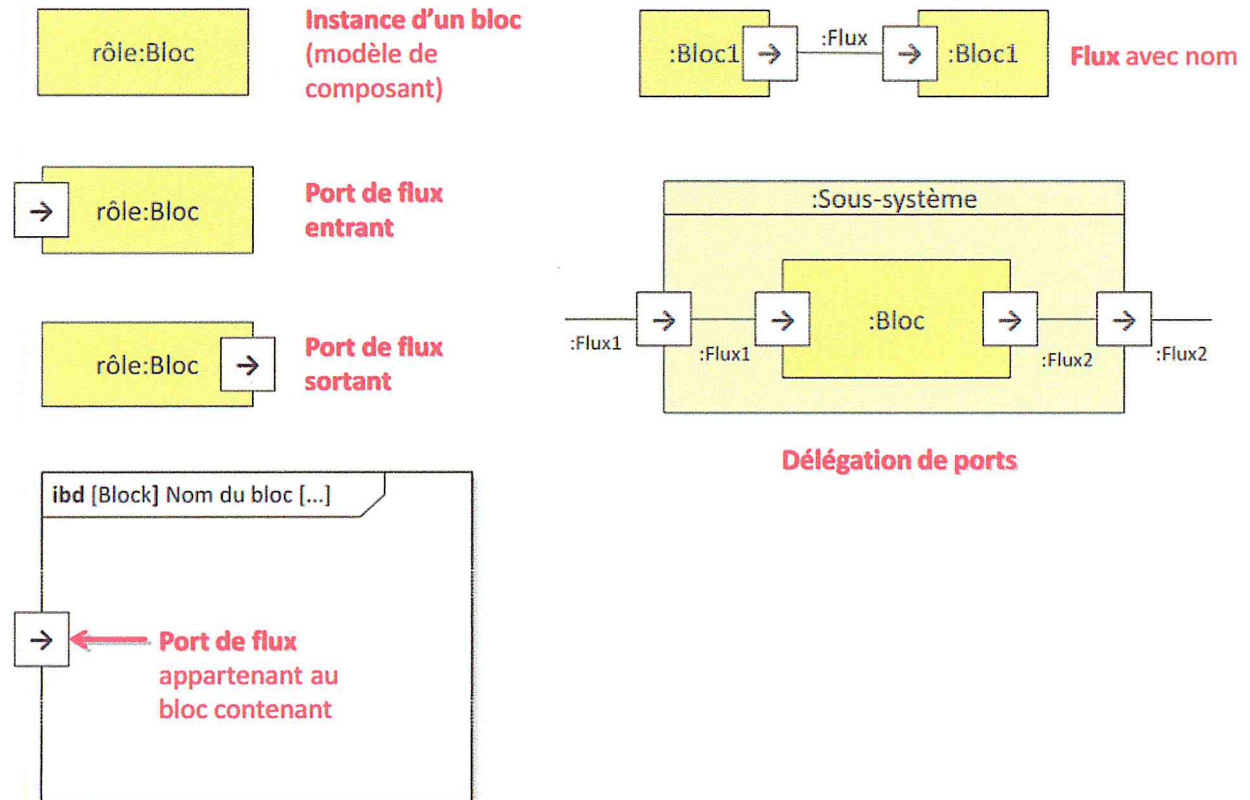


Figure III-7. Syntaxe du diagramme ibd

C'est un diagramme statique. Il est utilisé pour décrire l'architecture matérielle du système. Il montre l'organisation interne d'un élément statique complexe.

Il représente les instances des parts d'un bloc (objets). L'IBD est cadré à l'intérieur des frontières du bloc concerné. Les circulations de flux (MEI) entre les parts s'effectuent grâce aux connecteurs qui relient leurs ports.

L'IBD d'un bloc est défini à partir du BDD correspondant. Un flux entre ou sort d'une part via un port. [12]

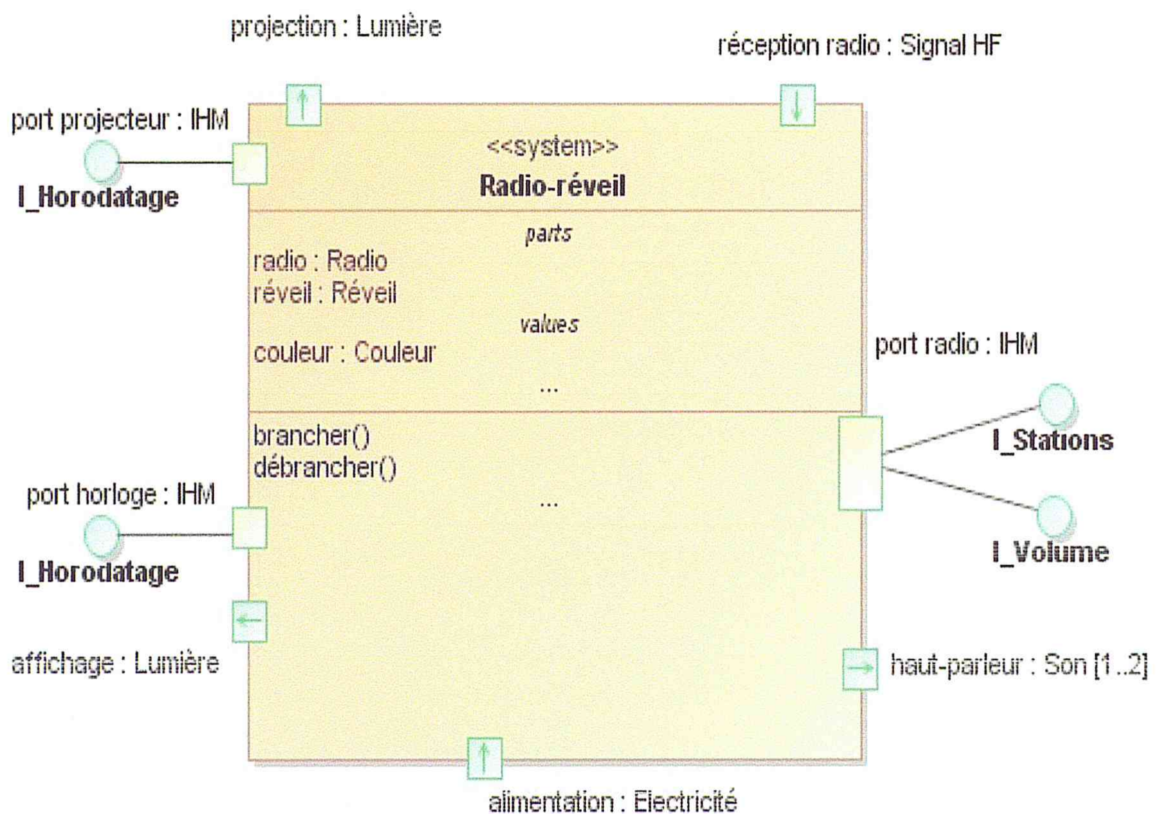


Figure III-8. Les éléments d'un diagramme de bloc interne.

III.2.2.1 Parties et connecteurs

On peut représenter la connexion entre les parties d'un bloc au moyen d'un diagramme de bloc interne. Ce diagramme montre principalement les relations entre éléments de même niveau, ainsi que les éventuelles multiplicités des parties.

Le connecteur est un concept structurel utilisé pour relier deux parties et leur fournir l'opportunité d'interagir, bien que le connecteur ne dise rien sur la nature de cette interaction. Les connecteurs permettent également de relier plus finement les parties à travers des ports (comme décrit au paragraphe suivant). L'extrémité d'un connecteur peut posséder une multiplicité qui décrit le nombre d'instances qui peuvent être connectées par des liens décrits par le connecteur.

Un connecteur peut être typé par une association. Il peut ainsi posséder une flèche unidirectionnelle, si l'association qui le type la possède. Son nom complet est de la forme : nom_connecteur : nom_association.

Une partie peut être connectée à plusieurs autres parties, mais il faut représenter un connecteur séparé pour chaque liaison. [12]

III.2.2.2 Ports et interfaces

III.2.2.2.1 Types de ports

Le diagramme de bloc interne permet également de décrire la logique de connexion, de services et de flots entre blocs grâce au concept de « port ». Les ports définissent les points d'interaction offerts (provided) et requis (required) entre les blocs. Un bloc peut avoir plusieurs ports qui spécifient des points d'interaction différents. Les ports peuvent être de deux natures :

- flux (flow port) : ce type de port autorise la circulation de flux physiques entre les blocs. La nature de ce qui peut circuler va des fluides aux données, en passant par l'énergie ;

- standard : ce type de port autorise la description de services logiques entre les blocs, au moyen d'interfaces regroupant des opérations. La distinction entre ces deux types de ports est souvent d'ordre méthodologique. Les flow ports sont bien adaptés pour représenter des flux continus d'entités physiques, alors que les ports standards sont bien adaptés à l'invocation de services, typiquement entre composants logiciels. Une combinaison des deux est souvent utile, mais les ports standards ne peuvent être connectés directement aux flow ports et réciproquement. Dans notre exemple de radio-réveil, les boutons de marche-arrêt du projecteur et de la radio sont typiquement des ports standards. L'entrée d'énergie électrique comme les ondes radio, la projection de lumière ou la diffusion de son sont typiquement des flow ports.

- **Flow ports,** Les ports de type « flux » sont soit atomiques (un seul flux), soit composites (agrégation de flux de natures différentes).

- **Item flow,** Les éléments de flot (*item flows*) permettent de décrire ce qui circule réellement sur les connecteurs, alors que les *flow ports* définissent ce qui peut circuler. La distinction n'est pas toujours utile, mais peut se révéler néanmoins très pratique dans certains cas.

III.2.2.2.2 Interface

Pour décrire un comportement basé sur l'invocation de services, le port standard est tout à fait adapté. Mais au lieu d'assigner directement des opérations aux ports, il est plus intéressant de les regrouper en ensembles cohérents appelés interfaces.

