Table des matières

Liste des figures

Liste des tableaux

Acronymes

AGG : **A**ttributed **G**raph **G**rammar.

API : **A**pplication **P**rogramming **I**nterface.

ATL : **A**tlas **T**ransformation **L**anguage.

ATM : **A**utomated **T**eller **M**achine.

$AToM^3$ **: A To**ol for **M**ulti-formalism and Meta-Modeling.

CCS : **C**alculus of **C**ommunicating **S**ystems.

CSP : **C**ommunicating **S**equential **P**rocesses.

EMF : **E**clipse **M**odeling **F**ramework.

ETL : **E**psilon **T**ransformation **L**anguage.

GrGen : **Gr**aph Rewrite **Gen**erator.

GROOVE : GRaphs for **O**bject-**O**riented **VE**rification.

IDM : **I**ngénierie **D**irigée par les **M**odèles.

KM3 : **K**ernel **M**eta **M**eta **M**odel.

MDA : **M**odel **D**riven **A**rchitecture.

MOF : **M**eta **O**bject **F**acility.

OMG : **O**bject **M**anagement **G**roup.

PVS : **P**rototype **V**erification **S**System.

QVTr : **Q**uery **V**iew **T**ransformation **R**elations.

RAISE : **R**igorous **A**pproach for **I**ndustrial **S**oftware **E**ngineering.

RdPs : Les **R**éseaux **d**e **P**etri.

SPIN : **S**imple **P**romela for **IN**terpreter.

TG : **T**riple **G**raph **G**rammar.

TGG : **T**riple **G**raph **G**rammar.

TNs : **T**ransformation **N**ets.

UML : **U**nified **M**odeling **L**anguage.

VDM : **V**ienna **D**evelopment **M**ethod.

V&V : La **V**érification et la **V**alidation.

XMI : **X**ML **M**etadata **I**nterchange.

XML : e**X**tended **M**arkup **L**anguage.

Introduction Générale

Depuis l'avènement du génie logiciel, le développement des systèmes devient de plus en plus important dans différents domaines d'application et évolue d'une manière rapide et croissante. Pour faire face à cette situation, l'Object Management Group (OMG) a proposé l'Ingénierie Dirigée par des Modèles (IDM) ou Model-Driven Engineering en anglais (MDE) afin de prendre en charge les problèmes posés dans le contexte du génie logiciel tels que l'optimisation des coûts et du temps de développement des systèmes informatiques. Par conséquent, l'IDM essaye d'apporter une réponse à ces problèmes.

L'IDM se base sur deux aspects principaux : la métamodélisation et la transformation de modèles. Le premier aspect concerne la définition et la manipulation des métamodèles et des modèles pour spécifier les systèmes informatiques à un niveau très abstrait, et le deuxième aspect porte sur la transformation automatique ou semi-automatique des modèles pendant le développement afin de produire les applications informatiques.
Dans le cadre de l'IDM, la transformation de modèles joue un rôle central et très important et se fonde sur un ensemble de formalismes et d'outils afin de permettre la mise en œuvre de la technologie dirigée par les modèles. Ainsi, plusieurs travaux ont porté sur la transformation de modèles et leurs utilisations mais le défi actuel de l'IDM est de répondre aux problèmes de la validation et de la vérification de la transformation de modèles afin d'assurer une bonnes production du modèle final.

La majorité des travaux proposés dans le cadre de la transformation de modèle-à-modèle n'explicitent pas clairement comment sont prises en charge les deux notions de vérification et de validation. L'article de Daniel Calegari dans [20] présente un état de l'art sur la vérification de la transformation de modèles. Il est établi que beaucoup de travaux ne détaillent pas la prise en charge de la validation dans le processus de tranformation.

Nous situons nos travaux dans le cadre de l'intégration de techniques formelles

dans le processus d'IDM où nous nous intéressons à la vérification de la conformité des modèles par rapport à leurs métamodèles et aussi à la validation du processus de transformation. A ce niveau, notre objectif est d'entreprendre une expérimention d'intégration d'un outil formel à travers l'utilisation d'une étude de cas assez complexe afin d'étudier si l'intégration d'outils formels de l'ingénierie du logiciel est appropriée au contexte de l'IDM.

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

- Le **premier chapitre** détaille le contexte de l'ingénierie dirigée par les modèles en présentant les concepts de base ainsi que la métamodélisation et la transformation de modèles.
- Le **deuxième chapitre** est consacré aux méthodes formelles où nous présentons une définition simple de ces méthodes et les différentes les différentes classifications existantes de ces méthodes.
- Le **troisième chapitre** présente l'importance de la vérification et la validation dans la transformation de modèles.
- Le **quatrième chapitre** présente les outils utilisés pour notre approche tant au niveau de l'IDM qu'au niveau de la méthode formelle utilisée pour la vérification et la validation des transformations de modèles.
- Le **cinquième chapitre** détaille un exemple d'étude de cas en mettant l'accent sur les processus de métamodélisation et la transformation de modèles.
- Et le **sixième chapitre** décrit en détail notre approche de formalisation de la vérification avec le langage OCL et de la validation avec l'assistant de preuve Coq.

Et nous concluons ce mémoire par des perspectives qui permettront de donner une suite à cette recherche.

Chapitre 1

Ingénierie Dirigée par les Modèles

1.1 Introduction

Le développement du logiciel fait face à une augmentation de la complexité des systèmes malgré l'utilisation des techniques spécifiques qu'on les appelle modèles de cycles de vie d'un logiciel comme le modèle en cascade, le modèle en V et le modèle en W. A l'instar d'autres sciences, la modélisation est de plus en plus utilisée pour maîtriser cette complexité des systèmes. L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais s'inscrit dans cette évolution en prônant l'utilisation systématique de modèles pour automatiser une partie des processus de développement suivis par les ingénieurs. Donc l'IDM propose de modéliser les applications à un haut niveau d'abstraction où elle se place le modèle au cœur du processus de conception puis génère le code de l'application à partir des modèles. Ainsi, l'IDM est une approche générale et ouverte qui fait suite à la proposition du standard MDA (Model-Driven Architecture) proposé par l'OMG en 2000. Elle a permis plusieurs améliorations significatives dans le développement des logiciels en permettant de se concentrer sur l'utilisation intensive des modèles et leur transformation [10].

Dans ce chapitre, nous allons rappeler quelques concepts de base de l'IDM et ensuite nous décrivons les raisons qui y ont poussé vers la transformation de modèles. Aussi nous développerons les critères de transformation de modèles ainsi que la classification des différentes approches de transformation de modèles existantes.

1.2 Concepts de base de l'IDM

L'ingénierie Dirigée par les Modèles [33] a permis de prendre en charge la croissance de la complexité des systèmes logiciels développés où la modélisation de ces systèmes se base sur l'usage intensif de modèles. De cette manière, le développement est réalisée avec un niveau d'abstraction plus élevé que celui de la programmation classique. Cette approche permet alors d'automatiser, ou au moins de dissocier et de reporter, la part du développement qui est proprement technique et dédiée à une plate-forme d'implémentation.

L'ingénierie dirigée par les modéles (IDM) est donc une approche du génie logiciel sur laquelle le modèle est considéré comme une première présentation du système à modéliser, et qui vise à développer, maintenir et faire évoluer le logiciel en réalisant des transformations de ce modèle. Au sens large, le paradigme de l'IDM propose d'unifier tous les aspects du processus du cycle de vie en utilisant les notions de modèle et de transformation [10].

Dans ce qui suit, nous présentons des définitions [32] pour bien comprendre les concepts de base de l'ingénierie des modèles et les relations existants entre ces concepts.

Système : *un système est un ensemble d'éléments identifiables, interdépendants, c'est-à-dire liés entre eux par des relations telles que, si l'une d'elles est modifiée, les autres le sont aussi et par conséquent tout l'ensemble du système est modifié et transformé. C'est également un ensemble borné dont on définit les limites en fonction des objectifs (propriétés, buts, projets, finalités) que l'on souhaite privilégier [23].*

Pour modéliser un système dans le domaine de l'IDM, il est intéressant de bien comprendre et de bien maîtriser les notions de modèle et métamodèle. Nous retiendrons les définitions suivantes tirées de [11].

