REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

N° :……..

DOMAINE : MATHEMATIQUES ET INFORMATIQUE

FILIERE : INFORMATIQUE

OPTION : RESEAUX

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par: Bakri Asma

Intitulé

Développement d'un système de vérification des systèmes embarqués

Soutenu devant le jury composé de :

	Université de M'sila	Président
Meliouh Amel	Université de M'sila	Rapporteur
	Université de M'sila	Examinateur

Année universitaire : 2016 /2017

"قَالُوا سُبْحَانَكَ لَا عِلْمَ لَنَا إِلَّا مَا عَلَّمْتَنَا إِنَّكَ
أَنْتَ الْعَلِيمُ الْحَكِيمُ"

صدق الله العظيم

سورة البقرة الآية 32

Dédicace

À celui qui m'a toujours soutenu et m'a été l'ami et le conseiller, à mon père j'adore, en signe de reconnaissance pour tous les sacrifices lui ma supporté et ma donnée la force d'être aujourd'hui ce que je suis.

À celle que m'a fait voir la lumière la personne le plus chère à mes yeux, à ma mère qui à tout sacrifie pour ses enfants, qui veillé à notre édition, qui, sans elles je ne serai pas que je suis.

À mes grandes- parents qui sont le symbole de bonté et de gentillesse.

À mes parents que Dieu me les garder.

À frères Ishek et Sadem houssine.

À mes sœurs Fatiha , Fahima,Karima et Achouak.

À MA FIANSI BAKRI LOTFI .

À mes oncles et tantes ainsi mes cousins et cousines paternels et maternels.

À mes camarade et copains d'étude et tous la promotion 2012-2017 du département math et informatique.

À tous le personne que connaisse et que je n'ai pas citées.

À tous ceux que j'aime.

Je dédie le fruit de mes efforts.

Asma

Remerciements

Au nom d'Allah le clément le miséricorde

Nous ne savons pas comment exprimer ou commencer notre gratitude vers Dieu, le tout puissant, qui nous a donné la santé, l'énergie, le courage et la volonté pour réaliser ce projet et pour toutes ces faveurs incomptable ;

Un grand merci pour l'encadreur *Meliouh Amel* pour nous avoir proposé cet important sujet et pour les précieux conseils et les orientations. Nous remercions pour sa présence personnelle permanente, ses critiques constructives, ses explications et suggestions pertinentes qui nous aide.

Et enfin, pour avoir apporté tant de soins à la réalisation de ce projet de fin d'études ;

Nous remercions également tous les enseignants de la faculté des mathématiques et de l'informatique pour toutes les connaissances qu'ils nous ont inculquées ;

Nous tenant aussi à remercier les membres du jury qui ont accepté d'examiner notre mémoire ;

Merci pour tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail ;

Que Dieu nous protège et accepte notre travail.

TABLE DES MATIERES

Chapitre 3: Vérifications des systèmes embarquées

Introduction générale

Introduction générale

Aujourd'hui, les applications embarquées sont de plus en plus variées et elles prendront de plus en plus de place dans notre vie quotidienne de demain. Réservés, il y a quelques années, aux applications industrielles, les systèmes embarqués font leur apparition dans beaucoup d'autres secteurs tels que le transport, le multimédia, les consoles de jeux, ... En termes de complexité, les systèmes embarqués couvrent un large spectre allant du simple microcontrôleur (pour le contrôle de la fermeture/ouverture d'une vanne, par exemple), jusqu'aux systèmes répartis (pour le contrôle du trafic aérien, par exemple).

La vérification est une technique qui permet de vérifier des modèles, en considérant les propriétés qu'on veut vérifier. Cette action se fait par des outils manuels ou automatiques comme (SPIN, SMV..).

Le Model-Checker de Maude est le plus utile dans ce domaine, il se base sur le langage Maude et les règles de la logique de réécriture et la logique temporelle. Il s'applique automatiquement sur une spécification écrite dans la logique temporelle.

La vérification avec model-checking peut être appliquée sur les diagrammes du langage de modélisation UML (Unified Modeling Language). Car le modèle en UML peut avoir besoin d'être vérifié s'il est correct et répond exactement aux besoins fonctionnels du système. Alors le Model-Checker retourne la valeur boolienne true si les conditions spécifiée sont valides.

Les systèmes embarques sont des systèmes complexes, qui posent un problèmes lors de leurs modélisation et vérification de leur fonctionnement, l'une des solution est de les modéliser et générer un code viable à partir de ces modèles

Le but de ce travail consiste à proposer une approche de vérification formelle des diagrammes UML ; modélisant un système embarqué, plus particulièrement les diagrammes d'état transitions et de collaboration temporisés afin de vérifier certaines propriétés temporelles.

La vérification commence par la génération du code Maude à partir des diagrammes UML,utilisant l'outil ATOM3. Le code généré va être vérifié ensuite par l'outil de

vérification (Model-Checker) qui supporte le langage Maude . Le Model-Checker va vérifier des propriétés temporelles spécifiées avec la logique temporelle.

Ce mémoire est organisé en trois chapitres :

Le premier décrit l'approche MDA (Model Driven Architecteur) ,ses principes et fondements ainsi que l'IDM

le deuxième chapitre concerne les systèmes embarques , leurs définition et caractéristiques.

Dans le troisième chapitre , nous présenterons notre contribution dans ce travail

CHAPITRE 1

LES SYSTEMES EMBARQUEES

CHAPITRE 1

LES SYSTEMES EMBARQUEES

1. Introduction :

Actuellement les systèmes embarqués sont fréquemment utilisés dans tous les domaines .Dans ce chapitre nous présentons les principes de ces systèmes , leurs domaines d'application, leurs caractéristiques et leur Architecture générale avec des exemples illustratifs

2. Bref historique des systèmes embarqués :

- 1967 : Apollo Guidance Computer, premier système embarqué. Environ un millier de circuits intégrés identiques (portes NAND).
- 1960-1970 : Missile Minuteman, guidé par des circuits intégrés.
- 1971 : Intel produit le 4004, premier microprocesseur, à la demande de Busicom. Premier circuit générique, personnalisable par logiciel.
- 1972 : lancement de l'Intel 8008, premier microprocesseur 8 bits (48 instructions, 800kHz).
- 1974 : lancement du 8080, premier microprocesseur largement diffusé. 8 bits, (64KB d'espace adressable, 2MHz - 3MHz).
- 1978 : création du Z80, processeur 8 bits.
- 1979 : création du MC68000, processeur 16/32 bits. [14]

3. Définition :

Un Système: Toute "chose" constituée de parties organisées pour assurer une fonction ou un ensemble de fonctions dans son environnement. [2]

Selon la langue de Molière, le mot embarqué, est dérivé du verbe « embarquer » qui désigne le fait de « mettre quelque chose à bord d'un navire, un avion ou d'un véhicule ». Informatiquement parlant un système embarqué (appelé aussi un système enfui) peut être défini comme : « Un système électronique et informatique autonome, qui est dédié à une tâche bien précise ». [13]

4. Les types des systèmes embarqués

Il existe plusieurs types de systémes embarqués, parmi les types on peut citer les types suivants :

- **calcul générale** : dans ce type de systémes embarqués, on peut citer l'exemple suivant de jeu vidéo
- **contrôle de systèmes en temps réel** : dans ce type de systémes embarqués, on peut citer l'exemple suivant de système de navigation aérien
- **traitement du signal** : dans ce type de systémes embarqués, on peut citer l'exemple suivant de radar, sonar
- **transmission d'information et commutation** : dans ce type de systémes embarqués, on peut citer l'exemple suivant de téléphone, internet [6]

5. Domaines d'application

On trouve les systèmes embarquées dans plusieurs domaines, parmi ces domaines on peut citer les domaines suivants :

- ✓ Transport comme automobile.
- ✓ Astronautique comme satellite artificiel.
- ✓ Militaire comme missile.
- ✓ Télécommunication comme téléphonie, routeur ,téléphone portable.
- ✓ Electroménager comme télévision
- ✓ Impression comme imprimante multifonctions
- ✓ Informatique comme disque dur, lecteur de disquette
- ✓ Multimédia comme console de jeux vidéo
- ✓ Guichet automatique bancaire (GAB).
- ✓ Equipement médical.
- ✓ Automate programmable industriel.
- ✓ Autre …

6. Caractéristiques d'un système embarqué

Les systèmes embarqués contient plusieurs caractéristiques, parmi ces caractéristiques on a :

- Fonctionnement en Temps Réel :

✓ Réactivité : des opérations de calcul doivent être faites en réponse à un événement extérieur (interruption matérielle).

✓ La validité d'un résultat (et sa pertinence) dépend du moment où il est délivré.

✓ Rater une échéance va causer une erreur de fonctionnement.

- Temps Réel dur : plantage.
- Temps Réel mou : dégradation non dramatique des performances du système.

✓ Beaucoup de systèmes sont « multirate » : traitement d'informations à différents rythmes.

- Faible encombrement, faible poids :

✓ Electronique « pocket PC », applications portables où l 'on doit minimiser la consommation électrique (bioinstrumentation...).

✓ Difficulté pour réaliser le packaging afin de faire cohabiter sur une faible surface électronique analogique, électronique numérique, RF sans interférences.

- Faible consommation :

✓ Batterie de 8 heures et plus (PC portable : 2 heures).

- Environnement :

✓ Température, vibrations, chocs, variations d'alimentation, interférences RF, corrosion, eau, feu, radiations.

✓ Le système n'évolue pas dans un environnement contrôlé.

✓ Prise en compte des évolutions des caractéristiques des composants en fonction de la température, des radiations...

- Fonctionnement critique pour la sécurité des personnes. Sûreté :

✓ Le système doit toujours fonctionner correctement.

✓ Sûreté à faible coût avec une redondance minimale.

✓ Sûreté de fonctionnement du logiciel

✓ Système opérationnel même quand un composant électronique lâche.

✓ Choix entre un design tout électronique ou électromécanique.

- Beaucoup de systèmes embarqués sont fabriqués en grande série et doivent avoir des prix de revient extrêmement faibles, ce qui induit :

✓ Une faible capacité mémoire.

✓ Un petit processeur (4 bits). Petit mais en grand nombre !

- La consommation est un point critique pour les systèmes avec autonomie.

✓ Une consommation excessive augmente le prix de revient du système embarqué car il faut alors des batteries de forte capacité.

- Faible coût :

✓ Optimisation du prix de revient.[7]

7. Architecture d'un système embarqué

Le système est composé de plusieurs couches. La couche la plus abstraite est le logiciel spécifique de l'application. Cette couche communique avec une deuxième couche s'appelant le système d'exploitation qui contient deux parties : la première partie de la gestion des ressources et la deuxième partie de communication logicielle, suivie par la couche de réseau de communication matérielle embarquée, et à la fin on trouve la couche des composants matériels qui est la plus basse [4].

- La couche basse: comprend les composants matériels du système, parmi ces composants, on trouve les composants standards : (DSP, MCU,…etc).

- La couche de réseau de communication matérielle embarquée: contient les dispositifs nécessaires à la communication entre les composants.

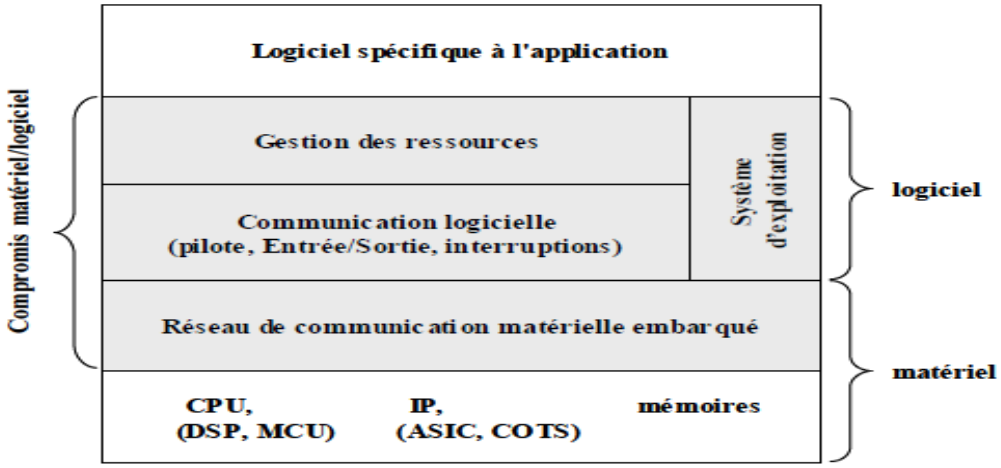

Figure 11: représentation des différentes couches d'un système embarqué [11]

Le système d'exploitation comprend deux couches:

1) Une couche de communication logicielle: composée des pilotes d'entrées/sorties, des interruptions, ainsi que les contrôleurs qui sont utilisées pour contrôler le matériel. Le logiciel de cette couche et le matériel sont intimement liés. Le logiciel et le matériel sont séparés grâce à cette couche.

2) Une couche de gestion des ressources: qui utilise des structures de données particulières et des accès non standards que l'on ne trouve pas des OS standards, par contre, les couches de

gestion des ressources des OS standards sont souvent volumineuses afin d'être embarquées. La couche de gestion des ressources permet d'isoler l'application de l'architecture.

- La couche de logiciel spécifique de l'application: est la couche la plus abstraite, elle communique avec la couche de plus bas niveau qui est la couche système d'exploitation.

8. La structure de base d'un système embarqué

Le système embarqué interagit avec son environnement pour lequel il rend des services bien précis (contrôle, surveillance, communication, …). Une information est captée par l'environnement, une transformation est réalisée sur cette information, avant d'être lisible par le cœur du système embarqué (constitué d'une partie matérielle et une partie logicielle), qui effectue un traitement spécifique à cette information pour rendre à son tour une réponse à son environnement, cette réponse doit être transformée avant d'être envoyée à l'environnement.