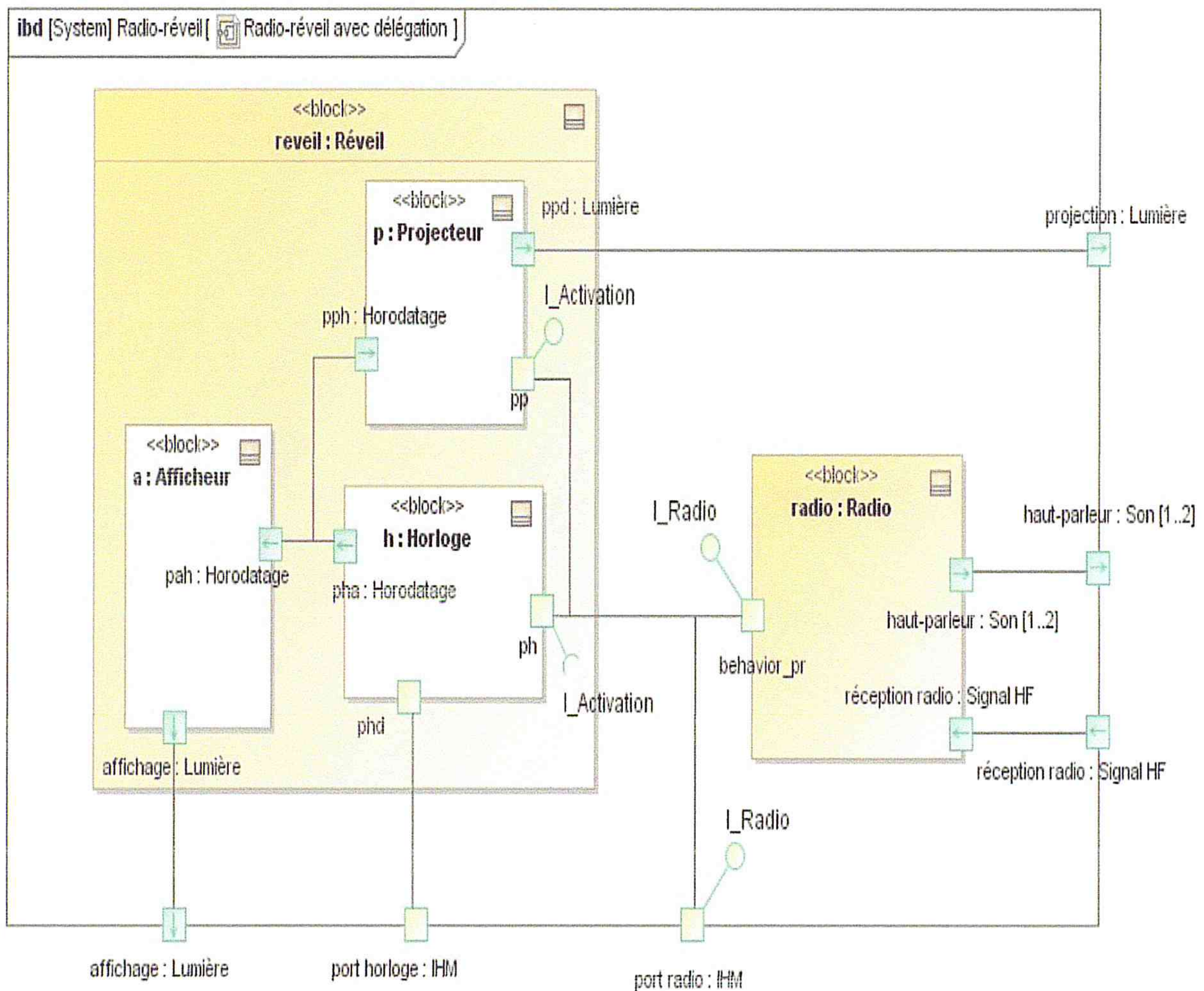


Figure III-9. Exemple de diagramme de bloc interne.

III.3 Le formalisme DEVS

Le formalisme DEVS [12] (Discrete EVent system Specification) a été introduit au milieu des années 1960 par le Professeur Bernard P. Il est de nos jours utilisé par une large communauté de scientifiques, qui l'ont étendu afin de l'adapter à certains domaines spécifiques. Il hérite des concepts de la théorie générale des systèmes, ainsi que des formalismes basés sur les états et les transitions, la gestion du temps via les évènements et la notion d'échéancier, et repose sur une base mathématique inspirée de la théorie des ensembles.

Ce formalisme est un formalisme abstrait, orienté vers la modélisation et la simulation de systèmes à évènements discrets ; il est ensuite implémenté, dans la plupart des cas, au moyen de langages orientés-objet. La popularité de DEVS dans le monde de la recherche académique vient principalement du fait qu'il permet d'appréhender un système de manière comportementale et structurelle : DEVS est modulaire et hiérarchique.

De plus, une propriété importante de DEVS est qu'il fournit automatiquement un simulateur pour chaque modèle : chaque modèle défini et implémenté selon DEVS peut être simulé directement. Il y a donc une séparation explicite entre modélisation et simulation.

Tout système, dont les différents états possibles sont connus et finis, et dans lequel les transitions (conditions de passage d'un état à un autre) sont définies, peut être modélisé en utilisant DEVS. Les transitions peuvent être déclenchées suite à l'expiration d'une horloge interne au modèle, ou bien sous l'action de stimuli extérieurs. Les modèles DEVS peuvent être de deux sortes : modèles atomiques ou modèles couplés.

III.3.1 Modèle atomique

Les modèles atomiques sont les plus petits éléments constitutifs de DEVS. Ils décrivent le comportement, ou une partie du comportement, du système. Ils sont définis comme suit : [12]

$$AM = \langle X, Y, S, t_a, \delta_{int}, \delta_{ext}, \lambda \rangle$$

Où :

- $X = \{(p,v) | p \in InputPorts, v \in Xp\}$ est l'ensemble des entrées, par lequel les évènements externes sont réceptionnés. InputPorts est l'ensemble des ports d'entrée, et Xp est l'ensemble des valeurs possibles pour ces entrées.

Où :

- X = {(p,v)|p∈InputPorts, v∈Xp} est l'ensemble des entrées, par lequel les évènements externes sont réceptionnés. InputPorts est l'ensemble des ports d'entrée, et Xp est l'ensemble des valeurs possibles pour ces entrées.
- Y = {(p,v)|p∈OutputPorts, v∈Yp} est l'ensemble des évènements de sortie, par lequel les évènements externes sont réceptionnés. OutputPorts est l'ensemble des ports de sortie, et Yp est l'ensemble des valeurs possibles pour ces entrées.
- D est l'ensemble des noms des composants, d∈D.
- Md est un modèle DEVS (soit atomique, soit couplé).
- *EIC* est l'ensemble des couplages des entrées externes : ce lien existe entre le port d'entrée du modèle couplé et le port d'entrée de l'un de ses sou-modèles.
- *EOC* est l'ensemble des couplages des sorties externes : ce lien existe entre le port de sortie du modèle couplé et le port de sortie de l'un de ses sou-modèles.
- *IC* est l'ensemble des couplages internes ; un couplage interne est un lien impliquant le port de sortie d'un sous-modèle du modèle couplé, et le port d'entrée d'un autre sous-modèle.
- *select* est la fonction de sélection : elle permet de lever les ambiguïtés dans le cas où, à la même date, plus d'un modèle atomique doit effectuer une transition interne : pour ce faire, elle définit des priorités entre les modèles de D, sous forme de liste ordonnée.

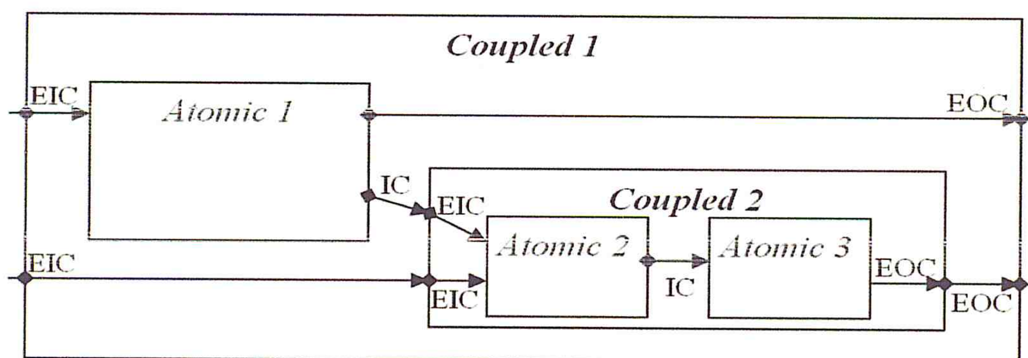


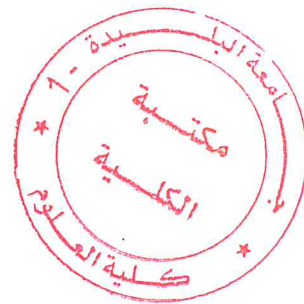
Figure III-11. Exemple de modèle couplé DEVS

III.4 Conclusion

La M&S ne serait rien sans la notion de modèle, abstraction de la réalité d'un ou plusieurs aspects du système. Créer ces modèles suppose un recours à des formalismes de modélisation, dont on a donné un aperçu tout au long de ce chapitre, ces formalismes sont en général de bas niveau d'abstraction. Parmi eux, le formalisme DEVS qui possède de nombreux avantages comme la séparation des modèles et de leurs simulateurs, la modularité, la capacité à décrire simplement des hiérarchies de modèles. Ce formalisme, qui repose sur de solides bases théoriques issues des mathématiques, s'implémente la plupart du temps au moyen de langages orientés objet.

Partie II: Contribution

CHAPITRE IV : Transformation SysML Vers DEVS 51
CHAPITRE V : Implémentation 64



Chapitre IV : Transformation Sysml Vers DEVS

Partie II: Contribution

IV.1 Introduction

La transformation de modèle par mes ces but, ses de donner une autre vue pour la conception de modèle l'application que l'en veut réaliser. Nous avons utilisé et proposer par la suite les metamodele et leur rôle, que ce soit pour les de diagramme du langage SysML que l'en veut prendre deux diagramme de ce langage de Système en lui propose la relation entre ce modèle et le formalisme DEVS et le mapping déjà proposer par en ces deux langage de modélisation

IV.2 La Conception

C'est une étape très importante pour la préparation de la mise en œuvre de n'importe quel projet de modélisation. Lors de ce chapitre, nous allons utiliser langage SysML et le formalisme DEVS comme source et cible de la transformation. Nous allons présenter une description de l'architecture de notre système en détail avec les règles de transformation qu'on a utilisées.

IV.3 Le méta-modèle SysML

IV.3.1 La structure du Métamodèle du diagramme de block SysML

Un méta-modèle partiel pour le diagramme de définition de bloc est illustré dans la figure ci-dessous [16] :

Ce métamodèle partiel représente un bloc qui appartient aux diagrammes de définition de bloc qui sont constitués de deux éléments de base: les blocs et les relations. Les blocs et les relations peuvent avoir différents types.

Un diagramme de définition de bloc peut également contenir différents types de ports et d'interfaces, ainsi que des flux d'éléments.. Les blocs décrivent les types qui existent dans un système, comme des propriétés ou des opérations ou fonctionnalité, tandis que les relations décrivent les relations entre les différents blocs. Un «diagramme de définition de bloc» est composé de un ou plusieurs «bloc», zéro ou plus «relation», zéro ou plus «port», zéro ou plus "Flux d'éléments" et zéro ou plusieurs "Spécifications d'interface». Chaque 'Relation' relie deux 'Blocs'.