Modèle : *Un modèle est une description abstraite d'un système construite dans un but donné. Donc, il doit pouvoir être utilisé pour répondre à des questions sur le système.*

Les modèles doivent en général, respecter des contraintes décrites par un méta-modèle.

Métamodèle : *Un métamodèle est un modèle qui permet d'exprimer un modèle.*

Le métamodèle est l'élément de structuration de modèles. Il permet de définir de façon précise les différents formalismes qui permettent de construire les modèles.

Donc on peut dire qu'un métamodèle est souvent défini comme un modèle d'un modèle. Un modèle n'est utilisable que si son langage de modélisation est précisé. Dans le sens de l'IDM, un modèle doit etre conforme à un métamodèle pour avoir une bonne définition d'un système (cf. Figure 1.1). Cette notion de conformité est très importante qu'on peut définir comme suit :

Conformité : *Elle est définie par un ensemble de contraintes entre un modèle et son métamodèle et qui expriment les relation entre eux. Ces contraintes permettent de générer les modèles d'une façon précise.*

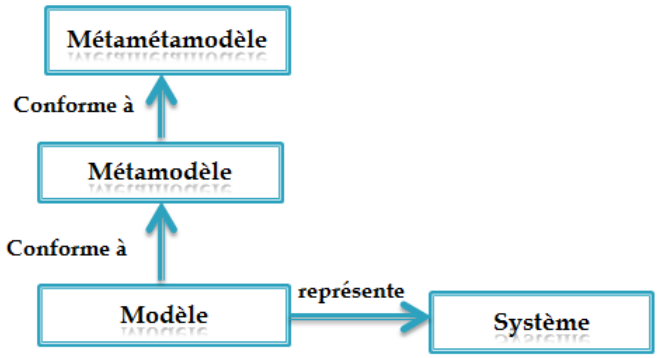

Figure 1.1: Notions de base en ingénierie des modèles

Il est important de rappeler que l'IDM se base sur deux concepts principaux: La métamodélisation et la transformation de modèles (cf. Figure 1.2). Chacun de ces concepts complète l'autre pour pouvoir résoudre les systèmes complexes et critiques. Dans les sections suivantes, nous mettons l'accent sur la transformation de modèles, les critères de classification de transformation de modèles, en plus des différentes approches proposées.

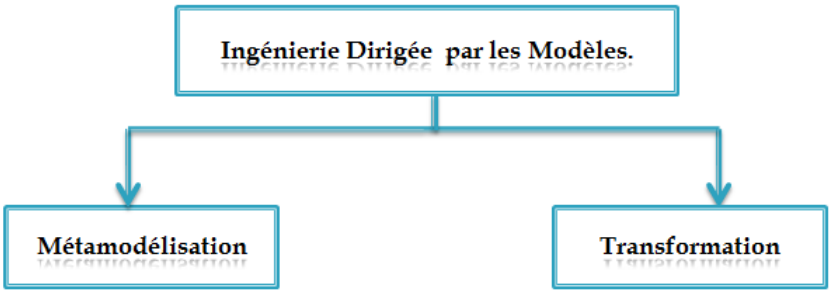

Figure 1.2: Composantes de l'ingénierie dirigée par les modèles

1.2.1 Métamodélisation

Pour manipuler un modèle, il est nécessaire de définir sa structure, sa sémantique et son langage de modélisation. Ces derniers sont représentés par un métamodèle qui illustre une définition formelle d'un modèle. En d'autres termes, le métamodèle exprime les éléments, la structure ainsi que la sémantique de ces modèles. En plus il définit la relation existante entre un modèle et le système à modéliser. La mise en oeuvre de cette relation représente alors la métamodélisation. Dans [52], la métamodélisation est une activité qui consiste à définir des métamodèles contemplatifs qui reflètent la structure statique des modèles.
Dans ce qui suit les notions de système, métamodèle et modèle seront explicités à travers l'exemple suivant qui décrit une pièce ayant 4 murs, 2 fenêres et une porte. Un mur possède une porte ou une fenêtre mais pas les 2 à la fois. Deux actions sont associées à une porte ou une fenêtre : ouvrir et fermer. Si on ouvre une porte ou une fenêtre fermée, elle devient ouverte. Si on ferme une porte ou une fenêtre ouverte, elle devient fermée. Les figures suivantes (1.3, 1.4) présentent le métamodèle et le métamodèle de ce système [21].

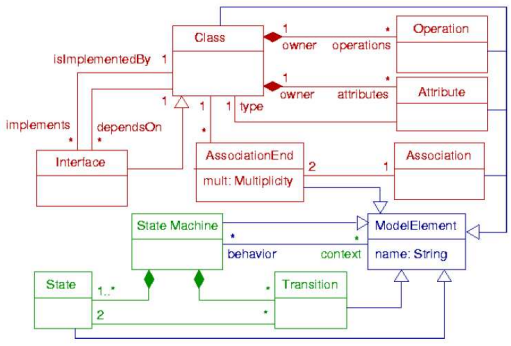

Figure 1.3: Métamodèle

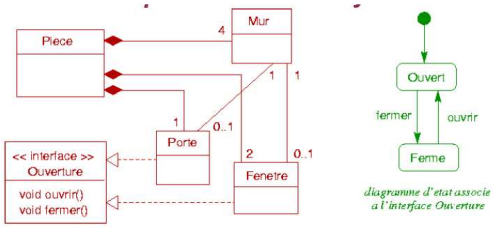

Figure 1.4: Modèle

1.2.2 Transformation de modèles

Dans l'IDM, la transformation de modèles définit la partie cruciale du processus de développement de logiciels. Généralement, elle exprime la phase de translation d'un modèle en un autre type de modèle à l'aide d'un programme de transformation écrit avec un langage dédié. Les langages de transformation de modèles peuvent être classifiés en trois catégories. La première catégorie concerne les langages de transformation déclaratifs comme QVT-Relations (QVTr) [73], Triple Graph Grammar (TGG) [57], et Transformation Nets (TNs) [82]. La deuxième catégorie regroupe les langages impératifs comme Kermeta [55]. Tandis que la troisième catégorie représente les langages hybrides regroupant les caractéristiques des deux précédentes catégories. Les langages les plus illustratifs de cette classe sont le langage ATL (Atlas Transformation Language) [43] et le langage ETL (Epsilon Transformation Language) [59]. Un processus de transformation de modèles se base sur les éléments suivants : la définition des métamodèles et leur modèles et la spécification des règles de transformation. Les métamodèles peuvent être définis avec des langages de métamodélisation comme MOF [71], Ecore [16], KM3 (Kernel MetaMetaModel) [54]. Les modèles sont calculés par une instanciation de leurs métamodèles.
Pour la transformation de modèles, elle se base sur la spécification des règles de correspondance entre les éléments des métamodèles associés. Dans ce qui suit, nous définissons les notions suivantes : la transformation de modèles, la transformation et les règles de transformation selon Kleppe et al [58] et qui sont illustrés dans la figure 1.5.

- **Transformation :** *Une transformation de modèles est une génération automatique d'un modèle cible à partir d'un modèle source selon la définition de la transformation.*
- **Définition de transformation :** *C'est un un ensemble de règles de transformation qui décrivent comment un modèle source peut être transformé en un modèle cible.*
- **Règle de transformation :** *Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un langage source peuvent être transformées en une ou plusieurs constructions dans un langage cible.*

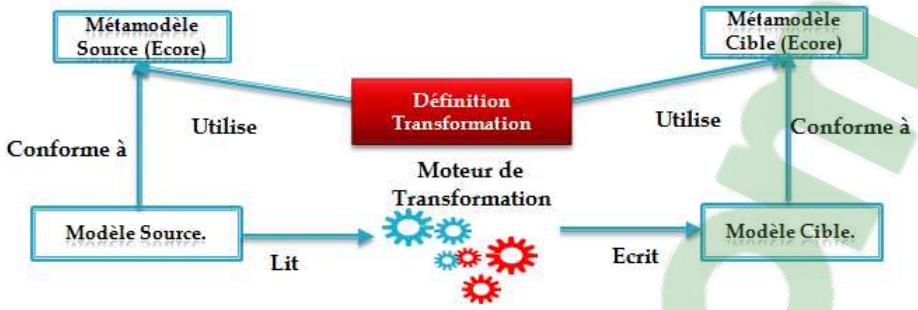

Figure 1.5: Concepts de base des transformations de modèles

La figure 1.5, empruntée à [58], résume une transformation de modèles. Etant donné un modèle Ma conforme à son métamodèle source MMa, un métamodèle cible MMb et une définition de transformation, Le programme de transformation (moteur de transformation) est généré à partir de la définition de la transformation dont l'exécution permet de calculer le modèle cible conforme à son métamodèle cible MMb.