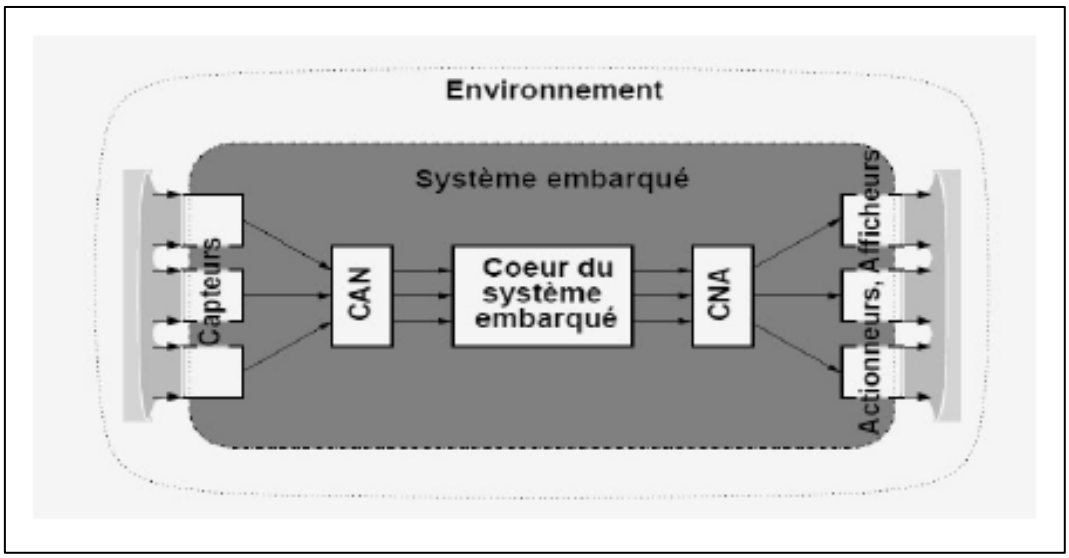

Figure 12: Structure de base d'un système embarqué dans son environnement

9. Complexité grandissante d'un système embarque

Dans les systémes embarqués, on a certains obstacles, parmi ces obstacles :

- Les systèmes embarqués sont de plus en plus complexes.
- Il est difficile de réfléchir à une solution complète optimisée du premier jet.
- Il est de plus en plus difficile de corriger les « bugs ».
- Il est de plus en plus difficile de maintenir le système au cours du temps (obsolescence des composants…).

• En conséquence, l'approche classique de développement d'un système embarqué doit changer... [7]

10.Quelques exemples

On peut citer quelques exemples :

Équipement mobile et bureautiques :

- ✓ Répondeurs,
- ✓ Copieurs,
- ✓ Téléphone portable,
- ✓ Imprimante.

Équipement dans le bâtiment :

- ✓ Ascenseurs, escalators,
- ✓ Système de surveillance,
- ✓ Contrôle d'accès,
- ✓ Systèmes d'éclairage.

Équipement de production :

- ✓ Productions automatisées,
- ✓ Systèmes de commande d'énergie,
- ✓ Équipements de stockage,

Transport :

- ✓ Avionique,
- ✓ Trains, Automobiles (+ de 100 processeurs),
- ✓ Contrôle de navigation,

Communications :

- ✓ Satellites,
- ✓ GPS,
- ✓ Téléphonie mobile,[6]

11.Conclusion

Dans ce chapitre, nous avons présenté de façon globale les systèmes embarqués.

Et dans le prochaine chapitre nous présentes présenté quelques concepts de l'architecture (L'architecture, terme issu du latin architectura, mot tiré du grec...) dirigée par les modèles (MDA).

CHAPITRE 2

MDA (Le Model Driven Architecture)

CHAPITRE 2

MDA (Le Model Driven Architecture)

1. Introduction

Dans ce chapitre nous présentons le Model Driven Architecture (MDA), leurs architecture, notion de modèle et méta-modèle, Transformation des modèles, Objectifs de l'approche MDA et les outils utilisée pour la modélisation.

2. Présentation de MDA

2.1. Définition de MDA

Le Model Driven Architecture (MDA) est un aspect de déploiement présentée par l'OMG. Elle permet de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme donnée. A cette fin, le MDA définit une architecture de spécifications structurée en modèles indépendants des plates-formes (PIM) et en modèles spécifiques (PSM).

L'approche MDA permet d'accomplir le même modèle sur certaines plates-formes grâce à des projections standardisées. Elle agrée aux applications d'inter opérer en reliant leurs modèles et supporte l'évolution des plates-formes et des techniques. La mise en œuvre du MDA est entièrement reposée sur les modèles et leurs transformations. [9]

Pourquoi le MDA?

L'OMG voulait f1aire de CORBA «LE» middleware, mais il est loin d'être le seul à être utilisé. [9]

D'autres comme EJB ou .NET sont largement répandus. C'est en partant de cette observation que l'OMG s'est engagé sur la voie du MDA, afin de résoudre les problèmes d'interopérabilité et de portabilité dès le niveau modélisation.

Le MDA se veut donc indépendant de toute plate-forme et de tout système, il permet de concevoir des applications portables au niveau des langages de programmation, des systèmes d'exploitation mais aussi des middlewares. Cette indépendance totale doit permettre de changer d'infrastructure sans perdre ce qui a déjà été conçu.

Cette approche permet ainsi de capitaliser le travail effectué pendant les phases d'analyse et de conception.

2.2. L'architecture de MDA

L'architecture du MDA se découpe en quatre couches qui réfèrent à chaque fois à des standards déjà adoptés par l'industrie ou en cours de normalisation (Figure 2.1).Le centre est composé par trois technologies spécifiées par l'OMG pour modéliser la logique métier de l'application: UML (Unified Modeling Language), MOF (Meta-Object Facility) et CWM (Common Warehouse Metamodel). Dans la couche suivante, se trouve aussi un standard XMI (XML Metadata Interchange) qui permet le dialogue entre les middlewares (Java, CORBA, .NET et web services)[9].

La troisième couche contient les services qui permettent de gérer les évènements, la sécurité, les répertoires et les transactions. Enfin, la dernière couche propose des Frameworks spécifiques au domaine d'application (Finance, Télécommunication, Transport, Espace, médecine, commerce électronique, manufacture,…).

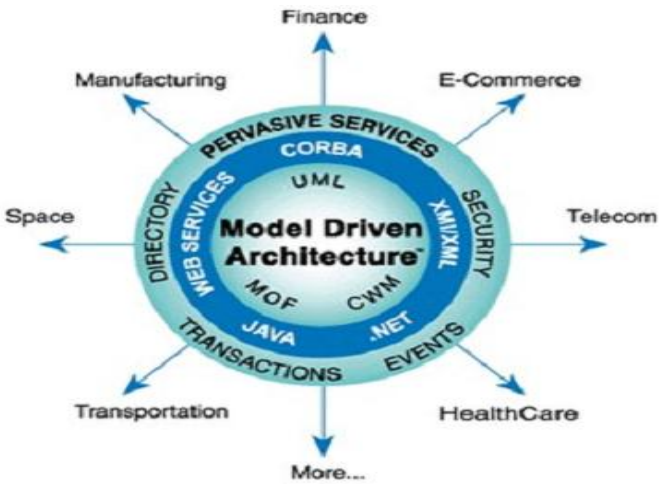

Figure 2.1 : L'Architecture de MDA[9]

3. Notion de modèle et méta-modèle

3.1. Le modèle

Le dictionnaire de la langue française en ligne TLFI [18] (Trésor de la langue française informatisé) donne les définitions suivantes du mot « modèle » :

- Arts et métiers : représentation à petite échelle d'un objet destiné à être reproduit dans des dimensions normales.
- Épistémologie : système physique, mathématique ou logique représentant les structures essentielles d'une réalité et capable à son niveau d'en expliquer ou d'en reproduire dynamiquement le fonctionnement.

- Cybernétique : système artificiel dont certaines propriétés présentent des analogies avec des propriétés, observées ou inférées, d'un système étudié, et dont le comportement est appelé, soit à révéler des comportements de l'original susceptibles de faire l'objet de nouvelles investigations, soit à tester dans quelle mesure les propriétés attribuées à l'original peuvent rendre compte de son comportement manifeste.

3.2. Le méta-modèle

est un modèle d'un langage de modélisation , le terme ''méta'' indique un niveau plus élevé, soulignant le fait qu'un méta-modèle décrit un langage de modélisation à plus haut niveau d'abstraction que le langage de modélisation lui-même .Un méta-modèle possède deux caractéristiques essentielles. Premièrement, il doit capturer les caractéristiques essentielles du langage de modélisation. Deuxièmement, il devrait être capable de décrire la syntaxe concrète, et la sémantique de ce langage.

La relation entre un élément d'un modèle et un ou plusieurs éléments de son méta-modèle est appelée la relation de **conformité**. Cette relation est notée χ.

3.3. L'architecture de méta-modélisation adoptée par l'OMG

L'OMG, dans le cadre de ses travaux concernant la méta-modélisation, a défini la notion de méta-méta-modèle ainsi que la standardisation d'une architecture traditionnelle décrivant les liens entre modèles, méta-modèles et méta-méta-modèles. Cette architecture est hiérarchisée en quatre étapes comme le montre la Figure 2.4. [12]

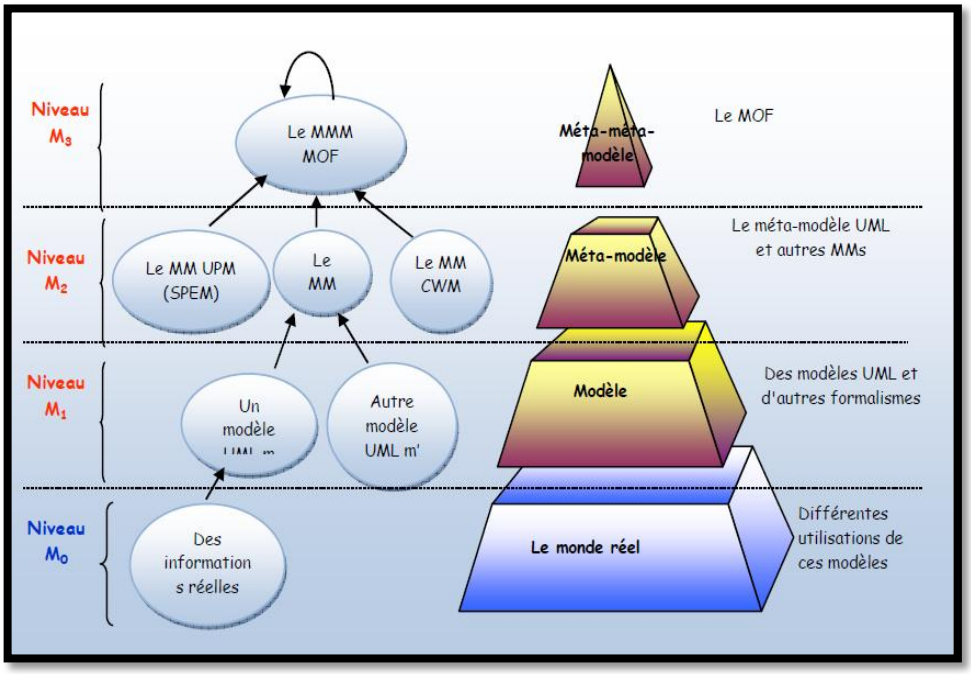

Figure 2.4 :L'architecture à quatre niveaux de la méta-modélisation adoptée par l'OMG

- *Niveau M3 (MMM):* Le niveau M3 (ou méta-méta-modèle) est composé d'une seule entité qui s'appelle le MOF (Meta Object Facility). Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif,
- *Niveau M2 (MM):* Le niveau M2 (ou méta-modèle) définit le langage de modélisation et la grammaire de représentation des modèles M1. Le méta-modèle UML, qui définit la structure interne des modèles UML suivant le standard UML, fait partie de ce niveau. Les profils UML, qui étendent le méta-modèle UML, appartiennent aussi à ce niveau. Les concepts définis par un méta-modèle sont des instances des concepts du MOF. [12]
- *Niveau M1 (M)***:** Le niveau M1 (ou modèle) est composé de modèles d'information. Il décrit les informations de M0. Les modèles UML, les PIM et les PSM appartiennent à ce niveau. Les éléments d'un modèle (M1) sont des instances des concepts décrits dans un méta-modèle (M2).
- *Niveau M0***:** Le niveau M0 (ou instance) correspond au monde réel. Il ne s'agit pas à proprement parler d'un niveau de modélisation. Ce niveau contient les informations réelles de l'utilisateur.

Il est important de noter que la proposition initiale d'OMG dans l'approche MDA était d'utiliser le langage UML comme unique langage de modélisation. Cependant, les limites d'UML ont rapidement été atteintes et il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin d'exprimer de nouveaux concepts relatifs à des domaines spécifiques. Ces extensions devenant de plus en plus importantes et la communauté MDA a élargi son point de vue en considérant des langages de modélisation spécifiques à un domaine. [12]

3.4. L'objectif de méta-modélisation

La modélisation est une méthode de conception de systèmes en utilisant un certain nombre de concepts prédéfinis. La méta-modélisation est une façon de définitions des concepts à utiliser pour préciser des systèmes. Deux expressions clés récapitulant cette approche sont [10]:

-«Une tentative pour décrire le monde » :

-« dans un objectif bien particulier».

De cette définition on peut déduire les aspects suivants d'un méta-modèle :

- il ne peut pas y avoir un méta-modèle universel utilisable pour décrire tous les systèmes informatiques.
- un méta-modèle doit être défini pour un objectif précis. Et donc, qu'il existe une multitude de méta-modèle.
- Le dernier aspect d'un méta-modèle, en plus des concepts et des relations du domaine d'application, est la sémantique. Il est en effet important de donner un sens aux éléments définis dans un méta-modèle.

4. Transformation des modèles

4.1. Mise en œuvre

La démarche MDA supporte toutes les étapes de développement et standardise les passages de l'une à l'autre. Elle peut se découper en quatre points (Figure 2.2), les points 2 et 4 peuvent être répétés un nombre indéterminé de fois :

1. la réalisation d'un modèle indépendant de toute plate-forme appelé PIM pour « Platform Independent Model ».
2. l'enrichissement de ce modèle par étapes successives.
3. le choix d'une plate-forme de mise en œuvre et la génération du modèle spécifique correspondant appelé PSM pour « Platform Specific Model ».
4. le raffinement de celui-ci jusqu'à obtention d'une implantation exécutable.

   Dans une démarche MDA, tout est considéré comme modèle, aussi bien les schémas que le code source ou le code binaire. Les deux types de modèles identifiés sont donc les PIM et les PSM [StOSSG].

Chaque étape correspond à la transformation d'un modèle vers un autre (du même type ou non).

4.2. PIM et PSM

4.2.1. Platform Independent Model (PIM)

La première étape du processus MDA comporte à accomplir un modèle indépendant de toutes plateformes (PIM), exprimé en UML. Il existe certains niveaux de PIM mais tous sont indépendants de n'importe quelle plate-forme [9].