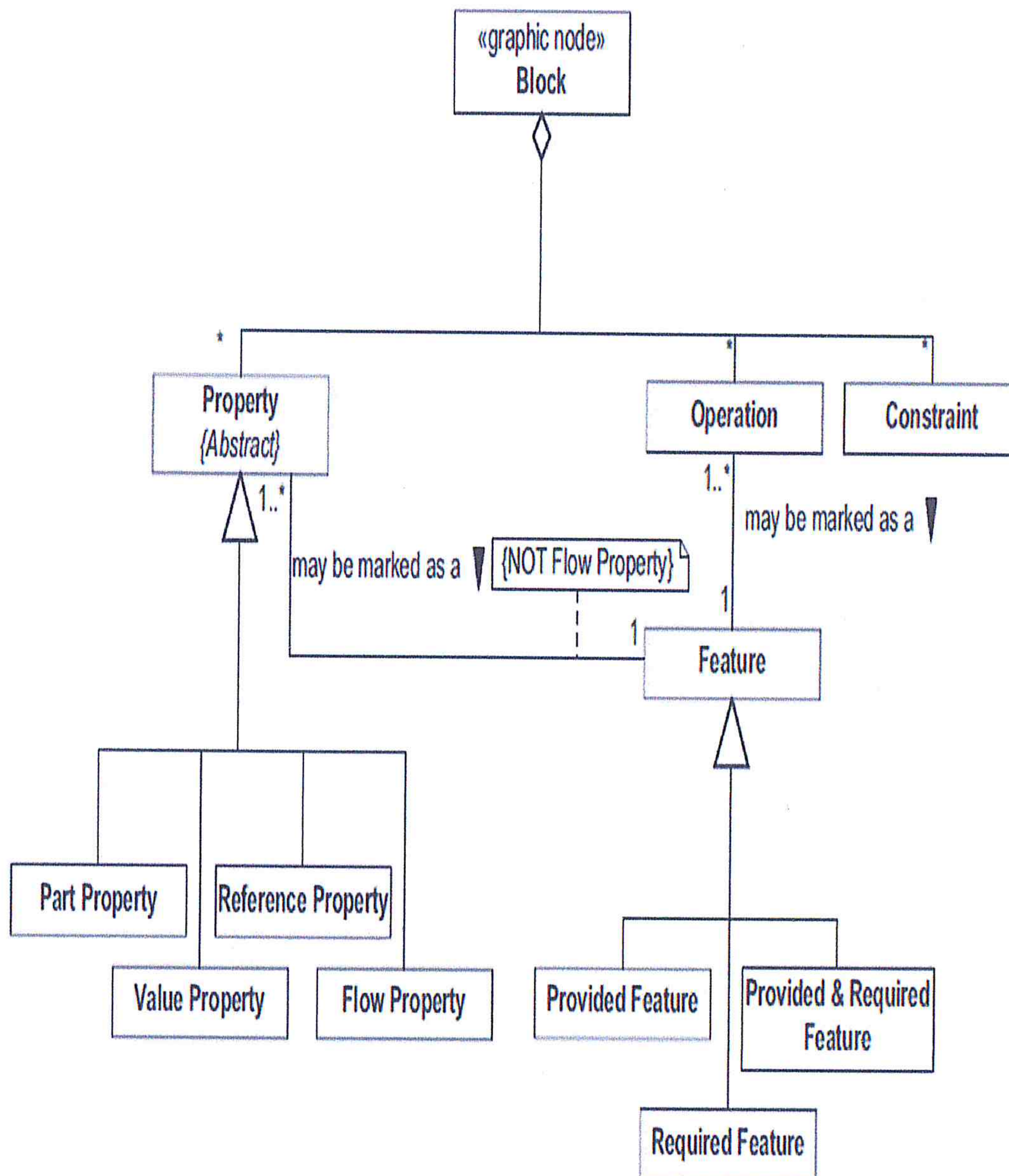


Figure IV-1 : Méta-modèle partiel montrant le bloc de BDD [16]

Chaque 'Bloc' est constitué de zéro ou plus de 'Propriété', zéro ou plus 'Opération' et zéro ou plus 'Contrainte' comme montre la Figure ci-dessous.

Parmi les type de propriété utilisées on a 'Flow Property', qui est utilisé lorsqu'on a un «flux d'éléments», on a aussi 'Part Property', qui appartient au bloc. Autrement dit, une propriété

intrinsèque au bloc mais qui aura sa propre identité ainsi une propriété peut appaartenir entièrement à un bloc parent ou qui peut être partagé entre plusieurs blocs parents.

'Reference Property', qui est référencé par un Bloc, mais ne lui appartient pas, et la dernière c'est 'Value Property', qui représente une propriété qui ne peut être identifiée que par la valeur elle-même, par exemple des nombres ou des couleurs.

Une caractéristique 'est une propriété ou une opération qu'un bloc prend en charge pour que d'autres blocs puissent être utilisés une «Provided Feature» caractéristique fournie ou dont il a besoin d'autres blocs à prendre en charge pour son propre usage une *fonctionnalité requise* «Required Feature»), ou les deux a «Provide & Required Feature ».

IV.3.2 Méta-modèle de base DEVS :

Le formalisme DEVS (Discrete Event System Specification) fournit un cadre conceptuel pour la spécification de modèles. Deux types de modèles sont définis dans DEVS: les modèles atomiques (représentation comportementale), à partir desquels les plus grands sont construits et décrivent la fonctionnalité de base du modèle, et les modèles couplés (représentation structurelle) exprimant comment les modèles de base sont hiérarchisés. La définition formelle du formalisme DEVS peut être trouvée dans Les modèles couplés sont constitués d'autres modèles DEVS (couplés ou atomiques). Les modèles communiquent via des ports d'entrée et de sortie correctement interconnectés. La figure 1 représente un exemple simple d'un modèle couplé DEVS. Le modèle SERVICE est défini comme le couplage de deux modèles atomiques: QUEUE et TELLER. Le couplage est décrit par la définition de la correspondance entre les ports d'entrée et de sortie des composants et le modèle couplé.

Les modèles atomiques sont décrits par les états de modèles correspondants et les transitions entre eux. Quatre fonctions sont utilisées pour décrire leur comportement. La fonction de transition interne spécifie l'état suivant vers lequel le système va transiter.

La fonction de transition externe spécifie l'état du système suivant lorsqu'une entrée est reçue (l'état suivant est calculé sur la base de l'état actuel, le temps écoulé et le contenu de l'événement d'entrée externe).

La fonction de sortie génère une sortie externe juste avant une transition interne.

La fonction d'avance temporelle contrôle le timing des transitions internes. Sur la figure 1, le modèle atomique TELLER attend un client dans l'état "inactif".

Lors de la réception d'un client dans le port d'entrée "Customer in", le TELLER change son statut dans l'état "occupé" et augmente la variable "client". Le temps de service est une variable aléatoire distribuée exponentiellement. Lorsque le service est terminé, il envoie le client au port de sortie "client out" et "prêt" message à la QUEUE informant son état [17].

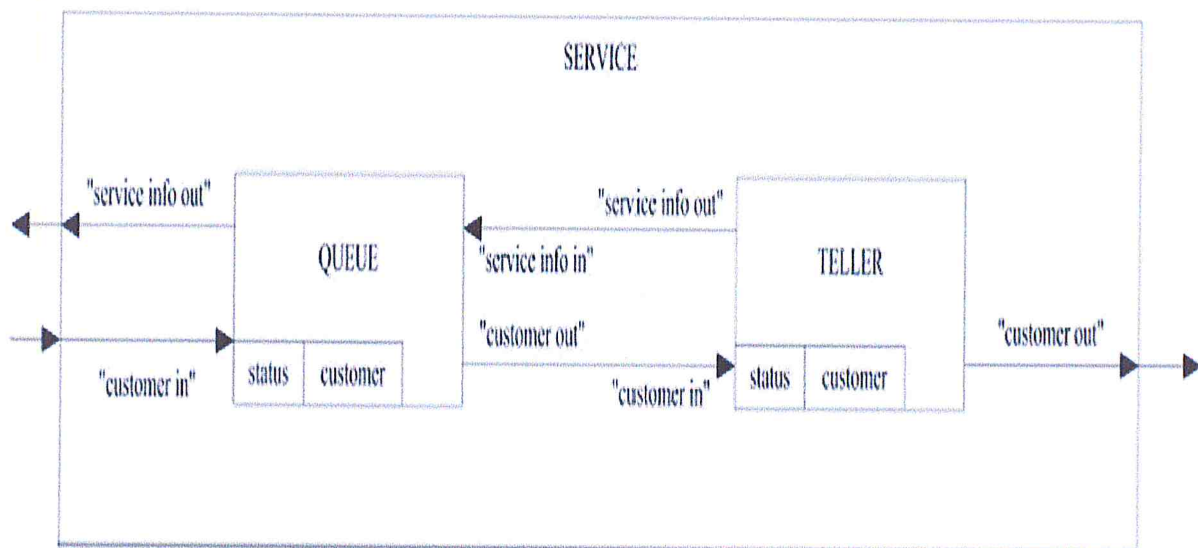


Figure IV-2 : Paradigme du modèle DEVS.

On va parler sur le métamodèle de DEVS dans la figure suivant et la hiérarchie qui représente les déférentes modèles DEVS, d'après le métamodèle basic suivant on a :

Concernant la métaclasse Message relier par la métaclasse DEVS Modèle, c'est le message qui circule entre les modèle DEVS, soit le modèle couplé ou bien le modèle atomique, après un traitement de ce message, ce dernier va diffuser sous forme d'information entre les déférente modèle. La métaclasse DEVS Modèle représente les modèle DEVS, on a déjà parlé sur ces modèles dans le chapitre précèdent

IV.3.2.1 Modèle couplé DEVS

C'est le Modèle couplé représenté par les deux métaclasses « coupled model » et « coupling definition » qui met la relation entre les déférente modèle « coupled » avec le port origine et le port destination, aussi qui est basé sur la constitution de modèles (atomiques ou couplés), leurs interconnexions à travers des points de connexion, appelés ports, et la capacité de composition [23], [24].

IV.3.2.2 DEVS Atomique modèle

La définition d'un modèle atomique DEVS comporte deux étapes: la description du modèle et la définition de caractéristiques statiques, telles que les états et les ports d'entrée et de sortie.