1.3 Pourquoi la transformation de modèles?

La transformation de modèles joue un rôle fondamental dans le développement des logiciels et vise des objectifs divers comme la automatise la génération automatique du code, la translation de modèle, la migration, et l'ingénierie inverse (cf. Figure 1.6) que nous détaillerons dans ce qui suit.

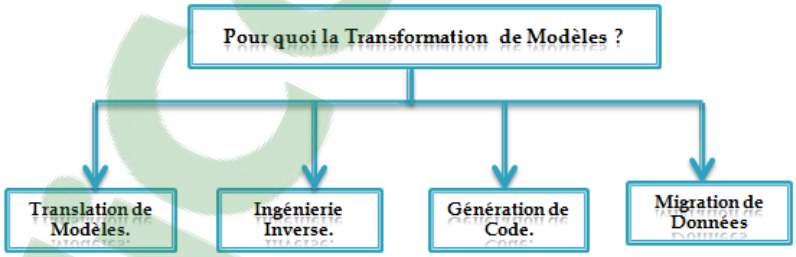

Figure 1.6: Les activités de la transformations de modèles

1.3.1 Génération automatique du code

La génération de code est l'une des principales activités dans la technologie de l'IDM qui a été utilisée fréquemment dans la littérature de recherche à des buts différentes. En général, l'objectif est de transformer un modèle vers un un modèle spécifique représentant un code

source pouvant être exécuté. Un exemple typique d'une telle transformation de modèles, qui a été utilisée fréquemment dans la littérature de recherche est la transformation de modèles Petrinet2Java. Son but est de convertir un modèle de réseau de Petri à un code exécutable Java.

1.3.2 Translation de modèle

La translation de modèle est une transformation d'un modèle à un modèle équivalent mais dans une différente représentation. Une des raisons pour l'utilisation de la translation de modèle est de faciliter l'interopérabilité. Il est souvent souhaitable d'utiliser et d'échanger des modèles entre les différents outils de modélisation, et même entre les différents langages de modélisation. Un exemple typique est la traduction de quelques outils de représentation spécifique de modèles UML dans XMI (et vice versa), langage de type XML et qui est un standard d'OMG pour l'échange de modèles. Cela facilite l'échange des modèles UML entre les différents outils de modélisation UML [76].

1.3.3 Migration de modèles

Une autre utilisation de la transformation de modèles est la migration de modèles. La migration peut être définit comme une transformation d'un modèle écrit dans un langage en un autre modèle écrit dans un autre langage, en gardant le même niveau d'abstraction des modèles. Par exemple, l'évolution du langage (EJB2 2.0) vers une version plus récente telle que (EJB3 3.0), nécessite de migrer les modèles vers une version plus récente à l'aide d'une transformation de modèles qui englobe et automatise cette migration au lieu de migrer chaque modèle individuellement.

1.3.4 Ingénierie inverse

L'ingénierie inverse est un autre exemple d'une transformation de modèles. Elle est l'inverse de la génération de code, et permet de comprendre la structure du programme. Prenant le code source en entrée, elle permet de construire un modèle mental ou visuel (par exemple un modèle UML) à un niveau supérieur d'abstraction, afin de faciliter la compréhension du code et connaître comment il est structuré. L'ingénierie inverse est utilisée pour l'adaptation des applications vers les nouvelles technologies.

1.4 Les approches de transformation

Il existe dans la littérature, plusieurs classifications des approches de transformation de modèles. Généralement, on distingue deux classes principales [20] : les transformations de "modèles à modèles" et les transformations de "modèles à code source". Pour chacune de ces deux catégories, on distingue plusieurs sous-catégories, la figure suivante 1.7 résume cette classification d'approches de transformation de modèles.

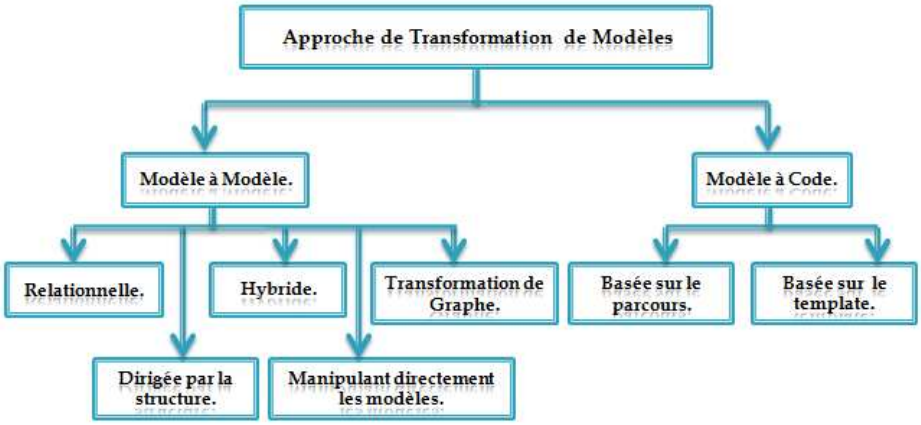

Figure 1.7: Les approches de transformation de modèles

1.4.1 Transformation de modèle à modèle

Plusieurs approches ont été proposées pour la transformation de modèle à modèle : les approches de manipulation directe de modèle, les approches relationnelles, les approches basées sur les transformations de graphes, les approches dirigées par la structure et les approches hybrides.

1.4.1.1 Approches manipulant directement les modèles

Ces approches se basent sur l'utilisation des langages de programmation où elles utilisent les APIs qui permettent de manipuler la représentation interne des modèles. La combinaison JMI (Java Metadata Interface) et Java sont souvent utilisées dans la mise en œuvre de cette approche.

1.4.1.2 Approches relationnelles

L'élément important est l'utilisation des relations mathématiques entre les éléments de modèle source et les éléments de modèle cible. Ces relations peuvent etre classifiées comme

des règles de transformation. Le langage QVT-Relations (QVTr) [73] est un exemple de langage de transformation de cette catégorie.

1.4.1.3 Approches basées sur la transformation de graphes

Cette catégorie s'appuie sur les travaux théoriques de transformation de graphes typés , attribués et étiquetés et où les graphes représentent des modèles de type UML. Viatra [85], ATOM [5] et GrGen [42] sont des exemples d'outils IDM basés la transformation de graphes.

1.4.1.4 Approches dirigées par la structure

Dans cette classe de techniques, la transformation de modèle source passe par deux phases distinctes: la première phase concerne la création de la structure hiérarchique du modèle cible, alors que la deuxième phase définit les attributs et les références et les valeurs de ce modèle. Comme exemples de cette catégorie, on cite : OptimalJ [28], Interactive Objects and Project Technology (IOPT) [67] où les utilisateurs ne se préoccupent que de la définition des règles de transformation et non de l'ordonnancement et de l'application de ces règles.

1.4.1.5 Approches hybrides

Les approches hybrides combinent différentes techniques des catégories précédentes. Les langages de transformation de cette classe combinent les aspects déclaratif et impératif. Cette combinaison est représentée en général, par des règles de mapping définissant les relations entre les éléments sources et cibles et des règles opérationnelles définissant les actions de la transformation. ATL [43], ETL [59], QVTop [73] et ModTransf [31] sont des approches de cette catégorie.