Le PIM de base représente uniquement les capacités fonctionnelles métiers et le comportement du système, sans "dégradations" par des considérations technologiques. La

clarté de ce modèle doit permettre à des experts du domaine de le comprendre bien mieux qu'un modèle d'implémentation.

Ils peuvent ainsi vérifier plus facilement que le PIM est complet et correct. Un autre bénéfice de l'indépendance technique de ce modèle est qu'il garde tout son intérêt au cours du temps et doit être modifié uniquement si les connaissances ou besoins métiers changent.

Les PIM suivants intègrent des aspects technologiques et architecturaux mais toujours sans détails spécifiques à une plate-forme. Ces modèles peuvent, par exemple, contenir des informations sur la persistance, les transactions, la sécurité, etc. Ces concepts permettent de projeter plus précisément le modèle PIM vers un modèle spécifique (PSM).

4.2.2. Platform Specific Model (PSM)

Une fois le PIM suffisamment détaillé, il est projeté vers un modèle spécifique (PSM). Pour obtenir un modèle spécifique, il faut choisir la ou les plates-formes d'exécution (plusieurs plates-formes peuvent être utilisées pour mettre en œuvre un même modèle).

Les caractéristiques d'exécution et les informations de configuration qui ont été définies de façon générique sont converties pour tenir compte des spécificités de la plate-forme. Comme pour les PIM, il existe plusieurs niveaux de PSM. Le premier, sous forme d'un schéma UML, est obtenu directement à partir du modèle PIM, les autres sont obtenus par transformations successives jusqu'à l'obtention du système exécutable.

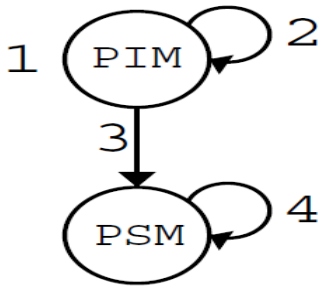

Figure 2.2 : Les étapes d'une démarche MDA [9]

4.3. La définition de transformation des modèles

Il existe quatre types de transformations différentes que nous allons montrer, les trois premières conviennent à ceux que l'on trouve dans la démarche habituelle que nous avons présentée, la finale doit permettre de "récupérer" dans un processus MDA des modèles spécifiques [9].

4.4. Les différents types de transformation

4.4.1. Transformation PIM vers PIM

Ces transformations visent à enrichir, filtrer ou spécialiser le modèle pendant le cycle de développement sans nécessiter d'informations dépendantes d'une plate-forme. Les transformations PIM vers PIM sont généralement utilisées pour le raffinement de modèle.

4.4.2. Transformation PIM vers PSM

Ces transformations sont utilisées quand le PIM est suffisamment raffiné pour être projeté vers une plate-forme d'exécution. Cette projection est basée sur les caractéristiques de cette plate-forme.

Ces caractéristiques doivent être décrites à l'aide d'UML et éventuellement d'un profil.

Le passage d'un modèle de composants "logique" à un modèle de composants existants comme EJB ou CCM est une transformation PIM vers PSM.

Elles permettent d'obtenir un modèle spécifique à une plate-forme donnée à partir d'un modèle indépendant.

Les PSM possibles peuvent être CORBA, Java/EJB, XML/SOAP ou toute autre plate-forme supportée par le MDA (Figure 2.3).

4.4.3. Transformation PSM vers PSM

Ces transformations s'appliquent sur un modèle spécifique et donnent un autre modèle spécifique à la même plate-forme. Elles sont utilisées pour la réalisation et le déploiement de composants.

La génération de code, la compilation, la mise en paquetages, l'initialisation et la configuration font parties de ce type de transformation.

4.4.4. Transformation PSM vers PIM

Ces transformations doivent permettre d'obtenir un modèle indépendant à partir d'une implantation existante sur une plate-forme spécifique. Bien que celles-ci ne fassent pas directement partie du processus.

MDA, elles doivent permettre "d'intégrer" des applications existantes afin de pouvoir les utiliser dans un processus MDA. Ce sont certainement les transformations les plus difficiles à automatiser.

Idéalement, le résultat de cette transformation devrait correspondre à la transformation inverse PIM vers PSM.

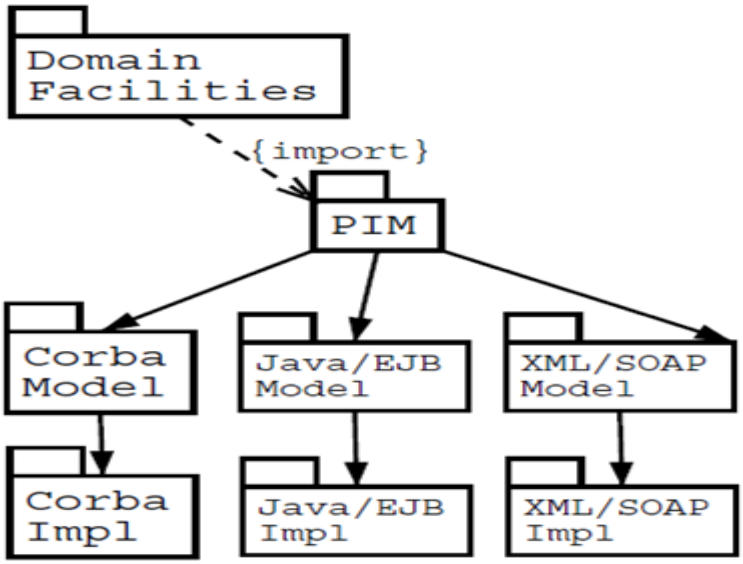

Figure 2.3 : Projections multi-plates-formes[9].

4.5. Le principe de transformations de modèles

Une transformation de modèle est un ensemble de règles qui s'appliquent aux éléments d'un méta-modèle définissant les modèles sources valides (c'est-à-dire auxquels la transformation peut s'appliquer) et qui définissent pour chacun de ces éléments leur(s) équivalent(s) parmi les éléments d'un méta-modèle cible. Le moteur de transformation (aussi appelé « transformateur de modèles ») lit le modèle source, qui doit donc être conforme au méta-modèle source, et applique les règles définies dans la transformation de modèle afin de créer le modèle cible qui sera conforme au méta-modèle cible. Les méta-modèles source et cible peuvent faire partie de la définition de langages de modélisation et, dans ce sens, les transformations de modèles peuvent être considérées comme des systèmes de traduction (ou de réécriture) d'un langage vers un autre [15].

Le principe général présenté ici pour les transformations de modèles est indépendant de la syntaxe concrète et de la sémantique associées aux méta-modèles source et cible. Concernant les différences de syntaxe concrète, il est à noter que les transformations modèle-à-modèle et modèle-à-texte sont souvent distinguées l'une de l'autre dans la littérature.

4.6. La structure d'une règle de transformation

Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible.

Une transformation de modèle est notamment caractérisée par la réunion des éléments suivants : des règles de transformation, une relation entre la source et la cible, un ordonnancement des règles, une organisation des règles, une traçabilité et une direction.

4.6.1. Règles de transformation

Une règle de transformation contient deux parties : une partie gauche (left-hand side «LHS») et une partie droite (right-hand side «RHS»). La LHS exprime des accès aux modèles sources, alors que la RHS indique les expansions (création, modification, suppression) dans les modèles cibles. Chacune des deux parties peut être représentée par une combinaison de:

- Variables : une variable contient un élément de modèle (source ou cible) ou une valeur intermédiaire nécessaire à l'expression de la règle.
- Patterns : un pattern désigne un fragment de modèle et peut contenir des variables. Il peut être représenté à l'aide d'une syntaxe abstraite ou concrète dans le langage des modèles correspondants. Cette syntaxe peut être textuelle ou graphique.
- Logique : une logique permet d'exprimer des calculs et des contraintes sur les éléments de modèle. Cette expression peut être non exécutable (expression de relations entre modèles) ou exécutable. Une logique exécutable peut être sous une forme déclarative ou impérative.

4.6.2. Relation entre les modèles source et cible

Pour certains types de transformations, la création d'un nouveau modèle cible est nécessaire. Pour d'autres, la source et la cible sont le même modèle, ce qui revient en fait à une modification de modèle.

4.6.3. Organisation des règles

L'organisation des règles définit comment composer plusieurs règles de transformation. Les règles peuvent être organisées de façon modulaire, avec la notion d'importation. Les règles peuvent également utiliser la réutilisation, par le biais de mécanismes d'héritage entre règles, ou la composition, par le biais d'un ordonnancement explicite. Enfin, les règles peuvent être organisées selon une structure dépendante du modèle source ou du modèle cible.

4.6.4. Ordonnancement des règles

Les mécanismes d'ordonnancement déterminent l'ordre dans lequel les règles sont appliquées. Dans le cas d'un ordonnancement implicite, l'algorithme d'ordonnancement est défini par l'outil de transformation. Dans le cas d'un ordonnancement explicite, des

mécanismes permettent de spécifier l'ordre d'exécution des règles. Cet ordre d'exécution peut être défini de manière externe ou interne : tandis qu'un mécanisme externe établit une séparation claire entre les règles et la logique d'ordonnancement, un mécanisme interne permet aux règles d'invoquer d'autres règles. Enfin, l'ordonnancement des règles se repose également sur des conditions, des itérations ou sur une séparation en plusieurs phases, certaines règles ne pouvant être appliquées que dans certaines phases.

4.6.5. Traçabilité

Les transformations peuvent stocker les corrélations entre les éléments des modèles source et cible. Certaines approches fournissent des mécanismes dédiés pour supporter la traçabilité. Dans les autres cas, le développeur doit implémenter la traçabilité de la même manière qu'il crée n'importe quel autre lien dans un modèle.

4.6.6. Direction

Les transformations peuvent être unidirectionnelles ou bidirectionnelles. Dans le premier cas, le modèle cible est calculé ou mis à jour sur la base du modèle source uniquement. Dans le second cas, une synchronisation entre les modèles source et cible est possible.

5. Objectifs de l'approche MDA

L'OMG a défini l'approche MDA pour répondre aux problèmes liés à l'évolution continue des technologies. MDA est à l'origine de l'ingénierie dirigée par les modèles. Tout ce qui constitue l'approche MDA, la modélisation, les transformations, les technologies de mises en œuvre… a été pensé afin de permettre à MDA de remplir les objectifs suivants :

- **La pérennité des savoir-faire** : La technologie évolue plus vite que les métiers de l'entreprise. Il semble alors plus intéressant de capitaliser les savoir-faire de l'entreprise indépendamment des préoccupations techniques. L'approche MDA vise principalement à rendre les spécifications métier des entreprises pérennes, indépendamment des technologies de mise en œuvre. Cette pérennité est possible grâce aux modèles, qui sont par nature des entités pérennes, et à l'utilisation d'UML qui est un standard stable et largement répandu.

- **Les gains de productivité** : Les modèles sont au cœur du processus MDA, ils facilitent la communication entre experts du domaine et développeurs, mais pas seulement. Dans MDA les modèles deviennent un outil de production grâce à l'automatisation des transformations de modèles. Le modèle devient un élément qui véhicule une information utile,

précise et nécessaire, permettant l'obtention du code source de l'application par génération automatique du code. L'automatisation des transformations que permettent les modèles dans le processus de développement MDA, entraîne un gain de productivité.

- **La prise en compte des plateformes d'exécution** : À partir du PIM, MDA permet d'intégrer les spécifications techniques d'une plateforme d'exécution dans les transformations PIM vers PSM, et d'obtenir ainsi un PSM spécifique à la plateforme d'exécution, à partir duquel le code source de l'application est généré. Grâce à ce procédé, une application tire pleinement parti des fonctionnalités de la plateforme d'exécution cible. Le développement d'applications multiplateformes, ou encore la migration logicielle, sont facilités puisqu'il suffit à partir du PIM, d'effectuer les transformations PIM vers PSM en y intégrant à chaque fois, le modèle de la plateforme concernée. Obtenant ainsi pour chaque plateforme, son PSM à partir duquel le code source sera généré.

6. Les standards de MDA

Pour obtenir une telle efficacité, plusieurs outils conceptuels sont mis à disposition.
La technologie MDA (Model Driven Architecture) est supportée par l'OMG (Object Management Group), qui propose également UML (Unified Modeling Language) et Corba (Object Request Broker). Ces outils sont :

- **UML**, largement utilisé par ailleurs, qui permet une mise en oeuvre aisée de MDA en offrant un support connu.
- **XMI**, XML Metadata Interchange, qui propose un formalisme de structuration des documents XML de telle sorte qu'ils permettent de représenter des méta-données d'application de manière compatible.
- **MOF**, Meta Object Facility, spécification qui permet le stockage, l'accès, la manipulation, la modification, de méta-données[10].
- **CWM**, base de données pour méta-données.

L'OMG n'a pas jugé utile de standardiser un processus associé à ces outils. Leur rôle est de répondre aux besoins des utilisateurs de manière générique, et non de proposer de solutions définitives pour certains types d'applications précises.

Un processus de génie logiciel exploitant les possibilités de MDA a cependant été proposé: le 'Model-Driven Software Development'[19].

7. ATOM^3

7.1. Présentation

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) est un outil de modélisation multi-paradigmes écrit en Python ,développé par le laboratoire MSDL (Modelling, Simulation and Design Lab) à l'université de McGill Montréal, Canada. Cet outil a été conçu en collaboration avec Juan de Lara de l'université de Madrid (UAM), Espagne [20].

Les deux principales fonctionnalités d'ATOM^3 sont la méta-modélisation et la transformation de modèles.

Dans AToM^3 les formalismes et les modèles sont décrits comme des graphes. Cet environnement génère un outil de manipulation graphique des modèles décrits dans un formalisme donné, à partir d'une méta-spécification de ce formalisme (généralement décrite dans le formalisme Entité/Relation). Les transformations entre modèles sont ensuite effectuées par réécriture des graphes (utilisant des grammaires de graphes) [20].