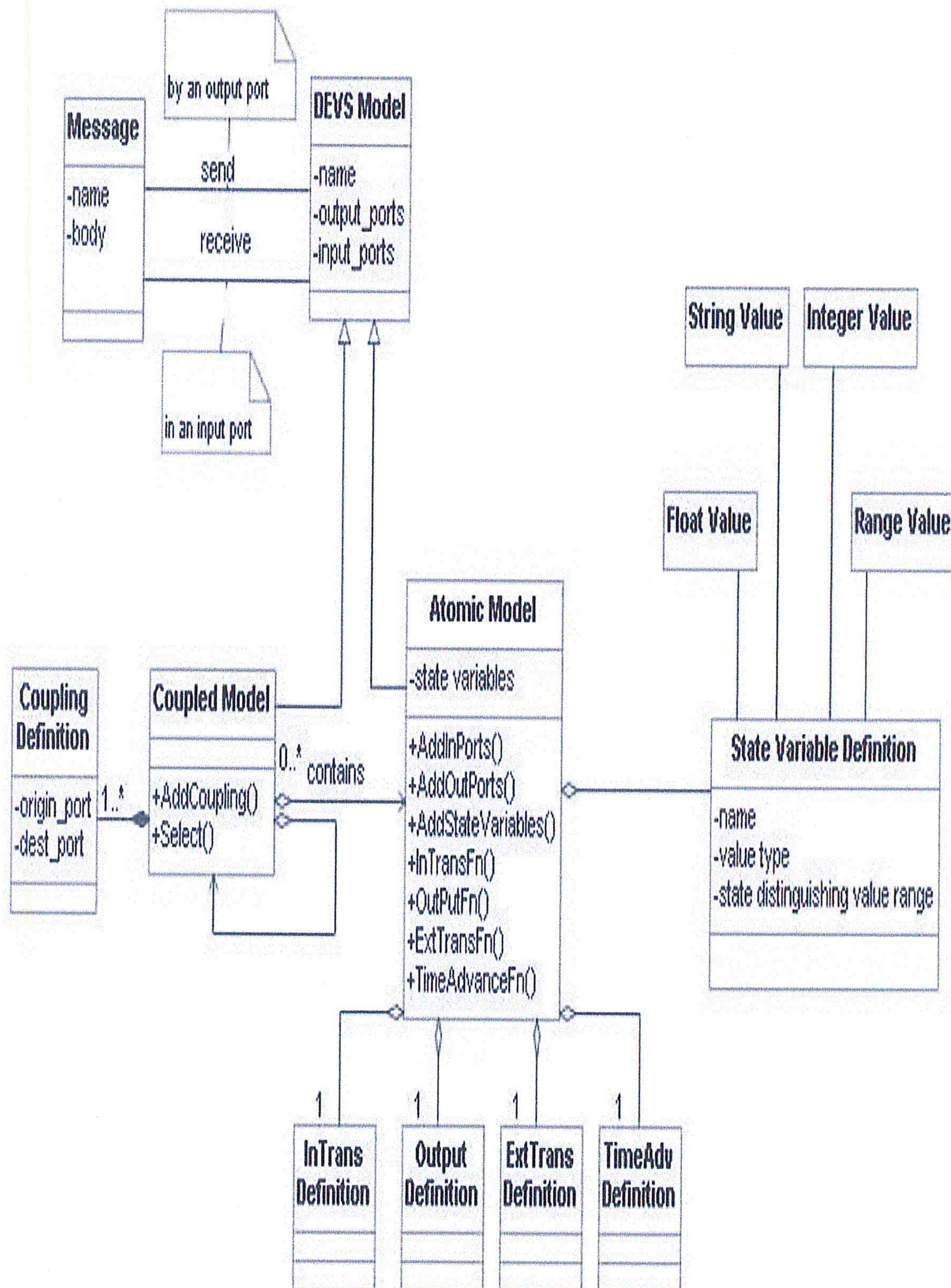


Figure IV-3 : basic métamodèle DEVS

IV.4. La création des Métamodèle

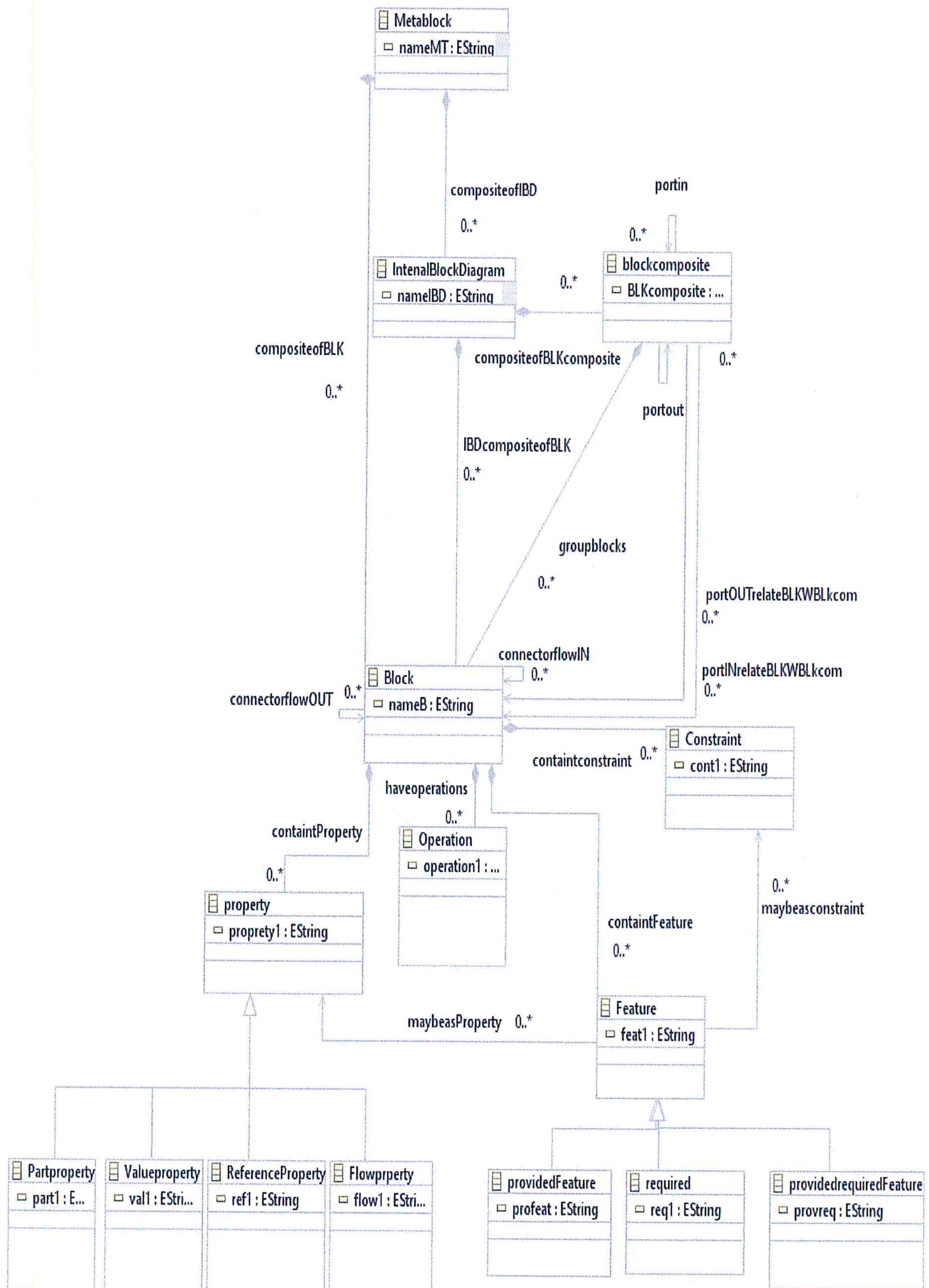
Selon MOF, un métamodèle définit la structure que doit avoir tout modèle conforme à ce métamodèle. Autrement dit, tout modèle doit respecter la structure définie par son métamodèle. Par exemple notre le métamodèle SysML définit que les modèles (diagramme de définition bloc et diagramme de bloc interne) contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc. Les métamodèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. MOF les représente sous forme de diagrammes de classes.

Rappelons que les diagrammes de classes permettent de représenter les notions d'un domaine et leurs propriétés, que ces notions ou entités soient organisées ou non sous forme d'objets. Cette utilisation des diagrammes de classes a le double avantage de permettre de définir très précisément les métamodèles et de les rendre eux aussi pérennes et productifs. Un métamodèle est donc une sorte de diagramme de classes qui définit la structure d'un ensemble de modèles [26]. La section suivante de ce chapitre donne notre métamodéle.

IV.5. Les métamodèles proposés

IV.5.1 Métamodele SysML des modèles (bloc et bloc interne)

On a proposé se métamodèle après une étude approfondie des deux modèle SysML et avec la partie de métamodèle, cette partie représenté un seul modèle seulement, c'est le diagramme de bloc comme on a vu dans le précédemment cette partie de métamodèle.



Metablock
nameMT : EString
compositeofIBD
0..*
portin
0..*
IntenalBlockDiagram
nameIBD : EString
0..*
blockcomposite
BLKcomposite : ...
compositeofBLK
0..*
compositeofBLKcomposite
0..*
portout
IBDcompositeofBLK
0..*
groupblocks
0..*
portOUTrelateBLKWBLkcom
0..*
connectorflowIN
0..*
portINrelateBLKWBLkcom
0..*
Block
nameB : EString
connectorflowOUT
0..*
Constraint
cont1 : EString
containtconstraint
0..*
haveoperations
0..*
Operation
operation1 : ...
containtProperty
0..*
property
proprety1 : EString
containtFeature
0..*
0..*
maybeasconstraint
maybeasProperty
0..*
Feature
feat1 : EString
Partproperty
part1 : E...
Valueproperty
val1 : EStri...
ReferenceProperty
ref1 : EString
Flowprperty
flow1 : EStri...
providedFeature
profeat : EString
required
req1 : EString
providedrequiredFeature
provreq : EString

Figure IV-4 : Métamodèle diagramme de (bloc et bloc interne).

On a vu que notre métamodéle SysML est constitué d'une metaclasse qui s'appelle Metablock, cette metaclasse est qui représente le diagramme de définition de bloc, qui est composer d'un autre diagramme qui s'appelle diagramme de bloc interne, donc pour le partiel métamodèle représente juste tout modèle de bloc, avec cette métaclasse Metabloc en peut construit n'importe quelle diagramme de bloc, et la deuxième metaclasse InternalBlockDiagram est composer des bloc et des bloc composer représente les caractéristique de ce bloc composer ,cela représenter par la metaclasse Blockcomposite, ces metaclasse nous permet de représenter n'importe quelle diagramme de bloc interne, et les autre metaclasse on a déjà expliqué précédemment.