1.4.2 Transformations Modèle à code

Dans cette catégorie, on retrouve les approches basées sur le parcours de modèles et celles basées sur l'utilisation de templates.

1.4.2.1 Approches basées sur le parcours de modèles

La transformation consiste à fournir un certain mécanisme de parcours de la représentation interne du modèle source et l'écriture du code dans un flux en sortie.

1.4.2.2 Approches basées sur les templates

Ces approches sont les plus couramment utilisées dans les outils MDA actuels de génération de code. Un template consiste en un fragment de texte contenant des bouts de métacode permettant :

- d'accéder aux modèles sources,
- d'effectuer une sélection de code,
- de réaliser des expansions itératives.

La structure du template est généralement très proche du code à générer. JET [79], Codagen Architect [27], OptimalJ [28] et ModTransf [31] sont des exemples dans cette catégorie.

1.5 Critères de classification des approches de transformation de modèles

Dans cette section, nous présentons les critères de classification des approches de transformation selon Krzystof Czarnecki et Simon Helsen [20]:

1.5.1 Règles de transformation

La règle de transformation exprime un traitement et possède deux parties : une partie gauche (left hand side "LHS") et une partie droite (right hand side "RHS"). La LHS constitue des accès aux modèles sources, tandis que la RHS présente les créations, les modifications, ou et les suppressions dans les modèles cibles. Chacune des deux parties peut être représentée par une combinaison de :

- *Variables :* Une variable peut contenir un ou plusieurs éléments des modèles source et ou cible (ou des éléments intermédiaires).
- *Patterns :* un pattern est un fragment de modèle qui peut contenir des variables.
- *Logique :* une logique exprime des calculs et des contraintes au niveau des éléments de modèles.

1.5.2 Portée des règles

La portée d'une règle permet de limiter les éléments d'un modèle qui participent à une règle de transformation. Par exemple, lors de l'exécution d'une règle, on ne sélectionne que les éléments du modèle source qui sont permis.

1.5.3 Relation entre le modèle source et le modèle cible

La relation entre le modèle source et le modèle cible peut être définit par trois types : le premier type exprime la relation très faible. Cette relation oblige la création d'un nouveau modèle où il est différent du modèle source. Le deuxième type supporte l'homogénéité du modèle source et le modèle cible où on peut trouver quelques modification ou mise à jour du modèle source. Le dernier type combinent les deux types précédents qu'ils permettent de produire un nouveau modèle cible qui est différent du modèle source où il peut être une copie du modèle source mais il porte quelques modifications.

1.5.4 Ordonnancement des règles

Le mécanisme d'ordonnancement détermine l'ordre d'exécution des règles de transformation. Ce mécanisme peut varier entre quatre cas principaux:

- *Forme :* Le mécanisme d'ordonnancement peut être exprimé implicitement ou explicitement. Dans le premier cas, l'ordonnancement se fait par l'outil de transformation et dans le deuxième cas l'utilisateur définit un algorithme de contrôle d'ordonnancement.
- *Sélection des règles :* Les règles peuvent être sélectionnées par une condition explicite (c.f. Jamda [51]) comme elle peut être implicite (c.f. BOTL [14]).
- *Itération sur une règle :* le mécanisme d'itération de la règle peut être soit une récursivité, soit une boucle, soit encore une itération à point fixe.
- *Décomposition en phases :* Le processus de transformation peut être organisé en plusieurs phases, où chaque phase a un but précis. A chaque phase, on trouve que certaines règles peuvent être invoquées.

1.5.5 Organisation des règles

Nous considérons trois points de variation de la composition et la structuration des règles de transformation :

- *Mécanismes de modularité :* Certaines approches permettent de regrouper les règles en modules (par exemple, OpenQVT [3] et Viatra [85]). Un module peut importer un autre module pour accéder à son contenu.
- *Réutilisation :* La réutilisation offre un moyen de définir une règle basée sur une ou plusieurs autres règles.
- *Structure organisationnelle :* Les règles peuvent être organisées selon la structure du langage source ou le langage cible ou indépendemment de tout langage.

1.5.6 Traçabilité

La traçabilité présente le lien entre les éléments du modèle source et les éléments du modèle. Quelques outils proposent la traçabilité comme CDIQVT [22], IOPT [67]), VIATRA [85], GreAT [40]), etc...

1.5.7 Directivité ou réversibilité :

La transformation peut être unidirectionnelle ou bidirectionnelle. Si elle est unidirectionnelle, elle s'exécute dans un seul sens : le modèle cible est construit ou modifié à partir du modèle source. Une transformation bidirectionnelle s'exécute dans les deux sens où les règles de transformation sont alors soit bidirectionnelles ou constituées de deux jeux de règles unidirectionnelles dont chacun permet la transformation dans un sens.

1.6 Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur l'ingénierie dirigée par les modèles où l'accent a été mis sur la présentation des concepts de base de l'IDM en général et plus spécifiquement la transformation de modèles. Nous avons aussi décrit le rôle principal de la transformation dans le cadre de l'approche IDM et ses activités. L'objectif visé dans ce travail de recherche est de prendre en charge les notions de vérification et de validation dans le processus de transformation de modèles. Ces deux concepts devenus très importants pour assurer une bonne application de l'ingénierie dirigée par les modèles. Le chapitre suivant explicitera plus particulièrement ces deux concepts et leurs domaines d'application.

Chapitre 2

Les Méthodes Formelles

2.1 Introduction

Dans l'informatique, spécifiquement l'ingénierie dirigée par les modèles, les méthodes formelles sont une sorte particulière de techniques mathématiques pour la spécification, le développement, la vérification, et la validation de systèmes et de logiciel.
L'utilisation des méthodes formelles dans le passé et jusqu'à récemment reste presque restreinte. Les notations sont difficiles à utiliser aussi le manque des outils faciles à manipuler pour l'intégration des techniques mathématiques donnent une image noire pour les bien comprendre. En conséquence on trouve très peu de personnes qui maîtrisent efficacement les méthodes formelles.

Les méthodes formelles ont été proposé pour la première fois issues des travaux Programming Research Group d'université d'Oxford qui consistaient à utiliser les concepts mathématiques pour la spécification des systèmes informatiques afin d'ajouter un formalisme rigoureux et valider les logiciels produits.

L'objectif de ce chapitre, est d'introduire les méthodes formelles d'une manière générale. Dans la section 2 nous commençons par la définition d'une méthode formelle, ensuite nous présentons l'utilisation de ces méthodes. La section 3, exprime les différentes classifications proposées pour les méthodes formelles. Enfin, dans la section 4, nous conclurons ce chapitre.

2.2 Pourquoi utiliser les méthodes formelles?

Les méthodes formelles sont une sorte de techniques permettant de raisonner rigoureusement sur des programmes informatiques afin de démontrer leurs conformités (corrections) en se basant sur des raisonnements de logique mathématique. En pratique, une méthode formelle est basée sur une notation formelle permettant d'atteindre des exigences de qualité élevées du système à modéliser.

Définition : *Une méthodes formelle permet à offrir un cadre mathématique permettant de décrire d'une manière précise et stricte les programmes que nous voulons construire. Ce cadre formel vise d'éliminer les ambiguïtés existant au niveau du cahier des charges et du langage naturel [47].*

Dans l'ingénierie de logiciel, les méthodes formelles peuvent être appliquées aux différentes étapes du développement, dans la définition des besoins du client, passant par la conception du système, l'implantation, le débogage, l'entretien, la vérification et jusqu'à à l'évaluation. Comme elles peuvent être aussi utilisées dans l'ingénierie inverse pour modéliser et analyser les systèmes existants.
Un éventail de langages et de techniques formels existe pour traiter différents types de propriétés à différents niveaux de développement des systèmes. En ajoutant que l'utilisation de ces méthodes ne garantit pas l'absence des erreurs, mais elle peut augmenter le degré de correction de ces systèmes. Si on utilise ces méthodes dans le cahier des charges, et les phases de conception on n'a pas seulement la détection des défauts, mais on peut aussi réduire le coût de réparation et de maintenance de ces systèmes. Les principaux avantages des méthodes formelles peuvent être résumés dans les points suivants [37] :

- *Les notations mathématiques et le non ambiguïté* : l'utilisation d'une méthode formelle basée sur des notations mathématiques et précises permet d'éviter les ambiguïtés et les redondances dans la phase de spécification.
- *La détection d'erreurs* : l'application de vérification, de validation et de preuve formelles au processus de développement garantie la détection des bugs de conception le plus tôt possible dans ce processus.
- *La réutilisation* : le raffinement des spécifications formelles et leurs décompositions successives permettent de mettre des niveaux d'abstraction intéressants pour la résolution du problème et pour promouvoir la réutilisation des spécifications.