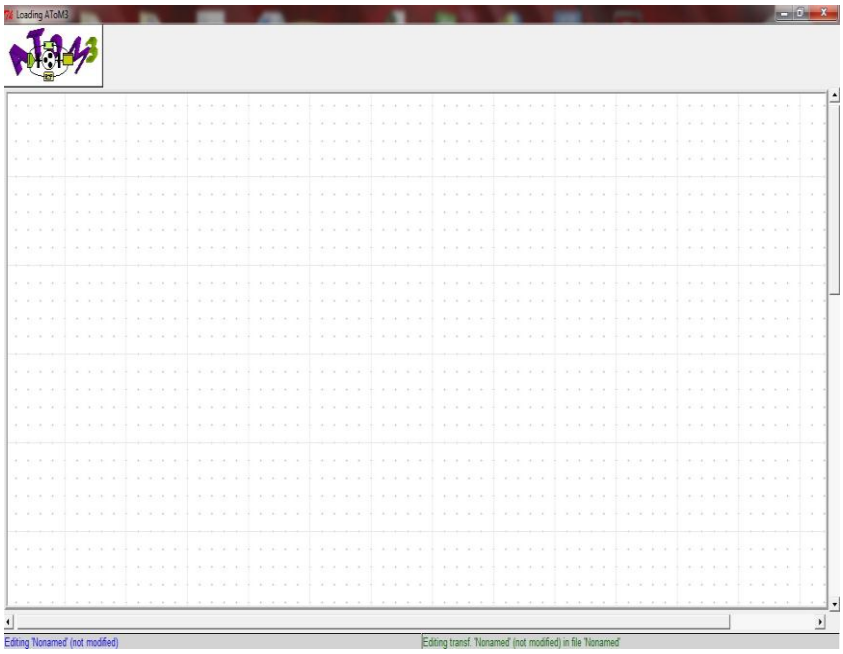

Figure 2.5 : Interface d'ATOM^3.

7.2. Architecture d'ATOM^3

Le composant principal d'ATOM^3 est le noyau (Kernel), qui est responsable du chargement, de la sauvegarde, de la création et de la manipulation des modèles (à tout méta-niveau). Par défaut, les méta-modèles et les méta-méta-modèles sont chargés quand ATOM^3

est lancé. Ces méta-méta-modèles permettent de modéliser les méta-modèles (modélisation des formalismes). Le formalisme Entité-Relation avec des contraintes est utilisé dans le méta-méta-niveau .Les contraintes sont du code en Python et le concepteur doit spécifier si ces contraintes sont (pré, post et sur quel événement une contrainte est évaluée).

Dans la modélisation à ce niveau, les entités qui doivent apparaître sur les modèles sont spécifiées ensemble avec leurs attributs, leurs apparences graphiques, leurs contraintes et leurs actions. Chaque relation entre deux entités peut être caractérisée par une contrainte en terme de cardinalités [21].

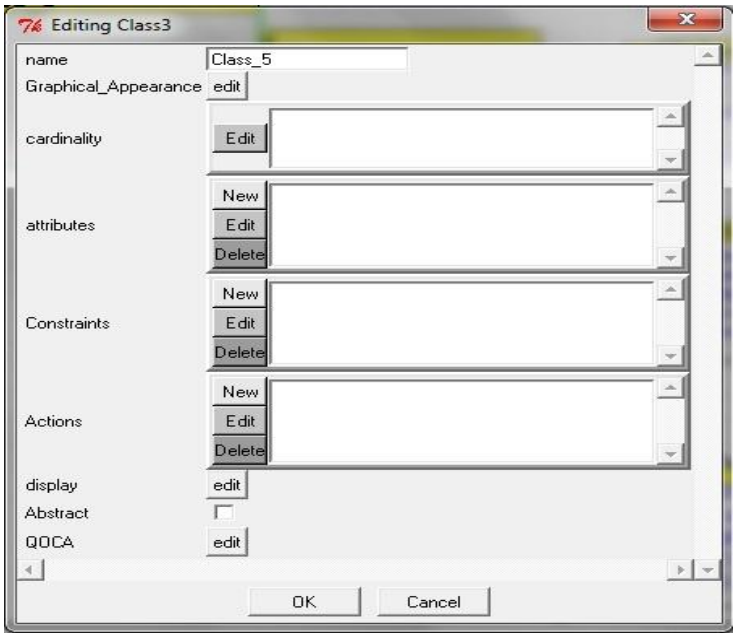

Figure 2.6 : Editions des caractéristiques d'une entité.

7.2.1. Les attributs

En générale, dans ATOM^3, il y a deux types d'attributs [21] :

- Les attributs réguliers, qui sont utilisés pour identifier les caractéristiques d'une entité.
- Les attributs générateurs, qui permettent de générer d'autres propriétés.

Chaque attribut a un type et un nom et une valeur initiale, comme il est montré dans la Figure 2.7.

Dans ATOM^3, chaque type a une classe Python, qui est responsable à l'édition des valeurs et les tests de leur validité. Il y a deux types de base :

- Type régulier tel que String, Integer, Float, Boolean…etc.
- Type générateur qui permet de générer des attributs, des contraintes, des attributs graphiques.

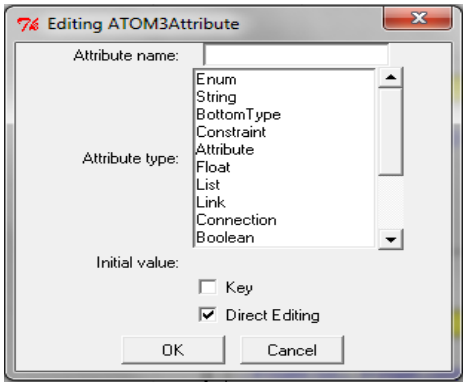

Figure 2.7 :Edition des valeurs d'un attribut.

7.2.2. Contraintes et actions

Dans ATOM^3 Une contrainte est un code en Python ou des expressions en OCL, il y a deux types de contraintes :

- Des contraintes locales associées à une entité, elle doit contenir que des attributs pour cette entité.
- Des contraintes globales, ou les informations de toutes les entités du modèle sont utilisées.

La Figure 2.6 montre la structure d'une contrainte. Une contrainte est composée de :

- Un nom de contrainte.
- Un événement déclenchant l'évaluation de la contrainte, l'événement peut être sémantique tel que la sauvegarde d'un modèle, création d'une entité ou graphique ou structurel, tel que le déplacement ou la sélection d'une entité.
- Une spécification quand la contrainte doit être évaluée, avant l'événement (pré-condition) ou après (post-condition).
- Une zone pour écrire un code Python ou en OCL.

Si la pré-condition d'un événement échoue alors ce dernier ne va pas être exécuté, si la post-condition d'un événement échoue ce dernier ne va pas être accompli [8].

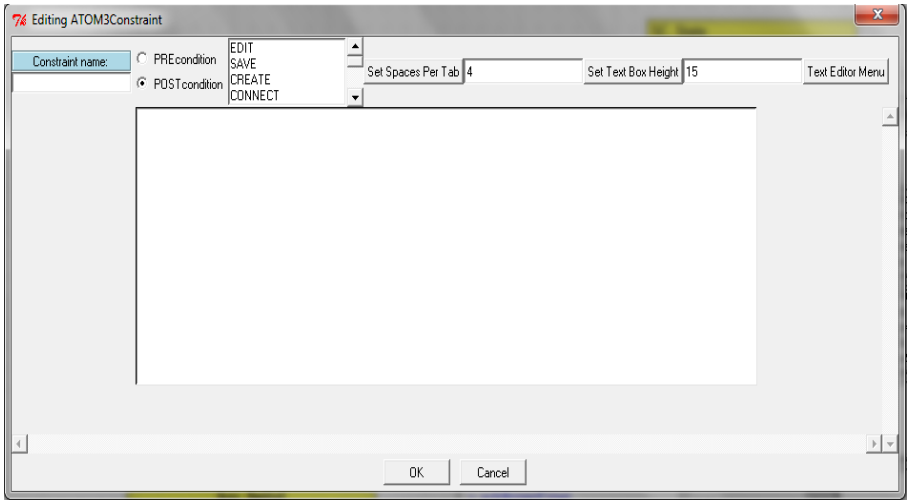

Figure 2.8 : Structure d'une contrainte.

Une action est similaire à une contrainte sauf qu'elle a d'autres effets et elle est un code en Python seulement.Une action est composée de :

- Un nom d'action.
- Un événement déclenchant l'action, l'événement peut être sémantique tel que la sauvegarde d'un modèle, création d'une entité ou graphique ou structurel tel que le déplacement ou la sélection d'une entité.
- Une spécification quand le code de l'action doit être exécuté, avant l'événement (pré-condition) ou après (post-condition).
- Une zone pour écrire un code Python[8].

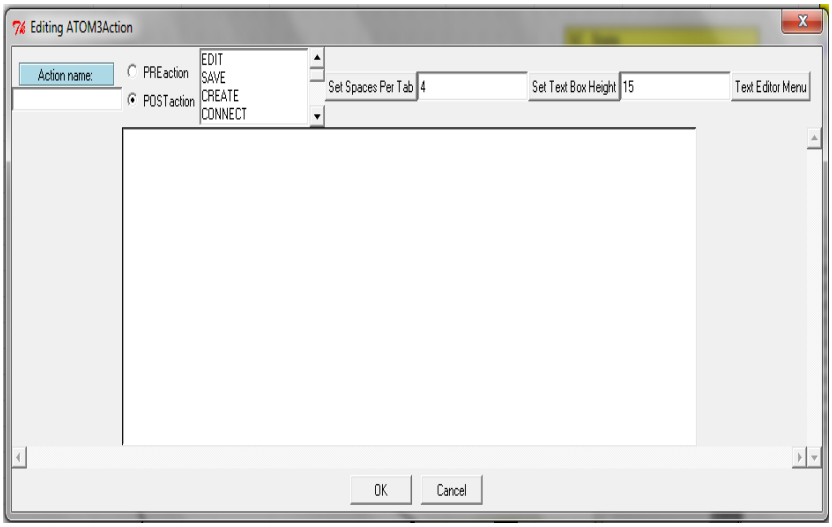

Figure 2.9 : Structure d'une action.

7.2.3. Transformation de graphes

Dans ATOM[3], une fois un modèle est chargé il peut être transformé à un autre modèle équivalent, exprimé par un autre formalisme ou dans le même formalisme.

Dès que les modèles sont représentés à l'intérieur sous forme de graphes alors les transformations entre modèle s'effectuent par des grammaires de graphes.

Une grammaire de graphes est un ensemble de règles, avec une action initiale et une action finale. Elle a un nom et un ensemble de menus permettant sa manipulation, tel que la sauvegarde, l'exécution…etc. comme le montre la Figure suivante :

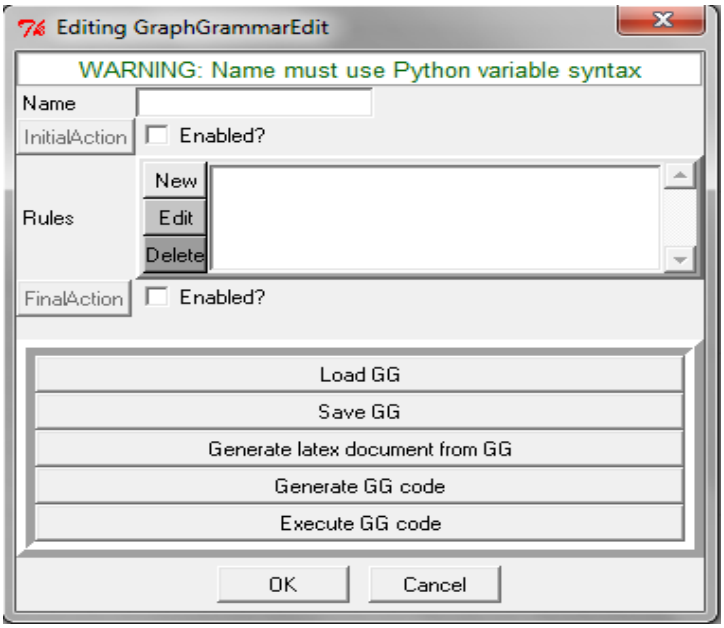

Figure 2.10 : Structure d'une grammaire.

Chaque règle est constituée de:

- Un nom spécifique pour la règle.
- Une priorité indiquant l'ordre dans lequel la règle est appliquée.
- Une partie gauche (Left Hand Side : LHS) qui est un graphe.
- Une partie droite (Right Hand Side : RHS) qui peut être un graphe.
- Une condition (Un code en Python) qui doit être vérifiée avant que la règle soit appliquée.
- Une action (Un code en Python) qui doit être exécutée une fois que la règle est appliquée.

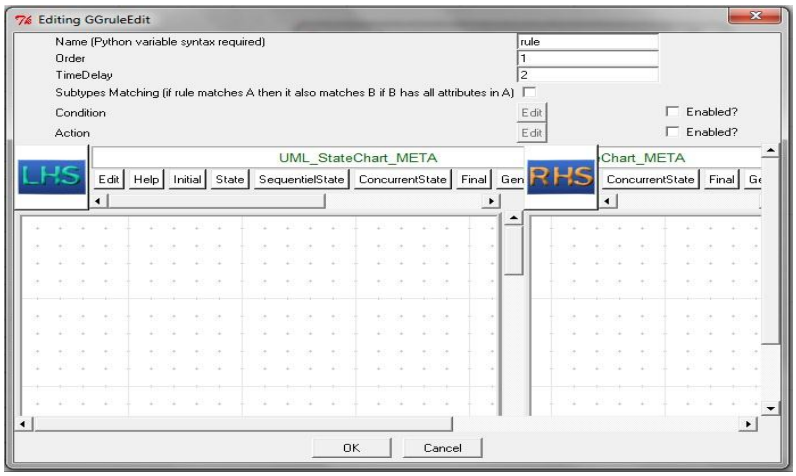

Figure 2.11 Structure d'une règle.

Les deux parties gauche et droite peuvent être des graphes illustrés par des formalismes différents, et pour faire la transformation il faut charger les méta-modèles qui concernent ces formalismes.

L'exécution de modèles peut être continue (aucune interaction d'utilisateur) ou étape par étape ou le client doit intervenir après chaque exécution d'une règle, ou d'une manière artificielle [8].

7.2.4. Langage Python

Python, est un langage autorisant la programmation impérative et la programmation orientée objet, inspiré du langage C, portable sur la plupart des systèmes d'exploitation (Unix, Mac Os, Windows, ..), placé sous la licence BSD (pour Berkeley Software Distribution) et disponible gratuitement, aussi bien utilisable pour écrire des programmes de quelques lignes que des projets très complexes [21].

La syntaxe de Python est très simple et combinée à des types de données évolués (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.

Python est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles. Il intègre, un système d'exceptions, qui permettent de simplifier considérablement la gestion des erreurs.