IV.5.2 Métamodèle DEVS

C'est un métamodèle basé sur le basic métamodèle DEVS comme vu dans la description de métamodèle basic DEVS, il porte le même comportement expliqué déjà sauf qu'on crée trois métaclasse au lieu de une seule qui s'appelle « State variable definition » car cette métaclasse et remplacé dans le SysML les propriété et les contraints et les caractéristiques quelle que soit, les métaclasse mère ou les sous métaclasse, et des relations suivant les besoin de notre transformation voilà notre métamodèle proposer :

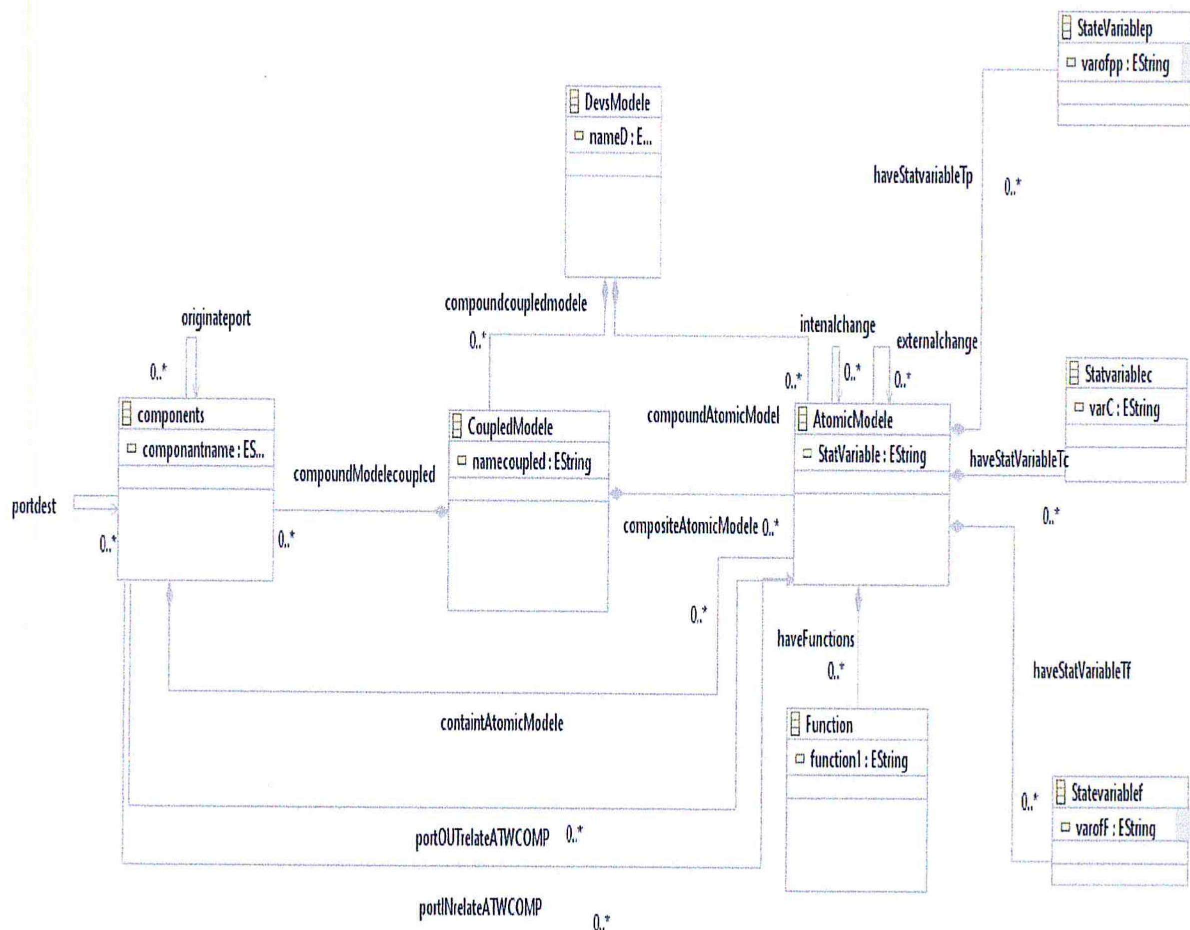


Figure IV-5 : Métamodèle DEVS.

On déjà expliqué le métamodèle DEVS, la seule différence est que ceci est plus générale car on a ajouté la metaclasse components qui nous permet de créer un modèle couplé dans un autre modèle couplé qui contient aussi des blocs atomique par la relation containtAtomicModéle , anssi la relation portOUTrelateATWCOMP and portINrelateATWCOMP ,c'est relation représente la relation entre les modèle couplé ou les modèle atomique.

IV.6. Les règles de transformation du diagramme de block SysML vers DEVS-XML

On a commencé cette transformation M2M en se basant sur le Table de mapping qu'on a défini. Le tableau suivant montre les Stéréotypes structuraux du modèle entre le formalisme

DEVS et les entités SysML, à partir de tableau de similarité on a commencé notre transformation par lister les ressemblances entre les différentes entités :

DEVS modèle	Diagramme de définition de bloc	Il y a un BDD contenant le modèle global. Seules les entités DEVS CM et DEVS AM participent à ce diagramme
Diagramme interne couplé DEVS	Diagramme de bloc interne	Un diagramme interne couplé DEVS doit être associé à un CM DEVS. Le diagramme peut uniquement contenir les modèles DEVS directement utilisés par le CM DEVS correspondant. Tous les ports de flux doivent être connectés de manière appropriée (sortie vers le port de flux d'entrée).
DEVS CM	Bloc	Un CM DEVS peut uniquement avoir des parties de propriété et des ports de flux unidirectionnels (entrants ou sortants). Chaque CM DEVS est associé à un diagramme interne couplé DEVS.
DEVS AM	Bloc	Un DEVS AM ne peut avoir que des ports de flux unidirectionnels (entrants ou sortants), des propriétés de valeur et des contraintes sur les propriétés de valeur. Quatre sous-diagrammes doivent être associés à chaque DEVS AM pour décrire son comportement: DEVS States Definition, DEVS States Association, DEVS Atomic Internal et DEVS Atomic External.

Tableau IV-1 : tableau représente la similarité entre l'entité SysML et les stéréotypes DEVS.

Donc le tableau suivant va nous détailler plus le 'mapping' entre le formalisme DEVS et l'entité SysML, on a déjà parlé sur la plus part de ces entités sauf 'Internal coupling' qui représente dans le modèle SysML le connecteur entre l'élément 'part' de diagramme de bloc interne et le 'External coupling' représente le connecteur entre les éléments de blocs composé et les blocs [18].

DEVS Formalisme	Entité SysML
Modèle atomique	Bloc
Port d'entrée pour les événements	Port de flux avec flux d'éléments
Port de sortie pour les événements	Port de flux avec flux d'éléments
Variables d'état	Propriétés de valeur et contraintes
Paramètres	Value properties
Fonctions du modèle atomique DEVS	Diagrammes de comportement
modèle couplé	Diagramme bloc interne
Composants	Blocs / Partie
Couplage interne	Connecteurs entre les ports de débit des parties d'IBD
Couplage externe	Connecteurs entre les orifices d'écoulement du bloc d'enceinte de l'EIA et ses parties

Tableau IV-2 : Le tableau de mapping entre SysML et modèle DEVS.

Cette transformation permet aux modélisateurs de systèmes d'intégrer les informations spécifiques à la simulation dans le modèle DEVS. Par conséquent, un modèle DEVS équivalent, peut toujours être construit pour chaque modèle SysML valide. Bien que le modèle SysML équivalent théorique, puisse présenter des relations pour le modèle DEVS, l'expression d'un modèle SysML dans DEVS-XML est cruciale, car cette dernière va être automatiquement transformée en code pour une série d'environnements de simulation DEVS spécifiques.

Les règle de transformation proposées son construit à cause de la similarité bien définie entre les entités de modèle SysML (diagramme de block) et le formalisme DEVS.

On basé sur six règle de base pour cette transformation :

- Un bloc SysML devient le modèle atomique.

- Les opérations dans un bloc sont transformées en DEVS en fonctions du modèle atomique.
- Internal bloc diagramme est devenu dans le modèle DEVS Coupled Model.
- Les connecteurs dans le modèle SysML remplacent par Internal coupling et External coupling dans le modèle DEVS pour la communication et le transfert des informations.
- Stat variable prend le rôle des propriétés et les fonctionnalités et les contraints.

IV.7. Conclusion

Dans ce chapitre on a présenté notre proposition. Les métamodèles ont été adaptés proposé selon nos besoins. La table de mapping va être traduite en règles ATL. Nous présentons les détails d'implémentation dans le prochain chapitre.

Chapitre V : Implémentation

Partie II: Contribution

V.1. Introduction

Comme déjà vu le mapping fait entre le SysMl et DEVS, en se basent sur ce mapping on a déterminé la relation entre le SysML avec ces entité et le formalisme DEVS nous passons à l'étape d'implémentation ou on a mis en place l'architecture du système et la solution de générer la transformation d'un modèle vers un autre. Donc dans ce chapitre on va présenter trois parties. La première sert à présenter l'environnement de développement et les outils utilisés. La deuxième partie présente des aperçus réels de notre application et la troisième partie est consacré à l'évaluation et la discussion des résultats obtenus et aussi la complexité de notre exécution.

V.2. Environnement du développement

V.2.1 Eclipse

Eclipse est un environnement de développement libre, extensible et polyvalent, initié par IBM en 2001.il permet le développement d'applications Java principalement, mais également d'autres langages grâce à l'utilisation de plugins. L'objet de la solution Eclipse est de fournir des outils favorisant la productivité, mais pas seulement celle qui concerne le codage logiciel. On y trouve des environnements de développement intégré mais également de conception, de modélisation, de tests, de reporting, etc.

La version utilisée dans notre développement est Oxygen .3 March 2018 [31].

Les produits phares sont :

- Eclipse Classic : plateforme Eclipse, outils de développement Java, environnement pour le développement de plugins.

- IDE Eclipse pour développeurs Java comprenant les outils essentiels pour tout développeur Java : IDE, client CVS, éditeur XML, intégration Mylyn, Maven, WindowBuilder…

- IDE Eclipse pour développeurs Java EE (anciennement J2EE) : outils pour les développeurs Java créant des applications Java EE et Web. Inclut une IDE Java, des outils pour Java EE, JPA, JSF, Mylyn & autres.

V.2.2 Le plugin Graphical Modeling Framework

Le cadre de modélisation graphique Eclipse (GMF) fournit une composante générative et l'infrastructure d'exécution pour le développement éditeurs graphiques basés sur EMF et GEF. "Comme il est spécifié, GMF se compose de deux parties - générative et runtime. L'exécution partie pourrait être décrite comme un ensemble de plug-ins prolongeant existant Fonctionnalités EMF [5] et GEF [6]. L'exécution ne permet pas seulement faciliter l'intégration entre la FEM et le FEM, mais fournit des services tels que: le support des transactions, la méta-modélisation étendue installations, méta-modèle de notation, points de variabilité utilisés pour l'extensibilité à l'exécution du code généré, etc [27]

V.2.3 GMF métamodèle

Comme il a été mentionné, GMF définit un certain nombre de métamodèles spécifiques. Certains de ces méta-modèles seront instanciés lors de l'exécution et seront utilisés par l'éditeur de diagramme généré.

V.2.4 Modèle de définition graphique

Le modèle de définition graphique est instancié pour rassembler des informations sur un futur éditeur de diagramme à partir d'un outilleur et pour alimenter les paramètres correspondants dans le sous-système du générateur de code. Ce modèle n'est pas disponible dans l'environnement d'exécution, car ces informations ne sont utilisées que pour la génération de code lors de la création de l'éditeur de diagramme. Le GMF contient un certain nombre de méta-modèles décrivant différents aspects des futurs éditeurs de diagrammes [27].

V.2.5 Éditeur de diagramme de modèle Ecore

Reflective Ecore Model Diagram Editor est un plugin Eclipse basé sur GMF qui fournit un éditeur graphique pour tout fichier de modèle EMF, en utilisant uniquement le méta-modèle [33].