2.3 Classification des Méthodes Formelles

Il existe différentes classifications basées sur des critères spécifiques. Dans ce qui suit, nous présentons trois classifications qui nous semblent importantes :

2.3.1 Classification de J. Wing

J. Wing [86] considère qu'il y a deux grandes catégories de méthodes formelles (Figure 2.1). La première catégorie concerne les méthodes orientées propriétés. La deuxième catégorie représente les méthodes orientées modèle qui sont basées sur l'utilisation d'un modèle formel du système.

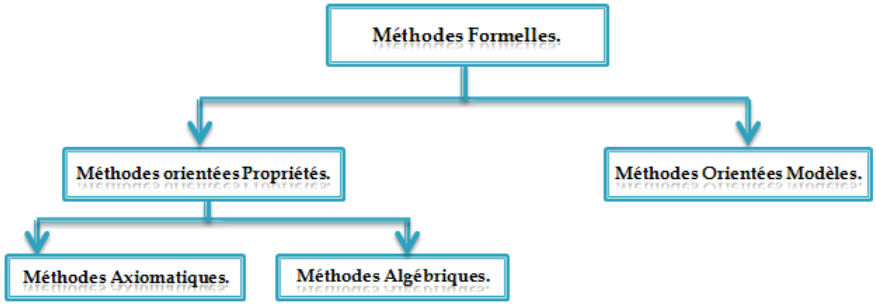

Figure 2.1: Classification de J. Wing

2.3.1.1 Méthodes Orientées propriétés

Les Méthodes Orientées propriétés définissent le comportement du système de manière indirecte en établissant un ensemble de propriétés. Généralement, sous forme d'axiomes que le système doit satisfaire. On distingue dans cette classe, deux sous-catégories :

- ***Méthodes Axiomatiques :*** Le traitement axiomatique du logiciel pourrait bien être le type le plus ancien de méthodes formelles. Un exemple de ces méthodes la logique de Floyde-Hoare [48] qui est généralement appelée la logique de Hoare. Elle est basée sur le traitement axiomatique (utilisation de la sémantique axiomatique). Le but de cette logique est de formaliser la preuve de la correction des programmes. Elle utilise le triplet de Hoare qui se présente de la manière suivante Q P R. Où P est programme et le Q c'est la pré-condition et le R est la post-condition. Si Q est vrai au début de P, alors R sera vrai à la fin du programme P.

- ***Méthodes Algébriques :*** Les Méthodes Algébriques sont apparues en 1970. Ces méthodes se basent sur la spécification algébrique qu'elle utilise les axiomes pour

spécifier les propriétés des systèmes, mais les axiomes sont restreints à des équations. Parmi les méthodes les plus connues, on peut notamment citer : Larch [45], OBJ [35] et Lotos [12] sont des exemples de langages de spécification algébriques.

2.3.1.2 Méthodes Orientées Modèles

Les méthodes orientées modèle définissent le comportement d'un système en construisant un modèle du système en termes de structures mathématiques telles que les tuples, les relations, les fonctions, les ensembles, et les suites. Dans le domaine des programmes séquentiels et des types abstraits de données, on trouve les méthodes VDM [46], Z [13] et B [2]. Pour le domaine des systèmes concurrents et distribués, on cite les Réseaux de Petri [77], CCS de Milner [66], Unity [4], Raise [44] (qui combine VDM et CSP).

1. *Méthode B :* La méthode B [2] est une méthode formelle introduite au milieu des années 80 par Jean Raymond Abrial afin de développer des logiciels sûrs. Elle est définit par Abrial comme suit «B est une méthode pour spécifier, développer et coder les logiciels.». Les utilisations de cette méthode ne se limite pas à la phase de description mais elle permet d'établir une chaîne de production qui va de la spécification du programme au code source associé. Cette methode a rapidement donné une naissance à deux outils : l'Atelier B [6] et le B-Toolkit [62].

2. *Méthode VDM :* VDM [46] (Vienna Development Method) est une des méthodes formelles la plus largement connues et utilisées pour le développement des logiciels et des systèmes complexes et critiques. elle est plus ancienne par rapport à la méthode B et à la méthode Z et a été développée dans un environnement industriel. Mural [53] est l'outil de mise en œuvre de cette méthode.

3. *Méthode Z :* La méthode Z [13] a été développée par un groupe de recherche de l'université d'Oxford avec la participation de Jean-Raymond Abrial après la méthode VDM dans les années 70. Cette méthode est basée sur la théorie des ensembles et le calcul des prédicats. CZT [64] est un des outils les plus connus de cette méthode.

4. *Réseaux de Petri :* Les réseaux de Petri [77] (RdPs) sont des techniques graphiques et mathématiques pour modéliser le comportement dynamique des systèmes concurrents à événements discrets comme les systèmes manufacturiers, les systèmes de télécommunications, les réseaux de transport, ...etc. Leur représentation graphique exprime d'une manière naturelle le parallélisme, la synchronisation, le partage de ressources, et les choix (conflits). Leur représentation mathématique permet d'utiliser les équations d'états, à partir des quelles il est possible de préciser les pro-

priétés du modèle et de les comparer avec le comportement du système modélisé. TINA [78] est un exemple d'outils de cette méthode.

5. *CCS de Milner :* Le langage CCS [66] (Calculus of Communicating Systems) est un langage de traitement concurrent avec une structure algébrique. Ce langage a été introduit par Robin Milner en 1980. Sa syntaxe utilise les processus pour décrire les termes (ou agent). Ces processus peuvent avoir des capacités d'interaction qui correspondent à des demandes de communication sur des canaux nommés.

2.3.2 Classification de Meyer

La classification de Meyer [65] depuis 2001, regroupe les méthodes formelles en quatre catégories : les méthodes algébriques, logiques, dynamiques et ensemblistes (Figure 2.2).

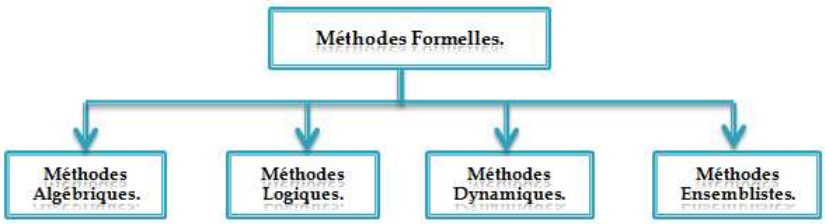

Figure 2.2: Classification de Meyer

2.3.2.1 Méthodes logiques

Généralement, les méthodes logiques se basent sur la théorie des types et des logiques d'ordre supérieur afin d'ajouter une démonstration de programme en appliquant les théories de démonstration automatiques ou semi-automatique de théorèmes. Ces méthodes supportent plusieurs outils pour la mise en œuvre tel que l'assistant de preuve Coq [49].

2.3.2.2 Méthodes dynamiques

Les méthodes dynamiques sont utilisées dans le domaine des protocoles qui se base sur les interactions entre les processus. Ces méthodes peuvent être utiles pour le traitement des systèmes de transitions comme les automates, les réseaux de Petri [77], les algèbres de processus (CSP) et la logique temporelle.

2.3.2.3 Méthodes ensemblistes

Les méthodes ensemblistes utilisent les types abstraits pré-définis afin de préciser l'état du système. Chaque opération est décrite par son effet sur l'état du système. La syntaxe des méthodes ensemblistes correspond aux types et aux opérations de ce modèle abstrait, leur sémantique repose sur la théorie correspondante à ce modèle abstrait (théorie des ensembles, théorie des types, logique du premier ordre). VDM, Z et B sont des exemples de langages de spécifications ensemblistes.