Un programme écrit en Python est aussi bien interprète que compilé. Les programmes sources seront écrits dans des fichiers dont le nom aura l'extension ''**.py''**, et seront exécutés par la commande **Python nom_prograamme.py**, lancé dans un terminal [21].

Notre utilisation de ce langage est limitée à certaines instructions, mais la plupart des autres instructions sont propres à ATOM^3, mais sur la base de ce langage.

8. Conclusion

Dans ce chapitre, nous avons présenté quelques concepts de l'architecture dirigée par les modèles (MDA), ainsi que les outils utilisée pour la modélisation atom3.
Le prochain chapitre présente les techniques de vérifications des systèmes embarquées.

CHAPITRE 3

Vérifications des systèmes embarquées

Chapitre 3

Vérifications des systèmes embarquées

1. Introduction

Dans ce chapitre nous avons présenté les techniques de vérifications des systèmes embarquées, nous allons représenter les propriétés temporelles du diagramme, nous allons aussi générer le code Maude en utilisent un outil de modélisation et de génération (transformation diagramme-code). Après obtenir le code Maude nous allons vérifier la validité temporelle du code en utilisent un outil de vérification automatique (Model-Checker). A la fin du chapitre nous allons obtenir les résultats de vérification.

2. Logique de Réécriture

1.1. Présentation

La logique de réécriture est introduite par José Méseguar [1] comme une séquence de travaux sur des logiques générales. Cette logique est un cadre sémantique pour spécifier des systèmes concurrents et des langages. C'est également un cadre logique pour présenter et exécuter différentes logiques et langages [10].

La logique en général est une méthode de raisonnement correcte pour quelques classes d'entités [3]. La logique de réécriture permet de raisonner d'une manière aussi correcte sur les systèmes concurrents non-déterministes ayant des états et qui changent d'états à travers des transitions. Cette logique, qui englobe plusieurs modèles formels qui exprime la concurrence, permet d'expliquer n'importe quel comportement concurrent dans un système comportant une sémantique de vraie concurrence.

Formellement, la logique de réécriture est représentée par une théorie de réécriture ***R***= (Σ, E, L, R) où **Σ** est un ensemble d'opérateurs (sorte, symboles de fonction), chaqu'un est associé à un nombre Naturel. E est un ensemble de **Σ**-équations. La signature (Σ, E) est une théorie équationnelle qui décrit la structure algébrique des états du système. L est un ensemble d'étiquettes et R est un ensemble défini par $R \subseteq L \times (T_{\Sigma,E}(X))^2$ qui est un ensemble de pair tel que son premier composant est une étiquette et le deuxième est une paire de classes d'équivalence des termes avec $X = \{x_1, x_2, \dots, x_n\}$ est un ensemble nombrable et fini de variables x_i.

Les éléments de R sont des règles des réécritures conditionnelles et

inconditionnelles, une règle de forme $(r, ([t], [t']))$ est une séquence étiquetée notée par $r : [t] \rightarrow [t']$. Pour dire que $\{x_1, x_2, \dots, x_n\}$ est un ensemble de variables en$[t]$ et/ou $[t']$. On écrit $r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)]$, où en abréviation $r : [t(\bar{x})] \longrightarrow [t'(\bar{x})]$ [10].

Les règles de réécriture $[t] \rightarrow [t']$ dans R ne sont pas des équations car au niveau du calcul, elles sont interprétées comme des règles de transition locales dans un système concurrent et logiquement elles sont interprétées comme des règles d'inférence [15].

Pour une théorie $\mathfrak{R}$, nous disons que la séquence $[t] \rightarrow [t']$ est prouvable dans $\mathfrak{R}$ et nous écrivons $\mathfrak{R} \vdash [t] \rightarrow [t']$Si seulement si $[t] \rightarrow [t']$ peut être obtenu par l'application finie des règles de déduction suivantes :

- **Réflexivité** : pour chaque $[t] \in T_{\Sigma,E}(X)$, $\overline{[t] \longrightarrow [t']}$

- **congruence** : pour chaque $f \in \sum_n n \in N$, $\frac{[t_1] \longrightarrow [t'_1] \dots [t_n] \longrightarrow [t'_n]}{[f(t_1,\dots,t_n)] \longrightarrow [f(t'_1,\dots,t'_n)]}$

-**Remplacement** : pour chaque règle de réécriture : $[t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)]$ dans R, $\frac{[w_1] \longrightarrow [w'_1] \dots [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\bar{x})] \longrightarrow [t'(\overline{w'}/\bar{x})]}$ tel que $t(\overline{w}/\bar{x})$ indique la substitution simultanée des w_i pour x_i dans t.

-**Transitivité** : $\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$

La logique équationnelle (modulo un ensemble d'axiome E) est obtenue à partir de la logique de réécriture par l'ajout de la règle suivante [10]:

-**Symétrie** : $\frac{[t_1] \longrightarrow [t_2]}{[t_2] \longrightarrow [t_1]}$.

Avec cette règle, les séquences dérivables dans la logique équationnelle sont bidirectionnelles. Ainsi, nous pouvons utiliser cette notation $[t] \leftrightarrow [t']$ et nous appelons cette séquence bidirectionnelle une équation. [10]

Les unités de base de spécification dans la logique de réécriture sont appelées des théories qui sont de deux types : théories fonctionnelles et théories systèmes.

1.2. Les théories fonctionnelles

sont des théories dans la logique équationnelle d'appartenance appelée MEL (MemberShip Equational Logics) qui est équivalente à la logique de Horn. Les séquences en MEL sont des égalités $t = t'$ ou bien des assertions d'appartenance de la forme $t: s$ exprime que t est de sorte s. Une telle logique étend la logique équationnelle des sortes ordonnées (order-sorted) et contient ainsi les sortes, les relations sous-sorte, la surcharge des opérateurs, et la définition des opérations.

1.3. Les théories systèmes

Quant à elles spécifient le modèle initial $\mathfrak{R}0$ d'une théorie deréécriture $\mathfrak{R}$, ce modèle initial est un système de transition dont :

- les états sont des classes d'équivalence $[t]$ des termes t modulo les équations E définies dans $\mathfrak{R}$,
- les transitions sont des $\alpha - prouves$ (-preuve) de la forme : $[t] \leftrightarrow [t']$.

Dans la suite nous détaillons les différents concepts utilisés dans les théories équationnelles et systèmes.

1.4. Concepts de bases des théories

Dans cette section, nous allons présenter les concepts de base de la logique de réécriture qui sont implémenté dans des langages de réécriture. L'exemple qui se trouvent dans cette section sont écris en Maude qui sera détaillé juste après. Les éléments de base de la logique de réécriture sont [10] :

1.4.1. Sortes

les sortes ou les types sont les premières choses à définir dans une spécification, ce sont partiellement ordonnées via une relation de sous-sorte. Une sorte est déclarée en utilisant le mot sort suivi d'un identificateur (le nom de sorte) suivi d'un espace et un point :

Sort <Sort_Name>.

Les sortes multiples sont déclarées en utilisant le mot-clé Sorts :

Sorts <Sort_Name1> …. <Sort_Name N>.

1.4.2. Sous-sortes

la relation de sous-sorte sur les sortes est équivalente à une relation de sous ensemble (relation d'inclusion). Les sous-sortes sont déclarées en utilisant le mot-clé **subsort** ou **subsorts**. La déclaration est sous forme :

Subsorts <Sort1> < <Sort2>.

Cette déclaration signifiée que <Sort1> est une sous-sorte de <Sort2>.nous pouvons aussi déclarer plusieurs relations de sous sorte en utilisant Subsorts :

Subsorts <Sort1> … <Sort2> < … < <Sortn>

1.4.3. Opérations

les opérations sont déclarées à l'aide du mot-clé **op** suivi de nom d'opération, suivi par deux points, suivi d'une série de sortes qui constituent ses arguments (arité), suivi de ⟶, suivi des sortes des résultats (co-arité), optionnellement suivis de la déclaration d'attributs, suivi d'un espace et point. Le schéma général a la forme suivante :

Op <OpName> : <sort0> ... <sortk> ⟶ <sort> [<OperatorAttributes>].

Un exemple de déclaration des opérations est le suivant [17] :

Op zero : ⟶ Zero .

Op _+_ _*_ : Nat Nat ⟶ Nat.

En cas ou l'arité de l'argument est vide, l'opération est appelée une constante comme c'est le cas pour zéro.

1.4.4. Surcharge d'opérateurs

dans la logique de réécriture, les opérateurs peuvent être surchargés, c'est-à-dire que nous pouvons avoir plusieurs déclarations d'opérations pour le même opérateur avec différents arités et co-arités. Voici un exemple sur la surcharge d'opérateur dans la logique de réécriture:

Op _+_ : NzNat Nat ⟶ NzNat.

Sort Nat3.

Ops 0 1 2 : ⟶ Nat3.

Op _+_ : Nat3 Nat3 ⟶ Nat3.

L'opérateur _+_ est surchargé et à trois déclarations (une dans l'exemple cité auparavant). Cependant, il y a deux types de surcharge dans l'exemple. La déclaration de _+_ avec le premier argument NzNat est un exemple de la surcharge de subsort où les deux opérateurs sont sensés avoir le même comportement.

1.4.5. Variables

dans la logique de réécriture, les variables sont déclarés à tout moment avec une syntaxe comportant un identificateur (le nom de la variable), deux points et un autre identificateur (sa sorte).

Par exemple **A: Nat** déclare une variable nommée **A** de sorte **Nat**. Des variables de la même sorte peuvent être déclarées en utilisant le mot-clé vars :

Vars M N : Nat.

1.4.6. Théories Fonctionnelles

À travers les théories fonctionnelles, on définit les sortes de données et les opérateurs sur ces données. Les sortes de données se composent des éléments qui sont utilisés par les termes. Par définition, deux termes dénotent le même élément si et seulement si ils appartiennent à la même classe d'équivalence déterminée par les équations.

La déclaration d'une théorie fonctionnelle suit la syntaxe suivante :

fmod NOM_TF is

. . . . . .

Endfm

La théorie fonctionnelle NOM_TF est déclarée par le mot fmod suivi par le nom de la théorie suivi par **is**. Les points représentent la déclaration et les expressions qui peuvent apparaître dans la théorie fonctionnelle. Les expressions incluent des axiomes équationnels et d'appartenance.

1.4.7. Equation inconditionnelles

La déclaration des équations dans une spécification à base de logique de réécriture se fait à l'aide du mot-clé **eq**. Suivi par deux termes séparé par le signe d'égalité (=). Et optionnellement, après le deuxième terme par des attributs et à la fin un espace et un point. La syntaxe générale de cette déclaration est la suivante :

eq <Terme1> = <Terme2> [<StatementAttributes>].

Les termes **Terme1** et **Terme2** dans l'équation doivent être de même sorte. Chaque variable qui apparaît dans la partie gauche doit apparaître aussi dans la partie droite.

1.4.8. Axiomes d'appartenance inconditionnelle

les axiomes d'appartenance inconditionnelle (unconditional memberships) sont déclarés avec le mot-clé **mb** suivi par un terme. Suivi par deux points, suivi par une sorte, suivi par un point. Les axiomes d'appartenance peuvent aussi optionnellement avoir des attribues désignés par <statements> [03].

mb <Terme> : <Sort> [<StatementAttributes>].

1.4.9. Equation et Axiomes d'appartenance conditionnelle

Les conditions dans les équations conditionnelles et d'appartenance se composent de différentes équations **T=T0** et d'appartenance **T : S**. une condition peut être une équation simple, une appartenance simple, ou une conjonction d'équation et d'axiomes d'appartenance en utilisant la conjonction binaires de liaison **/ ** qui est associative.

Ainsi la forme générale des équations conditionnelles (ceq) et des axiomes d'appartenance (cmb) est la suivante :

ceq <term-1> = <term-2>

if <EqCondition-1> / \ … / \ <EqCondition-2> [<StatementAttributes>] .

cmb <term> : sort >

if <EqCondition-1> / \ … / \ <EqCondition-2> [<StatementAttributes>] .

La syntaxe concrète de la condition équationnelle **" EqCondition"** à trois formes:

- équation ordinaire t=t'
- équation d'affectation t := t'
- équation booléennes dites abrégées de la forme t, avec t un terme dans la sorte Bool abrégeant l'équation t= true.

Ainsi, ces conditions peuvent apparaitre dans une équation :

(N = = zero) = true

(M =/= s zero) = true

(N > zero or M =/= s zero) = true

1.4.10. Confluence et terminaison des équations

La logique de réécriture ne spécifie pas dans quel ordre les équations dans une théorie seront appliquées en calculant la forme normale d'un terme. Par conséquent les équations doivent être confluentes et mener à une terminaison pour plus de détails veuillez consulter.

1.4.11. Attributs

La logique de réécriture permet de renforcer la déclaration des opérateurs pas des attributs qui fournissent des informations additionnelles à l'opérateur; des informations sémantique, syntaxique, pragmatique …etc. les attributs sont déclarés dans une seule paire de crochets après la sorte du résultat et avant le point [03].

1.4.12. Théories systèmes

Une théorie système spécifie une théorie de réécriture dans la logique de réécriture. Une théorie de réécriture a des sortes, des opérateurs, et peut avoir des équations, des axiomes d'appartenance et des règles de réécriture qui peuvent être conditionnelles. Par conséquent, une théorie de réécriture n'est en effet qu'une théorie équationnelle fondamentale, contenant des équations et des axiomes d'appartenance, plus les règles de réécriture.

Une théorie système est déclarée comme suit :

mod <modulename> is <declaration and statements> endm

par exemple :

mod refiner is

….

endm

Dont les points correspondent à toutes les déclarations et expressions dans une théorie :

- Importation des modules
- Sortes et sous sortes
- Opérations
- Variables
- Équations et axiomes d'appartenances (conditionnelles et non conditionnelles).
- Règles de réécriture (conditionnelles et non conditionnelles).

Nous remarquons que les éléments dans une théorie système sont les même que celles de théorie équationnelle à l'exception des règles de réécriture.

1.4.13. Règle inconditionnelles de réécriture

Dans la logique de réécriture, le comportement dynamique est modélisé par des règles de réécriture. Une règle de réécriture peut être vue comme une équation à sens unique, et de ce fait, la logique de réécriture est une logique équationnelle sans symétrie [03].