V.3. La transformation avec Atlas Transformation langage

Un moyenne de Spécifier la manière de produire un certain nombre de modèles cibles à partir de modèles sources.

ATL s'utilise dans le contexte de transformation présenté dans la figure suivante :

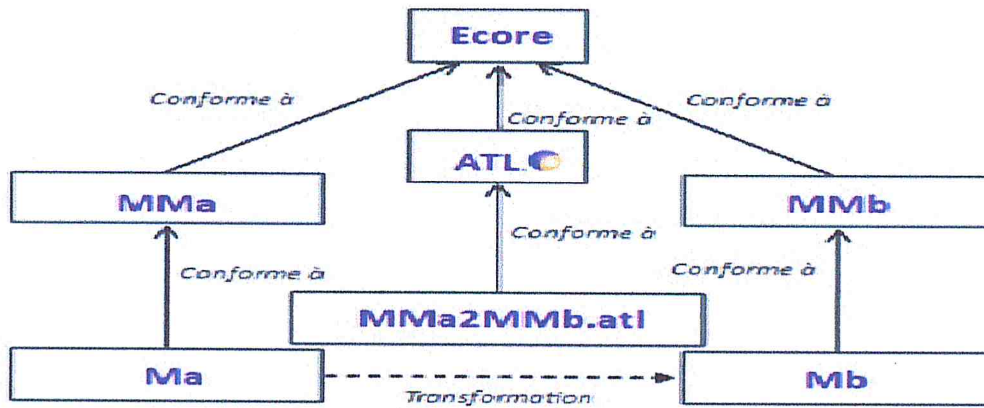


Figure V-1. Schéma de transformation dans ecore.

Dans la figure, un modèle source Ma qui est représenté par un diagramme de bloc ou un diagramme de block interne est transformé en un modèle Mb est un modèle qui représente un diagramme DEVS.

La transformation est dirigée par un programme de transformation mma2mmb.atl écrit en ATL. Les modèles source et cible ainsi que le programme de transformation sont conformes à leurs métamodèles respectifs : MMa est le métamodèles de diagramme de bloc et de diagramme de bloc interne et MMb est le métamodèles de diagramme DEVS. Ces métamodèles sont conformes au méta-formalisme Ecore de l'environnement EMF, puis nous montrons comment on peut les transformer vers DEVS [32].

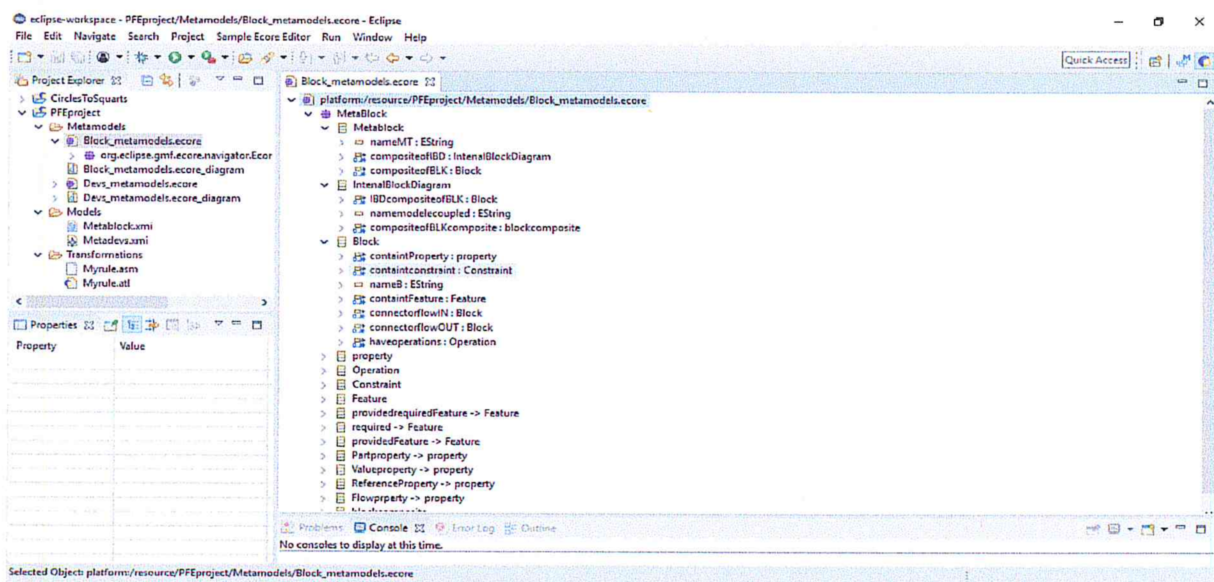


Figure V-2. Méta-formalisme Ecore

V.4. Échange de métadonnées XML

Les modèles cibles et source prennent dans l'environnement de développement Eclipse la forme de fichiers texte au format xmi avec une extension .ecore .

XMI permet de représenter n'importe quel modèle sous forme de document XML .Le principe de fonctionnement de XMI consiste à générer automatiquement une spécification de structuration de balises XML, à partir des informations d'un modèle source qui est généraliser par leur metamodele la ou on génère notre transformation et d'avoir notre simulation de modèle cible [32].

V.5. Présentation de l'application

On a commencé notre application par la création du metamodèle source qui comporte une généralisation de deux diagrammes. Le diagramme de bloc et le diagramme de bloc interne. La cible est le metamodèle du diagramme DEVS qui a déjà été définie dans le chapitre de transformation.

La partie test a été faite avec une architecture concrête issue d'une étude de cas « Gestion de Foule », Nous nous sommes limités à l'architecture relative au diagramme de bloc « Assessing Risks» pou. En appliquant les règles de transformation, une description du système en DEVS va être générée sous forme de fichier XML.

On commence la démonstration par la création d'une instance dynamique 'Create Dynamic instance 'à partir de la Meta-classe 'Metablock' qui représente le métamodèle de la source suivant de notre modèle source.

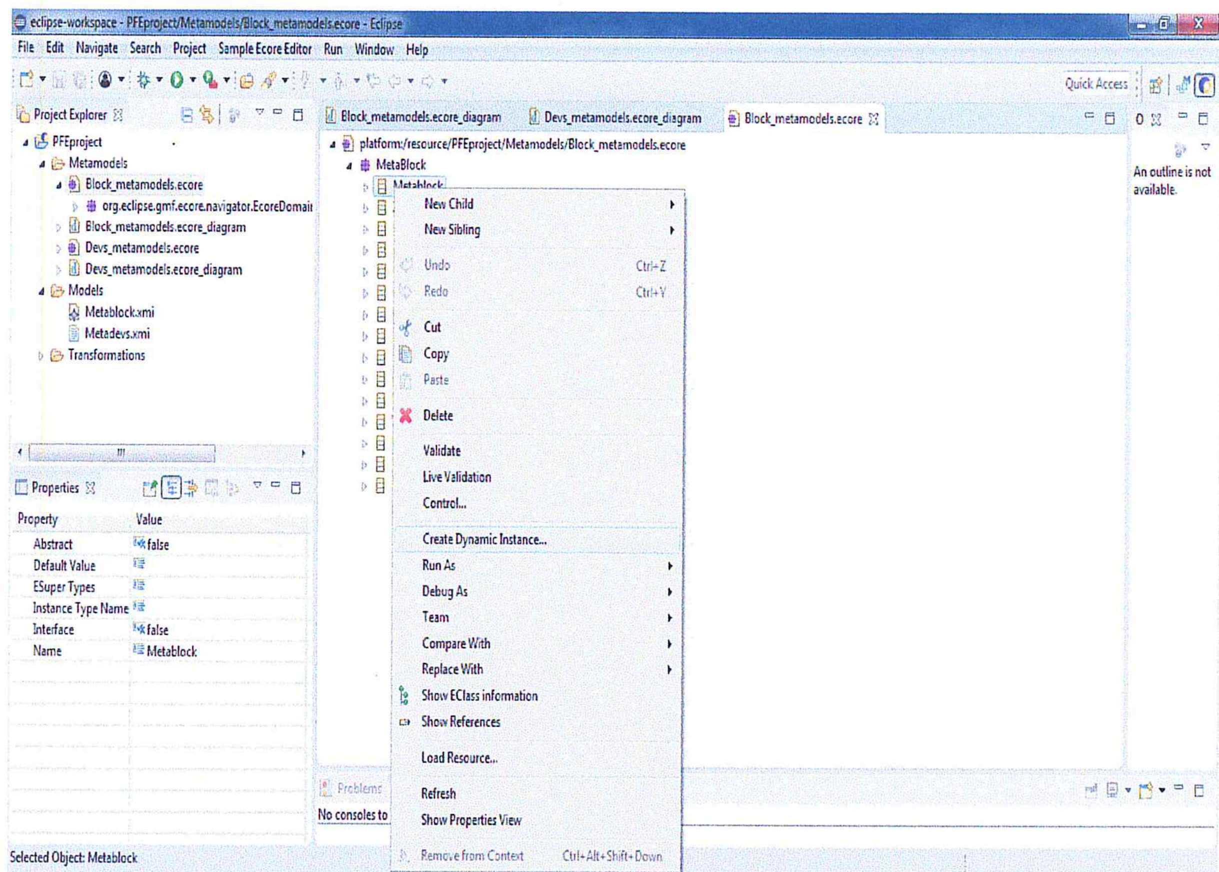


Figure V-3. Création d'une instance dynamique.

V.5.1 Instance de créations les modèles

A partir de cette instance montrée dans la figure ci-dessus, on peut créer n'importe quel modèle de diagramme de bloc ou bien un diagramme de bloc interne.

L'environnement va nous générer un fichier XML qui porte des caractéristiques de notre modèle sue l'on veut transformer vers un modèle DEVS.

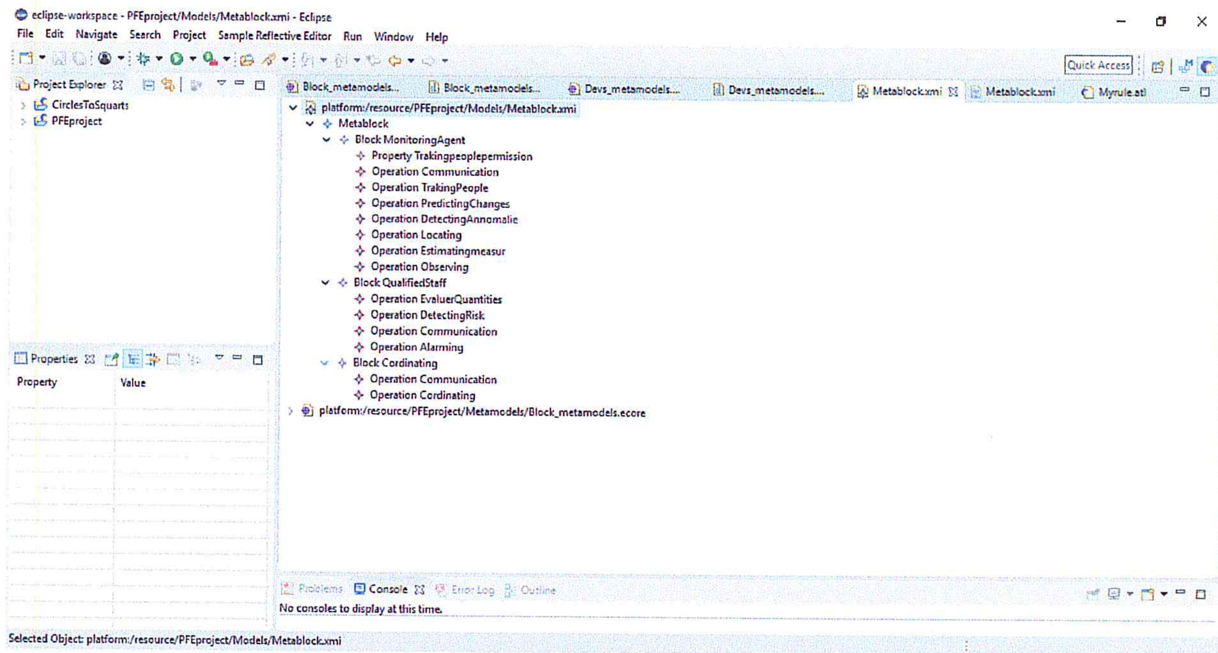


Figure V-4. Modèle SysML a transformé en DEVS.