2.3.2.4 Méthodes hybrides

Toutes les méthodes présentées précédemment sont souvent combinées afin de former des méthodes hybrides pour profiter au mieux de leurs avantages respectifs. La méthode Raise [44] est un exemple de ce type de méthode.

2.3.3 Autre classification

J. Wing [26] propose aussi une autre classification qui se base sur l'utilisation des méthodes formelles (Figure 2.3). Deux catégories de méthodes formelles sont proposées où la première catégorie se base sur la spécification des systèmes comme les méthodes Z, B et VDM et la deuxième catégorie s'intéresse à la vérification du produit final tels que le model cheking [25] et le théorème de preuve.

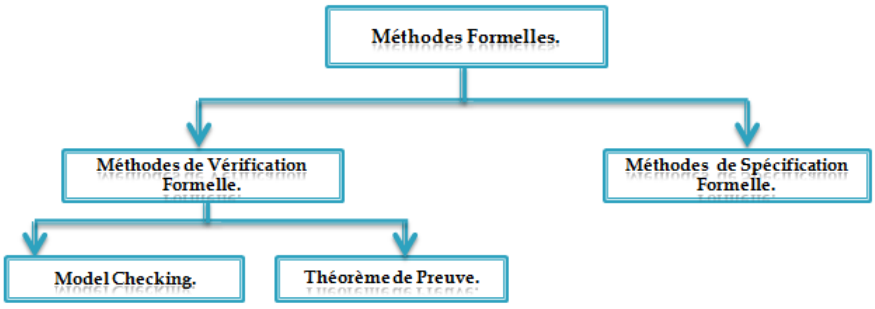

Figure 2.3: Autre classification de J. Wing

2.3.3.1 Méthodes de Spécification Formelle

Dans cette catégorie, l'accent est mis sur la spécification des systèmes. Certaine méthodes formelles comme Z, VDM, et larch sont basées sur la spécification du comportement des

systèmes séquentiels. D'autres méthodes telles que CSP, CCS et la logique temporelle sont basées sur la spécification du comportement des systèmes concurrents.

2.3.3.2 Méthodes de Vérification Formelle

Ces méthodes s'intéressent à la vérification formelle afin de garantir la sûreté de fonctionnement des systèmes complexes et critiques. Pour ce faire, il faut exprimer clairement le comportement souhaité du système. Les méthodes formelles englobent deux types de méthodes pour la vérification formelle : le modèle checking et le théoreme de preuve.

1. *Model checking :* Le model checking [25] est une technique de vérification automatique des système dynamiques. Elle permet de vérifier si un modèle du système satisfait une propriété souhaitée d'une façon automatique. Si cette propriété n'est pas satisfaite, fourni un contre exemple est fourni pour aider à expliquer la source de l'erreur. L'avantage principal de cette méthode est que la vérification de propriétés est complètement automatisée. Parmi ses différents outils, on cite: SPIN [41], et UPPAAL [8].

   (a) *SPIN :* Spin [41] (Simple Promela INterpreter) est un outil conçu pour la vérification générique des systèmes distribués. Cet outil utilise comme un langage de spécification le langage Promela (Protocol Meta Language). Ce dernier conçu pour spécifier les systèmes parallèles et asynchrones.

   (b) *UPPAAL:* UPPAAL [8] est un outil de vérification automatique des systèmes temps réel développé par une collaboration entre le département d'Information Technology de l'université d'Uppsala (Suède) et le département de Computer Science de l'université D'Aalborg au Danemark.

2. *Théorème de preuve :* Le théorème de preuve possède des racines qui remontent à la logique mathématique, en plus c'est un moyen de construire la preuve mathématique. Il existe de nombreux outils de preuve semi-automatique tels que Coq, Isabelle et PVS.

   (a) *Assistant de preuve Coq :* Coq [49] est un système de manipulation de preuves mathématiques formelles. Avec cet assistant, il est possible d'énoncer des théorèmes, des spécifications de programme, et de développer interactivement leurs preuves à l'aide de tactiques. Le programme est écrit par le langage fonctionnel Gallina qui se base sur le calcul des constructions inductives, étendu avec un système de développement modulaire des théories.

   (b) *Assistant de preuve Isabelle :* Isabelle [69] est un système de preuves développé par Paulson et Nipkov. La majorité des assistants de preuve sont basés sur

un unique calcul formel, en général la logique d'ordre supérieur. Isabelle a la capacité d'accepter une variété de calculs formels.

(c) *Assistant de preuve PVS :* PVS (Prototype Verification System) [75] est un outil de vérification intégrant le langage de spécification PVS qui se base sur la logique classique d'ordre supérieur. Cet assistant de preuves est écrit dans le langage de programmation Lisp.

2.3.4 Comparaison des classifications

Dans cette section, nous présentons une comparaison entre les différentes types de classification des méthodes formelles présentées dans les sections précédente. On Illustre à chaque fois une classification en fonction de critères différents. La première classification se fait selon la structure de raisonnement basée sur le type du système à résoudre (système séquentiel ou concurrent). La deuxième classification a été établit à base de la sémantique des langages des méthodes, et la troisième classification a été proposée en prenant en compte la nature de la tâche à résoudre.

2.4 Conclusion

Dans ce chapitre, nous avons rappelé l'importance de l'utilisation des techniques formelles, leurs différentes classifications en précisant dans quelle mesure elles sont pertinentes pour la spécification, la vérification et avec quel type de systèmes.

Ces méthodes ont été utilisées plus particulièrement dans certains domaines et nous allons essayer notamment de les classifier selon le niveau d'intégration avec l'ingénierie dirigée par les modèles en interprétant le concept de vérification et de validation. A cet effet, le chapitre suivant présente donc les différentes propositions qui intègrent les méthodes formelles avec l'ingénierie dirigée par les modèles autour desquelles nous avons décidé de construire notre approche, particulièrement axée sur la vérification et la validation des transformations de modèles.

Chapitre 3

Vérification et Validation dans l'Ingénierie Dirigée par les Modèles

3.1 Introduction

Dans ce chapitre, nous essayons d'introduire deux notions principales dans l'ingénierie dirigée par les modèles et plus particulièrement au niveau de la transformation : La vérification et la validation, ensuite nous décrivons l'état relatif à l'introduction des méthodes formelles pour ces deux notions où nous précisons, les techniques proposées pour la vérification et la validation dans la transformation de modèles ensuite, nous positionnons notre approche de transformation de modèles par rapport aux travaux existants.

Les notions de vérification et de validation ont été définies à la fin des années 70 dans le contexte des sciences expérimentales en général et plus précisément dans le génie logiciel. De temps en temps, ces définitions ont été progressivement précisées. depuis l'émergence de l'IDM, on assiste à adapter ces deux notions, pour détecter et corriger les erreurs en plus pour garantir une bonne production du logiciel. Ainsi, les chercheurs se sont intéressés à la prise en charge de ces deux notions au niveau de la transformation de modèles et qui consiste d'abord à déterminer s'il existe des techniques de vérification ou de validation pour faciliter à offrir des transformations fiables, correctes et prêtes à utiliser dans le contexte de l'IDM.

La première technique proposée pour résoudre cette problématique est le "Test". Cette dernière est une méthode classique de vérification et de validation par l'exécution d'une partie ou de la totalité du code implémenté. Le but du test de logiciel est de

détecter les erreurs des programmes le plus tôt possible. L'inconvénient majeure de cette technique est l'absence de raisonnement mathématique sur les propriétés de la spécification, car ces techniques reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise.

Puisque les techniques formelles se basent sur les aspects mathématiques, les chercheurs proposent de les intégrer dans l'IDM. Dans ce chapitre, nous citons quelques travaux relatifs sur les techniques de vérification et de validation au niveau de la transformation de modèles (Figure 3.1).