Mathématiquement, une règle inconditionnelle de réécriture à la forme L : $t \rightarrow t'$ Où t et t' sont des termes de même type et qui peuvent avoir des variables, et L est l'étiquette de la règle. Intuitivement, une règle de ce type décrit une transition concurrente locale dans un système. La déclaration d'une règle inconditionnelle suit la syntaxe suivante :

rl [<label>] : <Term-1> => <Term-2> [<StatementAttributes>] .

1.4.14. Règle conditionnelles de réécriture

Les règles conditionnelles de réécriture peuvent avoir des conditions très générales impliquant dans des équations, des axiomes d'appartenances et d'autres réécritures. Mathématiquement, les règles conditionnelles de réécriture ont la forme suivante:[18]

$$l : t \rightarrow t' \quad if \quad (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

La représentation des règles de réécriture conditionnelle en logique de réécriture suit la syntaxe suivante:

crl [<etiquette>] : <Term-1> => <Term-2> if <condition-1 /\ .. /\ condition-2> [<StatementAttributes>].

La condition peut se composer d'une expression simple ou peut être une conjonction avec le connectif associatif **/\.**

3. Langage Maude

3.1 Présentation

Maude [03] est un langage formel de spécification et de programmation déclarative basé sur une théorie mathématique de la logique de réécriture. La logique de réécriture et le langage Maude sont développés par Jose Meseguar et son groupe dans le laboratoire d'informatique en SRI International. Maude est un langage simple expressif et performant, il est considéré parmi les meilleurs langages dans le domaine de spécification algébrique et la modélisation des systèmes concurrents.

Maude spécifie des théories de la logique de réécriture, les types de données sont définies algébriquement par des équations et le comportement dynamique du système est définie par des règles de réécritures qui décrivent comment une partie d'un état est changé dans un pas. Maude supporte aussi la programmation orientée objet avec l'inclusion de l'héritage multiple et la communication asynchrone par le passage des messages [03].

L'interpréteur Maude exécute les programmes équationnels écrit en Maude en commençant par une expression initiale. Ensuite on appliquant les équations " de gauche à droite " jusqu'à ce qu'aucune équation ne puisse être appliquée. L'interpréteur exécute des programmes de réécriture par l'application "arbitraire" des règles de réécriture (aussi de gauche à droite) sur l'expression ou l'état initial jusqu'à ce qu'aucune règle ne soit applicable. Ou à un nombre de réécriture donné par l'utilisateur. Dans le cas des réécritures, les équations sont appliquées pour réduire chaque état intermédiaire à sa forme normale avant d'appliquer les règles de réécriture.

Le groupe de Maude ont focalisé leurs efforts aussi sur la performance, la version actuelle de l'interpréteur peut atteindre des millions de réécriture par seconde. Par ce fait, Maude est en concurrence avec les langages de haut niveau en termes d'efficience [10].

Une théorie de réécriture est généralement non déterministe et peut exhiber différent comportements. La commande **rewrite** du Maude peut être utilisée pour exécuter un seule comportement à partir d'un état initial. Pour analyser tous les comportements possibles à partir d'un état initial, on peut utiliser la commande **search** de haut performance et/ou le modèle checker de Maude qui est comparable en termes d'efficience avec d'autres modèle-checker. Les spécifications Maude peuvent être aussi analysées par le calcul méta-niveau du Maude [03].

Maude contient aussi un support d'utilisation des sockets pour la programmation réseau, ce qui permet non seulement de modéliser, de simuler et d'analyser des systèmes concurrents, mais aussi de les programmés [10].

Pour spécifier un système concurrent, Maude offre trois types de modules :

- Modules fonctionnelles
- Modules systèmes
- Modules orienté-objet, ce sont définis uniquement dans full Maude.

3.2 **Modules Fonctionnelles**

Les modules fonctionnels sont des théories fonctionnelles Que nous avons présentées précédemment. Ces modules définissent les types de données et les opérations qui sont

utilisés par les équations.

L'algèbre initiale sous-jacente est un modèle mathématique dénotationel pour les sortes et les opérations. Les éléments de cette algèbre sont des classes d'équivalence des termes sans variables (grounds termes [01]) modulo les équations. Si deux termes sans variables sont égaux par une équation, on dit qu'ils appartiennent à la même classe d'équivalence. Les équations0 sont utilisées comme des règles de réductions. À la fin du calcul, chaque règles est évaluée à sa forme réduite dite représentation canonique [10]. La représentation canonique est unique et elle représente tous les termes de la même classe d'équivalence.

Les équations dans un module fonctionnel sont orientées, elles sont utilisées de gauche à droite, le résultat de réduction est unique comme nous avons dit quelques soit l'ordre dans lequel les équations sont appliquées.

Les modules fonctionnels supportent aussi les axiomes d'appartenance (membership axioms) ces axiomes précisent l'appartenance d'un terme à un type. Les axiomes peuvent être conditionnels ou inconditionnels. En cas des axiomes conditionnelles, les conditions sont des jonctions des équations et des tests d'appartenance inconditionnels [03]

En Maude, une spécification équationnelle est un module fonctionnel qui est représenté par la syntaxe suivante : **fmod MODULENAME is**
BODY
endfm.

Où MODULENAME est le nom de module introduit, et BODY est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance et des commentaires. Les commentaire commencent par *** ou ---- et se termine par la fin de la ligne courante ou elles commencent par ***(ou ---(et se termine par l'occurrence de ")".

Un exemple d'un module fonctionnel des nombres naturels est le suivant [10] :

```
fmod BASIC-NAT is

sort Nat .                                          *** une sorte

op 0 : -> Nat [ctor].                               *** une opération
op s_ : Nat -> Nat [ctor] .

op _+_ : Nat Nat -> Nat .

op max : Nat Nat -> Nat .

vars N M : Nat.                                     *** deux variables
```

```
eq 0 + N = N.                                          *** une équation

eq s M + N = s (M + N) .

eq max (0, M) = M .

eq max (N, 0) = N .

eq max(s N, s M) = s max(N, M) . endfm
```

3.3 Modules Systèmes

Les modules systèmes sont des théories de réécriture [03]. Ils permettent de spécifier le comportement d'un système concurrent. Les modules systèmes ajoutent à la définition des modules fonctionnels les règles de réécriture qui peuvent être conditionnels et inconditionnels. L'introduction des règles de réécriture permet d'exprimer la concurrence dans les systèmes.

Un module système décrive une théorie de réécriture qui inclue des sortes, des opérations, des variables, des équations, des axiomes d'appartenances (conditionnelles et inconditionnelles) et des règles de réécriture conditionnelles et inconditionnelles.

Une règle de réécriture s'exécute quand sa partie gauche correspond (match) une portion dans l'état global du système et avec la satisfaction de la condition en cas d'une règle conditionnelle [10].

En Maude, une spécification est un module fonctionnel qui est représenté par la syntaxe suivante :

```
mod MODULENAME is

BODY

endfm.
```

Où **MODULENAME** est le nom de module introduit, et **BODY** est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance, et des règles de réécriture. Un exemple qui montre l'utilisation de ces modules est le suivant [03]:

```
mod CHOICE-INT is
including INT .

op _?_ : Int Int -> Int . vars I J :
Int .

rl [choose_first] : I ? J => I . rl
[choose_second] : I ? J => J . endm
```

3.4 Spécifications Orientées Objet en Maude

En effet, la plupart des spécifications réelles dans Maude sont des spécifications orientées objets [03]sont des exemples concrets de spécification des systèmes. Les objets concurrents peuvent être représentés naturellement et directement sur le Core- Maude par le bais des modules systèmes et fonctionnels. Néanmoins, le groupe qui à développer Maude ont étendu le Core-Maude au full Maude qui est dédié à la spécification orientée objet.

Le full Maude est un prototype Maude qui supporte la spécification orientée objets. Il est spécifié est programmé par Francisco Durân. Le Full Maude donne un support pour la spécification orientée objets se forme des modules orientées objets. Ces derniers offrent la syntaxe nécessaire pour déclarer les classes, les sous classes et les messages.

Afin, d'augmenter la vitesse des réécritures, le Full Maude permet de cacher les attributs de classes qui n'influent pas dans l'exécution d'une règle de réécriture, et qui ne sont pas effectués par l'exécution de la règle. L'exécution du full Maude est faite par la translation des modules orientés objet aux modules Maude ordinaires (Core Maude).

3.5 **Modules Orientés Objets**

Les modules orientés objets en Maude sont déclarés avec la syntaxe suivante :

(omod ModuleName is
…..
endom)

Les modules systèmes et fonctionnels peuvent être utilisés en Full Maude. Il faut juste les mètre entre deux parenthèses ( ).

3.5.1 Configuration

L'état concurrent d'un système orienté objet est appelé une configuration. une configuration en logique de réécriture est vue comme un multi-ensemble (multi-sets) qui est composé d'objets et de messages. La configuration à la forme suivante :

_ _ : Configuration Configuration -> Configuration.

Les opérateurs _ _ est déclarés pour satisfaire les lois structurelles d'associativité et de commutativité et d'élément d'identité. Les objets et les messages dans un multi-ensemble sont des sous sorte de la configuration.

Object Message < Configuration.

Les configurations les plus complexes sont générées de l'union de ces multi-ensembles.

Les sortes **Oid**, **Object**, **Msg** et **Configuration** sont définies dans le module Configuration qui se trouve dans le module prelude.maude qui est automatiquement importé dans les modules orienté objet .

```
mod  CONFIGURATION  is
 sorts  Attribute  AttributeSet .
 subsort  Attribute < AttributeSet .
op none : -> AttributeSet .
op _,_ : AttributeSet  AttributeSet -> AttributeSet
[format  (o m so o)  ctor  assoc  comm  id: none] .
sorts  Oid  Cid  Object  Msg  Portal  Configuration .
subsort  Object  Msg  Portal  <  Configuration .
op <_:_|_> :  Oid  Cid  AttributeSet  ->  Object
[ ctor  object  format  (b  r  b  g  b  o  b  o)] .
o.p none : -> Configuration .
op __ : Configuration  Configuration -> Configuration
[format  (o n o)  ctor  config  assoc  comm  id: none] .
Op  < > :  ->  Portal [ctor] .
Endm
```

La sorte **Cid** dénote l'identificateur d'une classe et la sorte **AttributeSet** représente un multi-ensemble d'attributs. [03]

3.5.2 Règles de réécriture pour l'objet

L'associativité et la commutativité d'une configuration multi-ensembles rend la dernière plus flexible. Elle est vue comme une soupe dans laquelle les objets et les messages se flottent. Ces derniers peuvent être ensemble à chaque instant pour participer à une transaction concurrente.

En général, une règle de réécriture dans R qui décrit le dynamique d'un système orienté objet peut avoir la forme :

$$
\begin{aligned}
r : M_1 \dots M_n \; &< O_1 : F_1 \mid atts_1 > \cdots < O_m : F_m \mid atts_m > \\
&\rightarrow < O_{i1} : F'_{i1} \mid atts'_{i1} > \cdots < O_{ik} : F'_{ik} \mid atts'_{ik} > \\
&< Q_1 : D_1 \mid atts''_1 > \cdots < Q_p : D_p \mid atts''_p > \\
&M'_1 \dots M'_q \\
&\Leftarrow C
\end{aligned}
$$

Où : r est l'étiquette de la règle, M_S sont des messages, i_1,…, i_k, sont les différent numéros

des objets, et C c'est la condition. [03]

Un exemple qui illustre l'utilisation des modules orientés objet est le suivant :

```
load full-maude
(omod POPULATION is
 Protecting  NAT .
 protecting  STRING .
 sort  Status .
 op single : -> Status [ctor] .
 ops engaged married separated : Oid -> Status [ctor] .
 subsort String < Oid .
 class Person | age : Nat, status : Status .
 vars N N' : Nat . vars X X' : String .
 crl [birthday] : < X : Person | age : N > => < X : Person | age : N + 1 >
 if N < 999 .
 crl [engagement] : < X : Person | age : N, status : single > < X'
 : Person | age : N', status : single > => < X : Person | status :
engaged(X')
 > < X' : Person | status : engaged(X) > if N > 15 or N' > 15 .
 Op  initState : -> Configuration .        *** etat initial
 Eq  initState =  < "Peter" : Person | age : 37, status : single > <
 "Lizzie" : Person | age : 34, status : single > < "Sam the Snake"
: Person | age : 40, status : single > .
 endom)
```

3.6 Exécution du Maude

L'interpréteur Maude est gratuit. Il est sous la licence de GNU et téléchargeable à partir Site de l'équipe Maude : http://maude.cs.uiuc.edu.

Une session Maude peut être démarré par l'invocation du fichier binaire Maude. Linux dans le répertoire Maude-linux/bin sur le Shell linux. (C'est presque la même chose avec MS-DOS). Nous avons choisi linux Mandriva pour le reste du travail et la figure (*Figure 1.14*) représente une session ouverte de Maude.

Durant une session Maude, l'utilisateur interagir avec l'environnement par la saisie des commandes Maude (Maude prompts). L'utilisateur peut saisir les modules directement en prompt. Mais il est très pratique d'écrire ces modules avec un éditeur de texte et les sauvegarder dans des fichiers. Ces derniers sont appelés du prompt Maude par la commande In ou laod.

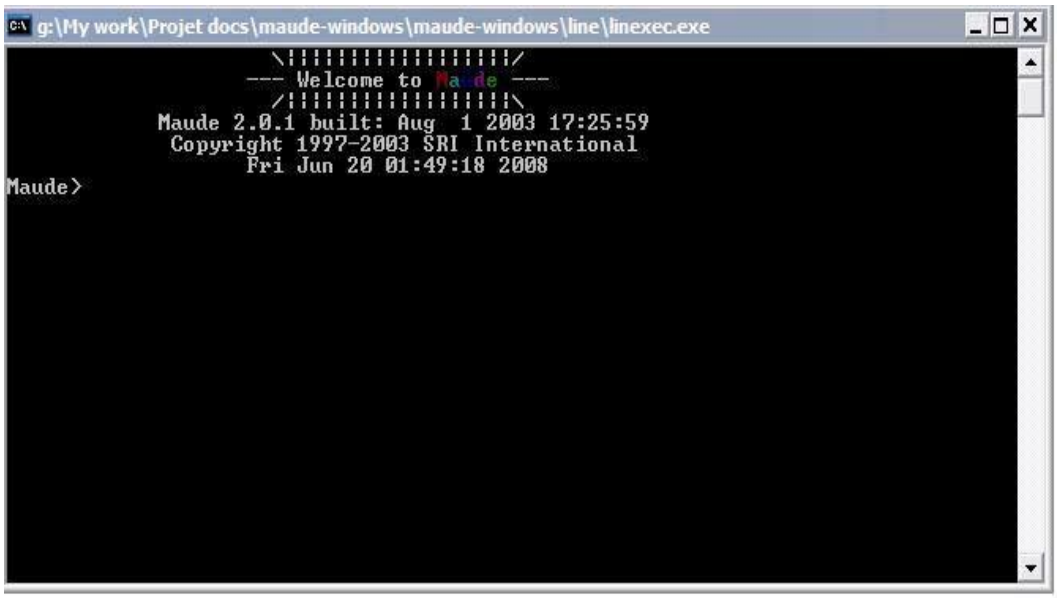

Figure 1.14 : session ouvert de Maude sous MS-DOS.