V.5.2 Modèle de test et vérification de la transformation

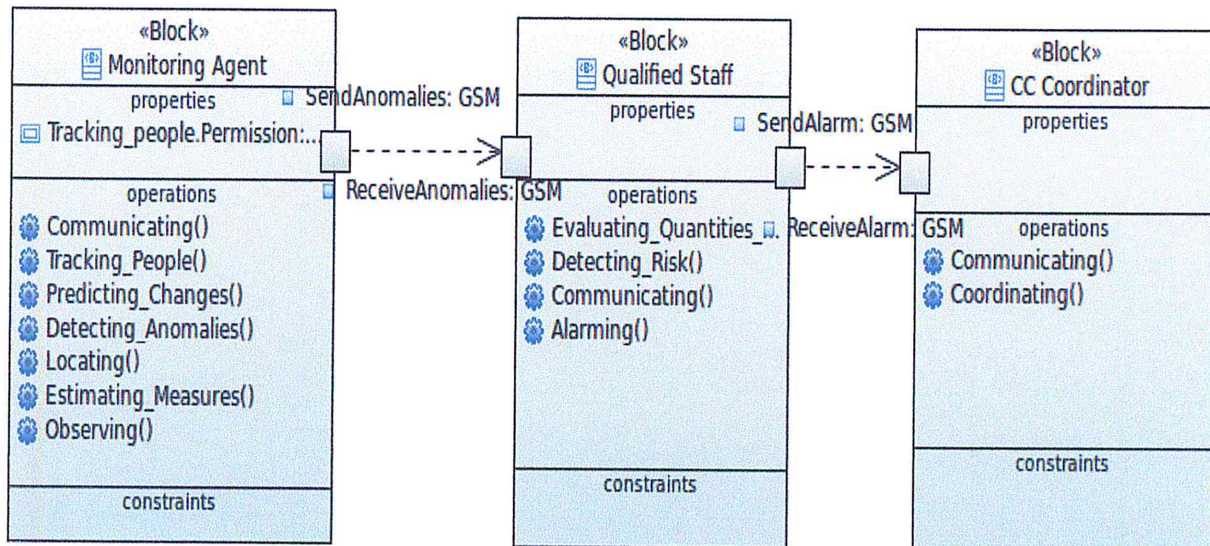


Figure V-5. Assessing Risks Concrete Architecture Modeling.

Voilà notre exemple à transformer, c'est un diagramme de bloc composé de trois blocs qui portent des noms du bloc et des propriétés, aussi des différentes opérations,

Tout bloc est composé des ports qui fond la communication entre les blocs on a modélisé le port par une association avec le bloc lui-même, lors de la l'instanciation des blocs l'environnement va nous créer une relation permet de relier le bloc.

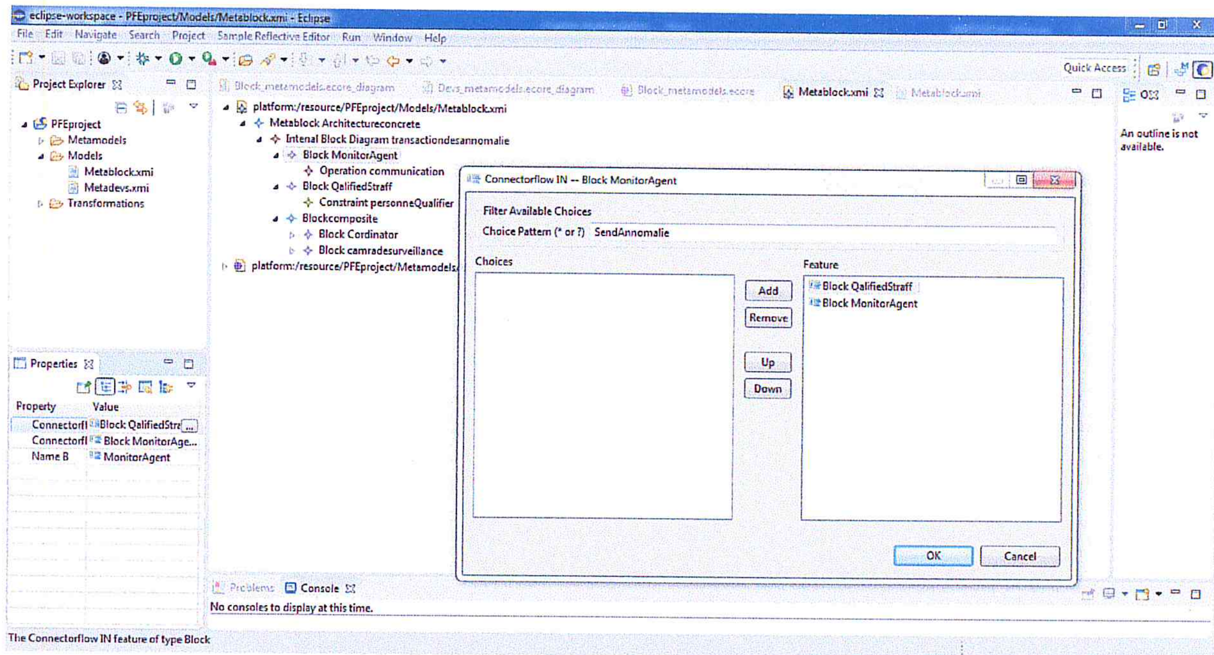


Figure V-6. Représentation de port de communication entre les blocs.

V.5.3 Résultat de l'application

Après avoir implémenter le modèle et créer la transformation à partir des règles de transformation comme suivant :

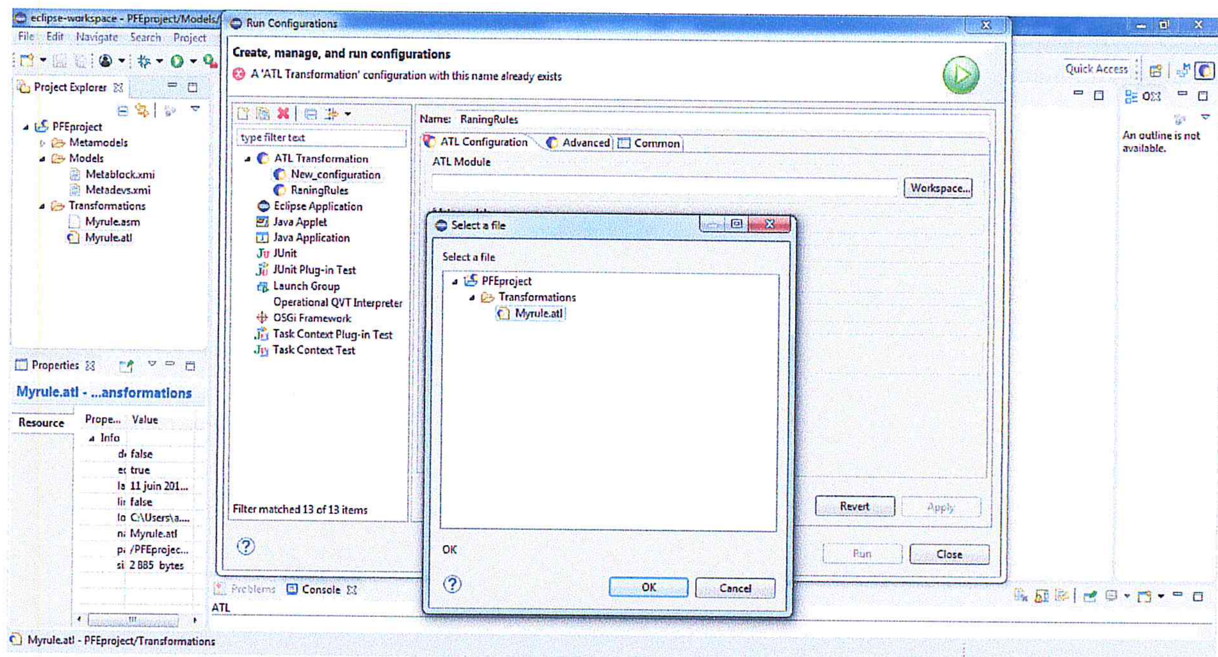


Figure V-7. Choisir les règles de transformation.

Apres l'implémentation du modèle source on va créer notre transformation on utilise les règles de transformation développées par le langage ATL, pour cette transformation on a utilisé les règles de transformation suivante :

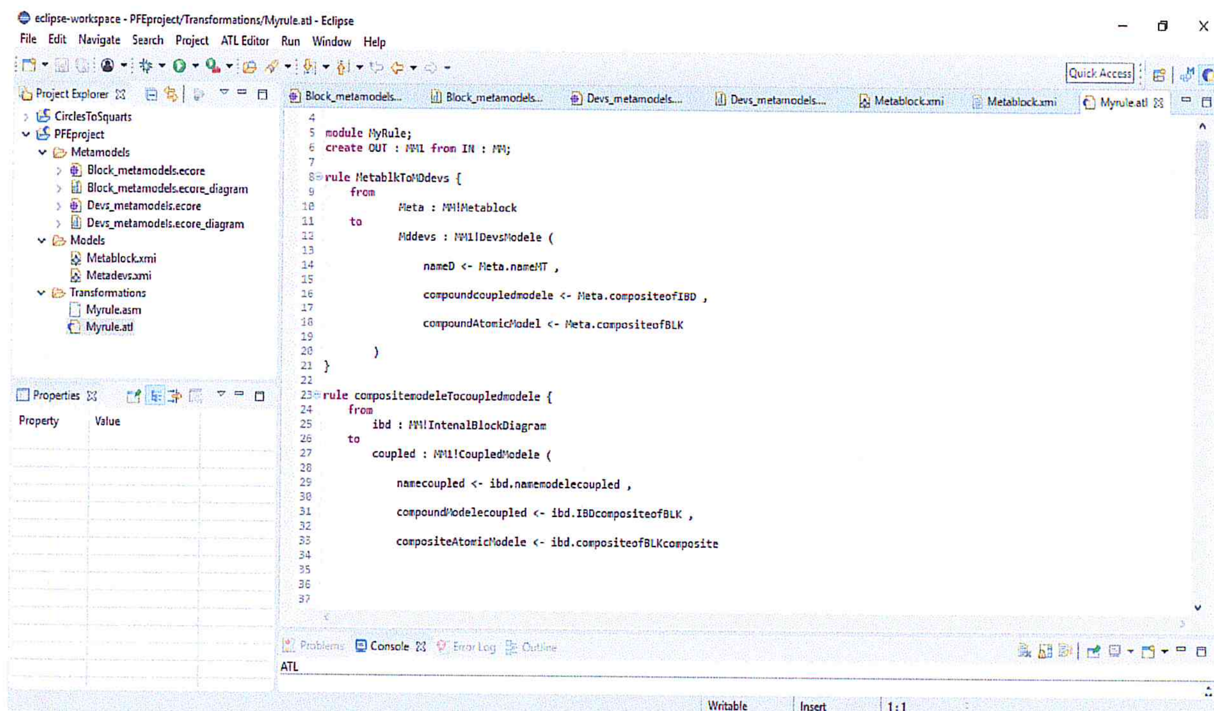


Figure V-8. Les règles de transformations.