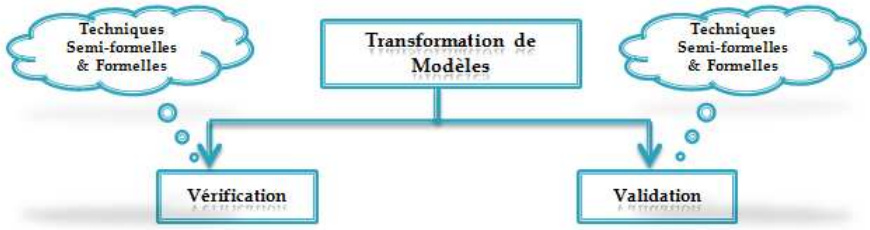

Figure 3.1: La Vérification et la Validation dans la transformation de modèles.

Dans ce qui suit, les sections 3.2 et 3.3 discutent sur la vérification de la transformation et les techniques proposées. La section 3.4 ajoute un sens plus large sur la validation dans l'IDM. L'utilisation des techniques semi-formelles et formelles dans la validation de la transformation de modèles est ensuite présentée dans la section 3.5. Et le chapitre termine par une conclusion sur ces deux notions.

3.2 Vérification dans la transformation de Modèles

L'intégration du concept de la vérification dans l'ingénierie dirigée par les modèles dans le développement des systèmes permettra de bien produire les modèles où la réutilisation de ces derniers est considérée comme une tâche importante. Avant de présenter les différentes techniques de vérification nous définissons la notion de vérification dans le contexte de l'IDM.

Vérification : *C'est une méthodologie assurant que le modèle est conforme à sa spécification. Elle cherche à répondre à la question suivante:"Construisons nous correctement le modèle ?" [50]. Elle traite soit la conformité des modèles à leurs métamodèles, ou le contrôle des règles de transformation.*

Dans la section suivante, nous décrivons les techniques d'application de la vérification dans la transformation de modèles.

3.3 Techniques de Vérification

Plusieurs techniques de vérification ont été utilisées dans la transformation de modèles. Elles peuvent être classifiées en deux catégories (Figure 3.2). La première concerne les techniques semi-formelles, c'est-à-dire que, malgré l'utilisation des définitions rigoureuses, elles ne se basent pas sur un raisonnement mathématique, car ces méthodes reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise. La seconde regroupe les méthodes qui font appel à la logique et aux mathématiques pour effectuer cette vérification.

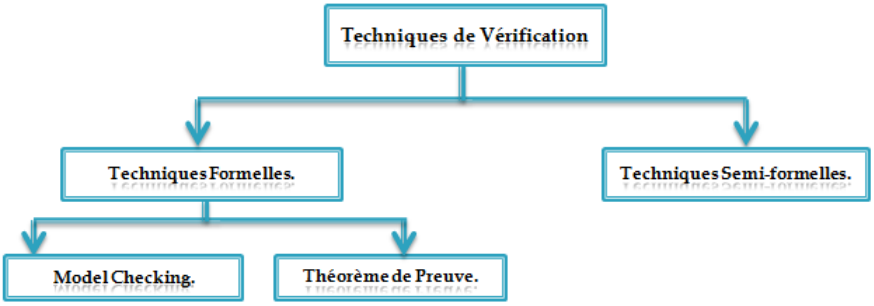

Figure 3.2: Les techniques de Vérification.

Dans ce qui suit, nous nous présentons les méthodes semi-formelles et les méthodes formelles, et particulièrement celles qui sont proposées pour définir la vérification de la transformation de modèles dans le cadre de l'IDM.

3.3.1 Techniques semi-formelles

Généralement, on trouve plusieurs techniques de vérification qui sont utilisées au sein de l'ingénierie dirigée par les modèles, notamment celles qui reposent sur les tests. Les méthodes de test permettent de détecter un grand nombre d'erreurs avec un faible coût de mise en œuvre par rapport aux techniques formelles comme la démonstration. Un test est une méthode de vérification par exécution d'une partie ou de la totalité du code implémenté [68]. En pratique, un test est effectué en deux phases : la construction des tests et l'exécution de ces tests.

La vérification de la transformation de modèles par les techniques semi-formelles peut être appliquée avec différentes méthodes: le test manuel, le test automatique et la vérification par relecture. Nous les présentons dans cette sous-section.
Fleurey et al. [34] proposent d'utiliser le test fonctionnel pour la transformation de modèles en se basant sur un ensemble de critères.
Dans [15], Brottier et al. proposent un processus et un outil pour la génération automatique de modèles de test satisfaisant les critères de test de Fleurey et al. [34].
Lamari et al. [61] présentent un outil qui génère automatiquement des modèles pour le test de n'importe quelle transformation. Pour appliquer cette idée, la spécification de la transformation est implémentés complètement en langage formel MTSpecL, langage proposé dans l'article.

3.3.2 Techniques formelles

Les techniques de vérification formelle désignent un ensemble de méthodes basées sur les mathématiques et la logique pour assurer qu'un produit final est conforme à ses spécifications. Ces dernières doivent être implémentées dans un langage défini mathématiquement (syntaxe et sémantique), et que nous présentons quelques-unes dans ce qui suit.

3.3.2.1 Vérification formelle par preuve

La transformation de modèles dans l'IDM nécessite l'implémentation des techniques formelles pour vérifier le processus de transformation. Ces techniques se basent sur la preuve d'un ensemble de propriétés contenues dans la transformation de modèles. Comme un exemple de ces techniques, on peut citer : Coq, SMT Solver, Isabelle, KIV, et Maude.

- **L'assistant de preuve Coq :** Dans [19], les auteurs définissent une transformation simple d'un diagramme UML vers une base de données relationnelle en utilisant le langage KM3 pour définir les métamodèles source et cible. Les règles de transformation sont implémentées par un langage hybride qui s'appelle ATL. Pour vérifier la correction de cette transformation, ils ont intégré une méthode formelle qui se base sur la preuve et qui est "l'assistant de preuve Coq". Cette intégration traduit les deux métamodèles vers le langage Gallina, plus les règles de transformation. La vérification se fait par la preuve interactive avec Coq.
  La même idée a été appliquée dans [18] mais avec un autre exemple simple qui translate une liste vers un arbre.

- **L'assistant de preuve Isabelle :** Le travail [81] de Bernhard Schatz et al. utilise l'assistant de preuve Isabelle pour vérifier la transformation des modèles EMF

Ecore. Aussi, le langage prolog est utilisé pour définir la structure des métamodèles, modèles et les règles de transformation. Ces structures seront ensuite traduites en Isabelle pour prouver la correction de la transformation.
L'idée de fournir une vérification par un assistant de preuve est largement utilisé [38] où Holger Giese1 et al. définissent une transformation d'un automate d'états vers un code exécutable avec le langage AGG et la vérification se fait par l'assistant de preuve Isabelle.

- **L'assistant de preuve Maude :** Le travail de Javier Troya et al. [84] se base sur une transformation d'un code source Java vers une table en utilisant le langage ATL et la vérification se fait par l'assistant de preuve Maude.
- **Le solveur SMT :** Fabian Büttner1 et al. [17] proposent d'utiliser le solveur SMT pour vérifier un exemple de transformation assez simple. Cette transformation a été implémenté par le langage dédié ATL.

3.3.2.2 Vérification formelle par "Model checking"

Le modèle checking est une autre méthode de vérification formelle. Elle se base sur la construction de toutes les situations possibles d'un modèle, ensuite de les vérifier automatiquement sans interaction avec l'utilisateur.
L'article de Garcia Miguel et al. [36] présente une vérification de la transformation de modèles en utilisant un model checking qui utilise le langage plusCal (+ CAL).

Suivant les exigences de la transformation de modèles, il est également nécessaire d'intégrer la notion de validation pour avoir une bonne production du modèle final. La section suivante explique cette notion et ses techniques d'application dans le concept de transformation.

3.4 Validation dans l'IDM

La notion de validation dans l'IDM en général, et spécifiquement dans la transformation de modèles, a un rôle crucial pour augmenter l'automatisation à ce niveau. En général, elle assure que la transformation implémentée est une bonne transformation. Comme nous pouvons *définir dans ce contexte la validation par l'équivalence entre le modèle obtenu et le modèle désiré.*

Validation : *La validation est une technique consiste à garantir que le modèle résultant doit satisfaire les exigences particulières pour un usage spécifique. Elle cherche à*

répondre à la question suivante:"Construisons nous le bon modèle ?" [50].