3.7 Commandes Maude

La liste qui suit présente quelques commandes Maude. En effet Maude à plusieurs autres commandes qui sont décrit en [10].

1. **red** Term.

   La commande demande de réduire le terme "Term" en utilisant les équations dans le module courant.

2. **red** in mod : Term .

   La commande réduire le terme "Term" dans le module "mod"

3. **rew** Term .

   Cette commande permet de réécrire le term "Term" en utilisant les équations les axiomes d'appartenance et les règle de réécriture jusqu'à ce qu'aucune règle ne peut être appliquée.

4. **rew** [n] Term .

   Réécrire le terme Term pour un nombre n de réécriture. (Le nombre des réductions équationnelles n'influent pas sur le nombre n).

5. **frew** Term .

   Pour "fair rewriting", c'est la même que la commande rew à l'exception que les règles sont applicable d'une façon uniforme. Cette commande est souhaitable pour les spécifications orienté objets parc qu'elle assure que chaque objet avoir à chance à se réécrire

6. **search** {[bound]} {in module :} subject searchtype pattern {such that condition}.

La commande search performe une recherche dans le calcul de la commande "rewrite". Cette recherche débute du terme 'subject' jusqu'à un état final qui correspond à "pattern" et qui satisfait une condition optionnelle.

Les valeurs possible de "search tapes' sont :

- => 1, un pas de calcul
- => +, un ou plusieurs pas de calcul
- => *, zero ou plusieurs pas de calcul
- => ! , seulement les états finaux sous forme canonique sont permis.

Le nombre maximal des chemins (solutions) trouvées est borné mais peut-être s'étendre à l'infini. Pour afficher le graphe généré par la commande search il faut utiliser la commande **Show search path.**

7. **select** mod.

   La commande permet de sélectionner le module mod pour qu'il soit le module courant.

8. **set trace on.**

   Demande à Maude d'afficher comment il applique les équations.

9. **show module**.

   Cette commande, permet à Maude d'afficher le module courant.

10. **show sorts**

    Permet d'afficher les sorts du module courant.

4. Les techniques de vérifications

4.1 L'analyse statique

L'analyse statique correspond à l'ensemble des techniques qui permettent de déduire algorithmiquement des propriétés sur le comportement d'un logiciel à partir de l'analyse de son code source et/ou de son code assembleur. Typiquement, l'analyse statique est utilisée lors de la compilation dans le but de détecter des bogues usuels et optimiser le code assembleur obtenu, sans modifier le comportement du programme. Le compilateur va construire un arbre de syntaxe abstraite à partir du code source qui représente toutes les exécutions possibles. Par la suite cette structure est convertie en code assembleur. Le compilateur va vérifier les propriétés sur l'arbre construit. Ce dernier va modifier afin d'avoir un code assembleur optimisé avec une condition qu'aucune de ces modifications modifie le comportement du programme du point de vue de l'utilisateur (le programme d'origine et le programme optimise doivent rester bisimilaires).

L'analyse statique permet de vérifier des propriétés telles que :

- Une variable est réutilisée par la suite ?
- Est-ce que il y a un accès à l'élément p+1 dans un tableau qui contient p éléments ?
- Le contenu d'une variable peut être corrompu par des accès aléatoires à la mémoire ? Le contenu d'une variable peut être corrompu par des accès aléatoires

à la mémoire ?

Néanmoins, cette technique a ses limites. Comme son nom l'indique, il s'agit d'analyse statique et non dynamique. C'est à dire que l'on ne peut vérifier que les variables qui sont initialisées comme des variables statiques dans le programme. Ce qui élimine d'emblées toutes les variables dont l'initialisation est dynamique. Pourquoi ne serait-il pas possible d'appliquer la même analyse aux variables dynamiques ?

Tout simplement parce que l'analyse statique revient à explorer toutes les exécutions possibles et si certaines variables sont dynamiques, le nombre des exécutions possibles est infini. Mais, peut-être existe-t-il un algorithme qui réduise le nombre de ces exécutions infini à un nombre fini ? En fait, il a été prouvé que cela n'est pas possible ; c'est ce qu'on appelle le théorème de Rice : Toute propriété extensionnelle non triviale de programmes écrits dans un langage récursivement énumérable est indécidable.

A l'heure actuelle il existe des résultats qui ordonnent l'analyse statique et l'interprétation abstraite afin d'inclure une partie de ces variables dynamiques dans la vérification [19].

La formalisation dans un langage de type langage de description de matériel n'est pas suffisante pour une approche de preuve formelle. Cette approche, alternative à la simulation, offre des techniques de validation parfaitement fiables, car il s'agit d'utiliser des méthodes "mathématiques" pour vérifier formellement l'adéquation du système à son comportement espéré (ou bien détecter d'éventuelles erreurs). Sans tester le comportement du système sur des jeux d'essais aussi significatifs que possible. Ce raisonnement formel nécessite la transformation de la description VHDL, Verilog,... dans une forme adaptée aux outils qui vont être mis en oeuvre pour la preuve formelle.

4.2 Vérification formelle

Les vérifications basées sur les preuves de théorèmes

- Vérification de modèles (Model-cheking)

Le modèle-checking consiste à vérifier si un modèle d'un système ou une abstraction, satisfait une propriété l'aide d'un traitement de toutes les exécutions possibles en parcourant d'une manière exhaustive le graphe d'état correspondant au système.

Le grand avantage de cette technique est dans le cas où aucune exécution satisfait la spécification vérifiée, elle fournit un contre-exemple qui permet d'analyser le bug éventuel d'où une correction plus rapide. L'analyse de toutes les exécutions possibles nécessite de contrôler chaque état par rapport à la propriété avec la possibilité de retour sur des états précédents pour lesquels il existe des transitions activées mais non explorées.

La vérification de modèles correspond à l'ensemble des techniques qui permettent de déduire algorithmiquement des propriétés à vérifier sur le comportement d'un système (logiciel, algorithme ou protocole) en utilisant des notations prédéfinis comme un modèle (automate fini, automate temporise, réseau de Pétri, algèbre de processus, ...).

La première étape consiste à produire un modèle du logiciel, algorithme ou protocole que l'on veut vérifier. Le formalisme de ce modèle est un système de transition étiquetée. Il convient ensuite de traduire les propriétés que l'on veut vérifier en formules logiques (LTL,CTL, CTL*, TCTL, FOL, ...).

La dernière étape, c'est la vérification proprement dite, elle est totalement la tâche effectuée par la technique de vérification. Celle-ci va ranger le modèle et la formule logique et calculer l'ensemble des états accessibles en avant (post ou forwardanalysis') ou en arrière (pre*ou 'backwardanalysis'). S'il existe un chemin entre l'état initial et l'ensemble des états qui vérifient la formule, alors la propriété est vérifiée.

5. CONTRIBUTION

1. Méta-modèle des Diagrammes d'état-transition temporisé

Ce méta-modèle permet de spécifier les attributs, les contraintes, les relations, ainsi que l'apparence graphique des états et des transitions. Le méta_formalisme utilisé dans cette phase est le diagramme de classe d'UML, et les contraintes sont exprimées en code python. Le méta-modèle proposé est composé de cinq classes reliées par huit associations, comme le montre la figure 3.1.

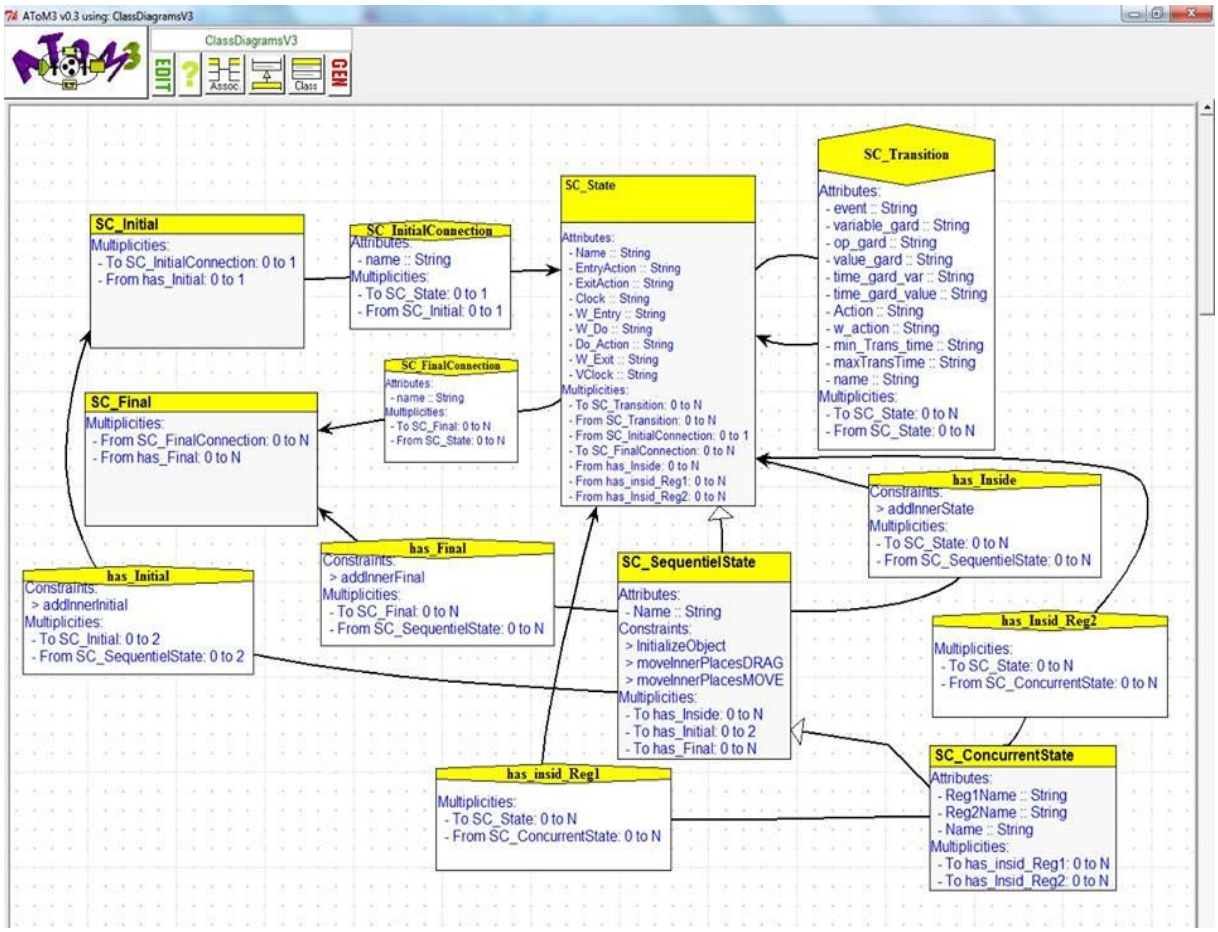

Figure 3.1 : Le méta-modèle des diagrammes d'état de transition temporisé.

2. Exemple Applicatif

L'exemple présenté concerne une montre à cadran numérique avec trois boutons [03], comme le montre la figure 3.20

Figure 3. 20 : Montre à cadran numérique simplifiée.

- Le mode courant est le mode « Affichage ». Quand on appuie une fois sur le bouton mode, la montre passe en « modification heure ». Chaque pression sur le bouton avance incrémente l'heure d'une unité.
- Quand on appuie une nouvelle fois sur le bouton mode, la montre passe en « modification minute ». Chaque pression sur le bouton avance incrémente les minutes d'une unité.
- Quand on appuie une nouvelle fois sur le bouton mode, la montre repasse en mode « Affichage ».
- l'unité de temps : msec .

En même temps il y a un bouton éclairage, en le pressant on éclaire le cadran de la montre, jusqu'à ce qu'on le relâche [03].

Nous sommes clairement en présence de deux comportements concurrents :

- la gestion de l'affichage.
- la gestion de l'éclairage.

Pour cela nous avons proposé un diagramme d'état-transition temporisé pour modéliser le comportement de la montre avec deux régions concurrentes (figure 3.21).

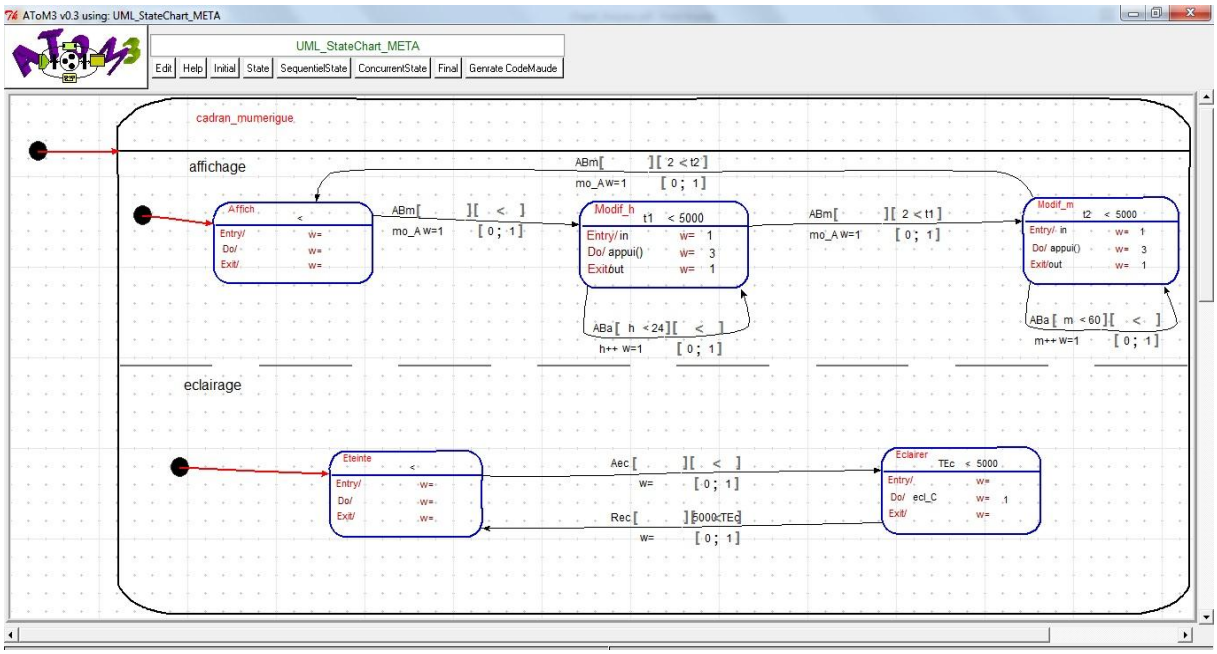

Figure 3.21 : Diagramme d'état-transition temporisé d'une montre a cadre numérique.