On a détaillé les règles de transformation de ce langage dans le chapitre 4 et on a spécifié quel type de règle on a utilisé. Le résultat de notre transformation est un modèle de simulation en DEVS sous la forme XML.

compl

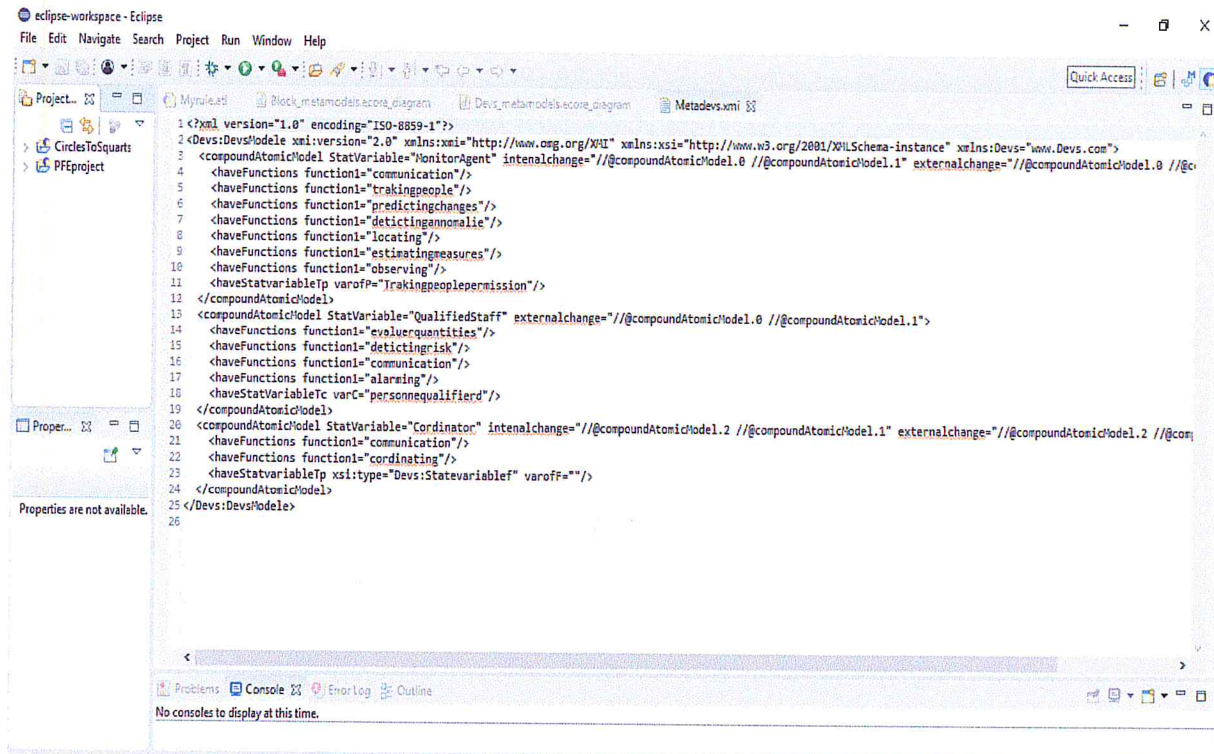


Figure V-9. Modèle de simulation en DEVS.

V.6. Evaluation de la complexité

Notre code source sous forme d'un traducteur est un procède automatique pour résoudre un problème de transformé un modèle vers un autre en un nombre fini d''étapes.

Donc la complexité de code sera de temps constant : O (1).

V.7. Conclusion

Dans ce chapitre, on a implémenté la solution de notre problème par la réalisation d'une transformation de modèle vers un langage permettant la simulation. On a présenté des aperçus sur cette dernière puis on a terminé par une validation du résultat.

Conclusion générale

Conclusion générale

Nous avons introduit une approche qui permet de transformer automatiquement un modèle vers un autre modèle qui est la transformation M2M. Pour cela nous avons utilisé deux métamodèles, le premier représente le digramme de définition de bloc et le diagramme de bloc interne qui son partie au langage source SysML, le second représente le formalisme DEVS comme un langage cible.

Pour attendre l'objectif de notre travail nous avons décomposé le travail sur deux parties essentielles :

- Nous avons débuté la première partie par une présentation du monde de la modélisation et simulation, de ses concepts fondamentaux, et de l'un des formalismes de modélisation les plus populaires : Discrete EVent system Specification (DEVS). Nous avons ensuite présenté une discipline du génie logiciel dont l'importance est grandissante : L'Ingénierie Dirigée par les Modèles (IDM).

- La seconde partie de ce document a montré notre contribution, articulées autour des méta-modèles source et cible et des règles de transformation associées.

Nous avons implémenté les règles de transformation avec ATL Nous avons utilisé une architecture abstraite d'une étude de cas « gestion de foule » pour tester notre transformation.

Bibliographie

[1] Stéphane Garredu, "Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS," 2014.

[2] M. S. Miloud *et al.*, "Vérification et validation de transformation de modèles," 2015.

[3] S. Turki, "Ingénierie système guidée par les modèles : Application du standard IEEE 15288, de l'architecture MDA et du langage SysML à la conception des systèmes mécatroniques.," *Lismma*, vol. PhD thesis, 2008.

[4] S. Diaw, R. Lbath, and B. Coulette, "Etat de l'art sur le développement logiciel basé sur les transformations de modèles," *Numéro spécial TSI - Ingénierie Dirigée par les Modèles*, vol. 29:4-5, no. 4–5, p. 2, 2010.

[5] L. Jean Zay -Thiers, "CPGE PT -S2I Les diagrammes SysML Cours," pp. 1–9,2016

[6] "atlpres." 'The ATLAS Transformation Language ' 2018

[7] R. Benabidallah, I. Cherfa, S. Sadou, and M. A. Nacer, "Situation / Reaction Paradigm for SoS Simulation," conférence SOSE 2018.

[8] M. Jamshidi, *SYSTEMS OF SYSTEMS ENGINEERING Principles and Applications*. 2010.

[9] S. Mittal, B. P. Zeigler, J. L. Risco-Martín, F. Sahin, and M. Jamshidi, "Modeling and Simulation for Systems of Systems Engineering," *Syst. Syst. Eng.*, pp. 101–149, 2008.

[10] J. R. Ruault and J. P. Meinadier, *Standardization in the Field of Systems and Systems of Systems Engineering*. 2013.

[11] D. Fagnon and S. Gaston, "SysML : les diagrammes," *Technol. n°179*, pp. 100–105, 2012.

[12] P. Roques, *SysML par l'exemple - Un langage de modélisation pour systèmes complexes*. 2009.

[13] F. Fleurey, “Langage et méthode pour une ingénierie des modèles fiable Franck Fleurey To cite this version : HAL Id : tel-00538288,” 2010.

[14] S. C. Et, T. E. D. E. L. I. Nformation, and E. T. Des, “Contribution à l ’ étude des langages de transformation de modèles,” *Sci. York*, 2006.

[15] F. Jouault, “The ATLAS Transformation Language (ATL) ATL project Transforming models with ATL ATL Project Goals Operational Context of ATL,” p. 44322,2015.

[16] M. Nikolaidou, “A SysML Profile for Classical DEVS Simulators,” 2006.

[17] S. Engineering and E. A. M. Approach, *The SysML notation*. 1900.

[18] G. D. Kapos, V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos, “From SysML models to DEVS executable code : The role of DEVS-XML.”

[19] P. Roques and J. G. Professeur, “Lycée de la Communication de Metz SYSML Langage de Modélisation de SYStèmes « Systems Modeling Language ».”

[20] P. Roques, *SysML par l’exemple - Un langage de modélisation pour systèmes complexes*. 2009.

[21] I. Atl *et al.*, “ATL / Guide du développeur Code source ATL.”

[22] V. V. Graciano Neto *et al.*, “Stimuli-SoS: a model-based approach to derive stimuli generators for simulations of systems-of-systems software architectures,” *J. Brazilian Comput. Soc.*, vol. 23, no. 1, 2017.

[23] S. Mittal, B. P. Zeigler, J. L. Risco-Martín, F. Sahin, and M. Jamshidi, “Modeling and Simulation for Systems of Systems Engineering,” *Syst. Syst. Eng.*, pp. 101–149, 2008.

[24] Stéphane Garredu, “Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS,” 2014.

[25] S. N. Jansen, “Block definition diagrams in the Systems Modeling Language (SysML),” 2014.

[26] G. Eyrolles, “No Title,” 2005.

[27] A. Shatalin and A. Tikhomirov, “Graphical modeling framework architecture overview,” *Eclipse Model. Symp.*, 2006.

[28] Métamodèle et Paradigme du modèle DEVS, consulte le 01 avril 2018,

https://www.researchgate.net/figure/DEVS-model-paradigm_fig1_221159069

[29] Tutoriels ATL , Créer une transformation ATL simple, consulté le 22 02 2018,

http://www.eclipse.org/atl/documentation/basicExamples_Patterns/

[30] Comparaison de langage de transformation, outils MDA, consulté le 01 05 2018,

https://fr.slideshare.net/medshili/comparaison-de-outils-mda

[31] download éclipse modeling tools , consulté le 15 01 2018,

https://www.eclipse.org/downloads/packages/release/Oxygen/3A

[32] Transformation M2M avec atL, consulet le 20 04 2018,

https://fr.slideshare.net/HalimaBouabdelli/transformation-m2m-avec-atl

[33] définition d'éditeur de diagramme de modèle Ecore , consulté le 05 05 2018,

https://sourceforge.net/projects/dynamicgmf/