Le code suivant est le code Maude généré.

load full-maude

```
( omod GENERIC-SChart is
 protecting NAT .
 sorts SIMSTATE COMSTATE STATE .
 sorts Clock Action Entry Exit DO WCET Time .
 subsorts Entry Exit DO < Action .
 subsorts Time < NAT .
 subsorts Clock WCET < Time .
 subsort SIMSTATE < COMSTATE .
 subsort COMSTATE < STATE .
 var X : NAT.
 var T : Time.
 var S : STATE.
 op none : -> COMSTATE [ctor] .
 op _||_ : COMSTATE COMSTATE -> COMSTATE [ctor assoc comm id: none].
 msg ABm ABa Aec Rec: -> Msg.
 ops mo_A /_ h++/_ m++ /_ : -> Action.
 ops InitState:->SIMSTATE .
 ops Affich Modif_h Modif_m Eclairer Eteinte : Entry Do Exit Clock ->SIMSTATE .
ops cadran_mumerique : COMSTATE-> STATE
 ops in/_  : WCET -> Entry .
 ops appui()/_ecl_c/_ : WCET -> DO .
 ops  out/_  : WCET -> Exit .
r[initT2]:cadran_mumerique(affichage(Initstate)||eclairage(Initstate))=>cadran_mumerique(af
fichage(affich(/;/;/;)||eclairage(Eteinte(/;/;/;).
crl[modif_heure]:cadran_mumerique(affichage(modif_h(in/1;appui()/3;out/1;5000))||eclairage
(s))ABa h++/1 s:h
=>cadran_mumerique(affichage(mdif_h(in/1;appui/3;out/1;5000))||eclairage(s))if
x<24/\min>=0/\max<=1.
crl[modif_minute]:cadran_mumerique(affichage(modif_m(in/1;appui()/3;out/1;5000))||eclaira
ge(s))ABa m++/1 s:m
=>cadran_mumerique(affichage(mdif_m(in/1;appui/3;out/1;5000))||eclairage(s))if
x<60/\min>=0/\max<=1.
```

crl[appui_mode_1]:cadran_mumerique(affichage(affich(/;/;/;))||eclairage(s))ABa mo_A/1 =>cadran_mumerique(affichage(mdif_H(in/1;appui/3;out/1;5000))||eclairage(s))if min>=0/\max<=1.
crl[appui_mode_2]:cadran_mumerique(affichage(modif_m(in/1;appui()/3;out/1;5000))||eclairage(s))ABm mo_A/1
T:t1=>cadran_mumerique(affichage(mdif_m(in/1;appui/3;out/1;5000))||eclairage(s))if T>2/\min>=0/\max<=1.
crl[appui_mode_3]:cadran_mumerique(affichage(modif_m(in/1;appui()/3;out/1;5000))||eclairage(s))ABm mo_A/1 T:t2=>cadran_mumerique(affichage(Affich(/;/;/;))||eclairage(s))if T>2/\min>=0/\max<=1.
crl[relache_eclairage]:cadran_mumerique(affichage(s)||eclairage(Eclairer(/;ecl_c/1;5000)))Rec T:TEc=>cadran_mumerique(affichage(s)||eclairage(Eteinte(/;/;/;)))if T>5000/\min>=0/\max<=1.
crl[appui_eclairage]:cadran_mumerique(affichage(s)||eclairage(Eteinte(/;/;/;)))Aec =>cadran_mumerique(affichage(s)||eclairage(Eclairer(/;ecl_c/1;5000)))if min>=0/\max<=1.
crl[initT0]:InitState =>cadran_mumerique(affichage(InitState)||eclairage(InitState)).

endom)

3. Définition des propriétés à vérifier

- Proposition intial (clock) : cette proposition est évalué à vraie lorsque le clock entre à l'état initial

```
op intial: string -> prop

eq< clock : intial | status :intial, bolocked : false >ccf |=

intial(clock)=true.
```

- Proposition éclairage (clock) : cette proposition est évalué à vraie lorsque le clock entre à l'état éclairer

```
op eclairage: string -> prop

eq< clock : eclairer | status :intial, bolocked : false >ccf |=

eclairage (clock)=true.
```

- Proposition affichage (clock) : cette proposition est évalué à vraie lorsque le clock entre à l'état affiche

op affichage: string -> prop

eq< clock : affichage | status :affiche, bolocked : false >ccf |=

affichage (clock)=true.

- Proposition modification (clock) : cette proposition est évalué à vraie lorsque le clock entre à l'état modifier

op modification : string -> prop

eq< clock : modification | status :modifier, bolocked : false >ccf |=

modification (clock)=true.

4. Les formules logique vérifier :

Les formules logiques que nous allons vérifier sur notre exemple sont :

1. ~[] itial(clock) : cette propriété vérifier que la mante reste jamais à l'état initial
2. ~[] eclairage(clock) :cette propriété vérifier que la mante reste jamais à l'état éclairage
3. ~[] affiche(clock) : cette propriété vérifier que la mante reste jamais à l'état affichage
4. ~[] modifier(clock) : cette propriété vérifier que la mante reste jamais à l'état modification
5. ~[] (itial(clock) /\ eclaire(clock) /\modifier(clock)/\affiche(clock)) : cette propriété permet de vérifier l'exécution à temps préciser
6. ~[] (itial(clock) \/ eclaire(clock) \/modifier(clock)\/affiche(clock)) : cette propriété permet de vérifier la présence de blocage
7. [] (itial(clock) \/ eclaire(clock) \/modifier(clock)\/affiche(clock)) : cette propriété permet de vérifier l'absence d'inter blocage
8. [] (itial(clock) => <> eclaire(clock)) : cette propriété garante qu'une clock a été allumer
9. [] (itial(clock) => <> modifier(clock)) : cette propriété garante qu'une clock a été allumer et modifier
10. [] <> ~ modifier(clock) cette propriété vérifier le clock a été modifier

5. Verification du module

Apres l'ouverture d'une session Maude, les commandes suivantes sont introduites afin de vérifier les propriétés définies en dessus.

A. `(red modelCheck(initState, [] ~ ((Entry(Modif_h) + Do(Modif_h) + Exit(Modif_h) < Clock(Modif_h))) .)`

B. `(red modelCheck(initState, [] ~ ((Entry(Modif_m) + Do(Modif_m) + Exit(Modif_m) < Clock(Modif_m))) .)`

C. `(red modelCheck(initState, [] ~ ((red ecl_C(Eclaire) < Clock(Eclaire)) .)`

L'exécution du Model-Checking s'effectue par l'ouverture d'une session Maude et l'introduction du module MCHECK. Puis l'introduction des commandes de vérification citées en dessus. Le résultat de vérification du code est présenté dans la figure suivante.

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.6 built: Mar 31 2011 23:36:02
            Copyright 1997-2010 SRI International
                  Wed May 27 20:56:36 2015
Maude> red modelCheck(initState, [] ~ ((Entry(Modif_h) + Do(Modif_h) + Exit(Modif_h)
< Clock(Modif_h))) .
reduce in MCHECK : modelCheck(initState, [] ~ (Entry(Modif_h) + Do(Modif_h) +
Exit(Modif_h)  < Clock(Modif_h))) .
rewrites: 1003 in 5ms cpu (5ms real) (201 rewrites/second)
result Bool: true

Maude> red modelCheck(initState, [] ~ ((Entry(Modif_m) + Do(Modif_m) + Exit(Modif_m)
< Clock(Modif_m))) .
reduce in MCHECK : modelCheck(initState, [] ~ (Entry(Modif_m) + Do(Modif_m) +
Exit(Modif_m)  < Clock(Modif_m))) .
rewrites: 1006 in 5ms cpu (5ms real) (202 rewrites/second)
result Bool: true

Maude> red modelCheck(initState, [] ~ ((red Do(Eclaire) < Clock(Eclaire)) .
reduce in MCHECK : modelCheck(initState, [] ~ ((red Do(Eclaire) < Clock(Eclaire)) .
rewrites: 1002 in 1ms cpu (1ms real) (1002 rewrites/second)
result Bool: true
```

6. Conclusion

Dans ce chapitre nous avons présenté notre contribution qui consiste à proposer une approche de vérification d'un système embarqué basée sur la transformation des diagrammes d'état-transition et de collaboration temporisés d'UML vers le langage Maude. La méthode proposée se base sur le choix d'une propriété temporelle spécifiée en LTL, et sa vérification avec model-checker . Enfin, nous avons présenté un exemple illustratif de notre travail

Conclusion générale

Conclusion générale

Dans ce travail, nous avons proposé une méthode de vérification d'un système embarqué, modélisé par deux diagrammes d' UML (timed statechart et timed collaboration diagrams), en utilisant la logique de réécriture et le langage Maude. La méthode repose sur la génération à partir des diagrammes UML d'une spécification formelle écrite en Maude dont le comportement du système modélisé est représenté par des règles de réécritures.

Les propriétés à vérifiées dans le système sont des propriétés exprimées en logique temporelle linéaire. Ces propriétés vont être vérifiées vis-à-vis la spécification Maude en utilisant le LTL model-checker du langage Maude.

Dans ce travail , nous avons utilisé l'outil ATOM ^{3} pour l'élaboration des outils de modélisation d'un système embarqué avec les deux diagrammes d'UML. Ensuite une grammaire de transformation est définie pour générer un code Maude depuis ces diagrammes. Pour la vérification de ce système, nous avons choisi une propriété temporelle pour être spécifiée dans la logique temporelle LTL. Ensuite cette spécification est combinée avec le code Maude pour former une entrée au model-checker. Ce dernier répond par une réponse logique 'vrai' ou 'faux', exprimant si le système embarqué assure cette propriété ou non.

Malgré que le travail réalisé nécessite beaucoup connaissances sur la vérification formelle et ses techniques, nous avons abouti à atteindre notre objectif visé, et nous proposons comme une perspective à ce travail, l'automatisation de l'opération de spécification des propriétés temporelles.

Bibliographie

Bibiographie

[1] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theoretical Computer Science, 360(1-3):386–414, 2006.

[2] Jean-Paul CALVEZ, Spécification et conception conjointe des systèmes matériel/logiciel

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework, volume 4350 of Lecture Notes in Computer Science, Springer, 2007.

[4] G. Denker, J. Meseguer, C. Talcott, Protocol Specification and Analysis in Maude, In N. Heintze and J. Wing, editors, Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana, 1998.

[5] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, dans F.Gadducci and U. Montanari, editors, Fourth International Workshop on Rewriting Logic and its Applications, volume 71 of Electronic Notes in Theoretical Computer Science, Elsevier, 2002.

[6] Richard Grisel et Nacer Abouchi, les systèmes embarqués introduction

[7] Patrice KADIONIK, Les Systèmes Embarqués Introduction, © pk/2005 v 2.2

[8] De Lara.J , Vangheluwe.H, AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling, Lecture Notes in Computer Science 2306, pp.174-18, 2002.

[9] Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et ré-parti), La démarche MDA, Mai 2002.

[10] N. Martí-Oliet, J. Meseguer, Rewriting logic: Roadmap and bibliography, Theoretical Computer Science, 285, 2002.

[11] P. C. Ölveczky, Formal Modeling and Analysis of Distributed Systems in Maude, Lecture Notes INF3230/INF4230, Department of Informatics, UNIVERSITY OF OSLO, 16 janvier 2008.

[12] Tewfik ZIADI, Manipulation de Lignes de Produits en UML, France, Décembre 2004.

[13] Boumassata Meriem, Vérification de code pour plates-formes embarqués, Mémoire En vue de l'obtention du diplôme de Magistère en Informatique

[14] Ramzi BOULKROUNE, Les systèmes embarqués, Memoire Online > Informatique et Télécommunications

[15] Cécile HARDEBOLLE, Composition de modèles pour la modélisation multi-paradigme du comportement des systèmes, pour obtenir le grade de Docteur en Sciences, France, Novembre 2008.

[16] Z. MAMMERI, Chapitre 1 Introduction aux systèmes embarqués et temps réel, Cours – Module ASTRE

[17] K. Megzari, REFINER : Environnement logiciel pour le raffinement d'architectures Logicielles fondé sur une logique de réécriture, thèse de doctorat, université de Savoie, France, 2004.

[18] http://atilf.atilf.fr/ , consulté le : 5/4/2014.

[19] mdsd, (http://www.mdsd.info/) , consulté le : 04/2014

[20] atom3,http://atom3.cs.mcgill.ca/ , consulté le : 04/2014

[21] Python, http ://www.python.org/, consulté le : 05/2014

[22] http://www.technologuepro.com

ملخص:

الأنظمة المضمنة هي أنظمة معقدة التي تطرح مشكلة كبيرة في نمذجتها والتحقق من بياناتها هذا العمل يسمح بتطوير نظام نمذجة وتحقق وذلك باستخدام ايامال و مدالتشكر

- **الكلمات المفتاحية:** ايامال،مود، لوجيك تيمبوريل لينير، اتوم3، مدال تشكين.

Résumé :

Les systèmes embarqués sont des systèmescomplexes qui posent le problème lors de leurs modélisations et de vérification

Ce travail permet de développer un système de modélisation et de vérification en utilisent UML et model-checking.

- **Mots clés** : UML, maude, LTL, ATOM3 , model checking.

Abstract:

Embedded systems are complex systems that pose the problem in their modeling and verification this work allows to develop a modeling and verification system using UML and model checking

- **Keywords**: UML, maude, LTL, ATOM^3, model-checking.