A partir de cette définition, nous présentons ci-dessous les techniques de validation proposées pour la transformation de modèles afin d'augmenter le niveau d'automatisation de développement des logiciels dans le conteste de l'IDM.

3.5 Techniques de validation

Plusieurs techniques de validation sont applicables sur la transformation de modèles dans le cadre de l'IDM. Elles peuvent être classifiées comme la vérification en deux catégories: les techniques semi-formelles et les techniques formelles (Figure 3.3).

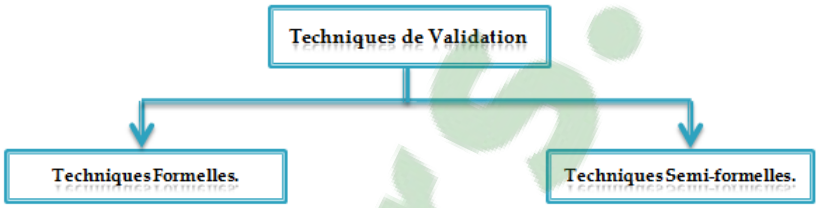

Figure 3.3: Les techniques de Vérification.

3.5.1 Techniques semi-formelles

La validation de transformation de modèles par test est l'une des plus anciennes techniques utilisées dans l'IDM. Elle se base sur le choix d'un ensemble de test pour s'assurer et augmenter l'usage de la transformation automatique. Dans [60], Küster a réalisé un travail sur le test de transformations de modèles UML en utilisant une spécification formelle de la transformation sous forme de règles de transformation de graphes attribués et typés.
Steel et al.[83] définissent un retour d'expérience issu de la validation d'une transformation déclarative de modèles. Pour le test de cette transformation, chaque donnée de test est composée du modèle d'entrée et du modèle attendu en sortie. Après l'exécution de la transformation du modèle en entré, le modèle de sortie obtenu est comparé avec le modèle de sortie attendu. Les auteurs remarquent que ce processus de test est très similaire à un processus de vérification de transformations de modèles, mais il pose des problèmes : premièrement la construction des modèles du test est trop difficile, deuxièmement le test n'est pas une solution systématique ou automatisable.

Dans ce qui suit, nous présentons les techniques qui se fondent sur les concepts mathématiques qui ont été utilisé dans le cadre de validation de la transformation de modèles.

3.5.2 Techniques formelles

La validation s'intéresse la vérification de la conformité par rapport aux spécifications. Comme elle peut être définit par la vérification de l'équivalence entre le modèle en sortie et le modèle attendu. Dans le contexte de transformation de modèles, elle peut être assurée par les techniques formelles. Dans ce cadre, l'intégration de ces techniques pour la validation n'est pas très aisée à appliquer. Généralement, elle se fait par des techniques semi-formelles.

3.5.2.1 Validation formelle par la méthode B

Idani et al. propose de définir un usage de méthodes formelles pour l'IDM. L'objectif principal de ce travail s'occupe de la validation des transformations de modèles avec une technique formelle de preuve qui s'appelle la méthode B où les contraintes de sûreté sont exprimées par des annotations en langage B.

3.6 Conclusion

La vérification et la validation sont deux notions essentielles au cycle de vie du développement des systèmes soit en génie logiciel ou en l'ingénierie dirigée par les modèles. Elles peuvent être opérées à chaque phase du développement ou aux certaines phases selon leurs méthodes d'application.

Dans ce chapitre nous avons donné un aperçu sur la vérification et la validation de la transformation de modèles dans le cadre de l'approche IDM. Nous avons défini ces deux notions, plus les techniques d'application proposées, et plus particulièrement les méthodes formelles. Dans le chapitre suivant, nous développerons l'environnement permettant d'intégrer la vérification et la validation dans le processus de l'IDM.

Chapitre 4

Outils d'expérimentation

4.1 Introduction

L'objectif de ce chapitre est de présenter les méthodes et les outils proposés dans notre approche qui se base sur l'hybridation entre une méthode formelle et l'ingénierie dirigée par les modèles.
L'IDM apporte des méthodes et des outils de méta-modélisation et de transformation de modèles afin de faciliter le développement des systèmes. Elle permet d'agir à haut niveau d'abstraction et de spécifier des transformations de modèles afin d'automatiser la production de tout ou partie d'un système. Parmi ces outils, nous présentons le langage Ecore qui appartient aux outils de méta-modélisation, le langage OCL qui permet d'exprimer les contraintes afin de vérifier la conformité au niveau de méta-modélisation, et le langage dédié de transformation QVT opérationnel.
Les méthodes formelles apportent aussi des outils de spécification, de vérification et de validation qui peuvent être utilisés dans différents domaines d'application. A cet effet, nous donnons un aperçu sur l'outil formel que nous avons intégré dans notre approche, l'assistant de preuve Coq qui utilise le langage formel Gallina et un ensemble de tactiques et de commandes qui peuvent être appliqués dans le processus de preuve.

4.2 Outils de l'IDM

L'émergence de l'IDM est le fruit d'une longue évolution en génie logiciel, en fournissant des méthodes, des techniques et des outils concourant à la production d'un logiciel. Cette approche a fournit des langages de méta-modélisation, plus abstraits et plus facile à maîtriser que des langages de programmation tel que Java ou C. En plus, l'IDM propose d'utiliser les langages de transformation de modèles dans le but de faciliter la translation

des modèles vers des autres jusqu'à un code exécutable.
Dans ce qui suit, nous présenterons les langages que nous avons utilisé pour notre recherche au niveau de la méta-modélisation et au niveau de la transformation de modèles.

4.2.1 Langage Ecore

Le langage Ecore [16] est un langage de méta-modélisation graphique et textuelle défini par IBM et utilisé dans le framework de modélisation d'Eclipse EMF (Eclipse Modeling Framework). Il est utilisé pour définir les métamodèles et le langage OCL est intégré pour d'écrire les contraintes dans le but de vérifier la conformité des modèles à leurs métamodèles.
Nous choisissons d'utiliser ce langage à base des critères suivants :

- Le langage Ecore est un langage graphique et textuel. La réprésentation graphique du langage permet de manipuler directement les concepts, sans passer par une formalisation textuelle abstraite et difficile à acquérir.
- Ecore intègre le langage formel OCL qui facilite l'implémentation et permet de d'écrire les contraintes. Une contrainte est une expression que l'on peut attacher à n'importe quel élément du métamodèle.

Les figures (4.1, 4.2) illustrent un exemple de métamodèle de la machine d'états et un autre métamodèle pour les réseaux de Pétri dont il existe de nombreux métamodèles dans la litterature.

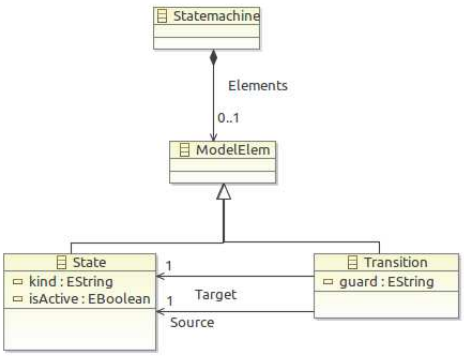

Figure 4.1: métamodèle de la machine d'états.

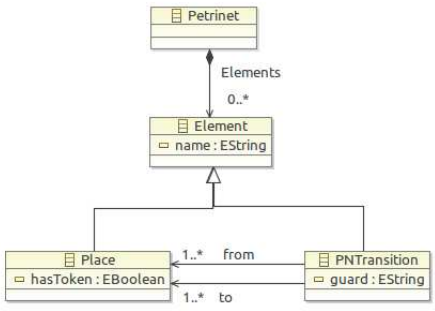

Figure 4.2: métamodèle de Réseau de Petri.

La figure 4.3 montre la représentation textuelle du métamodèle de la figure 4.1 avec le langage Ecore.

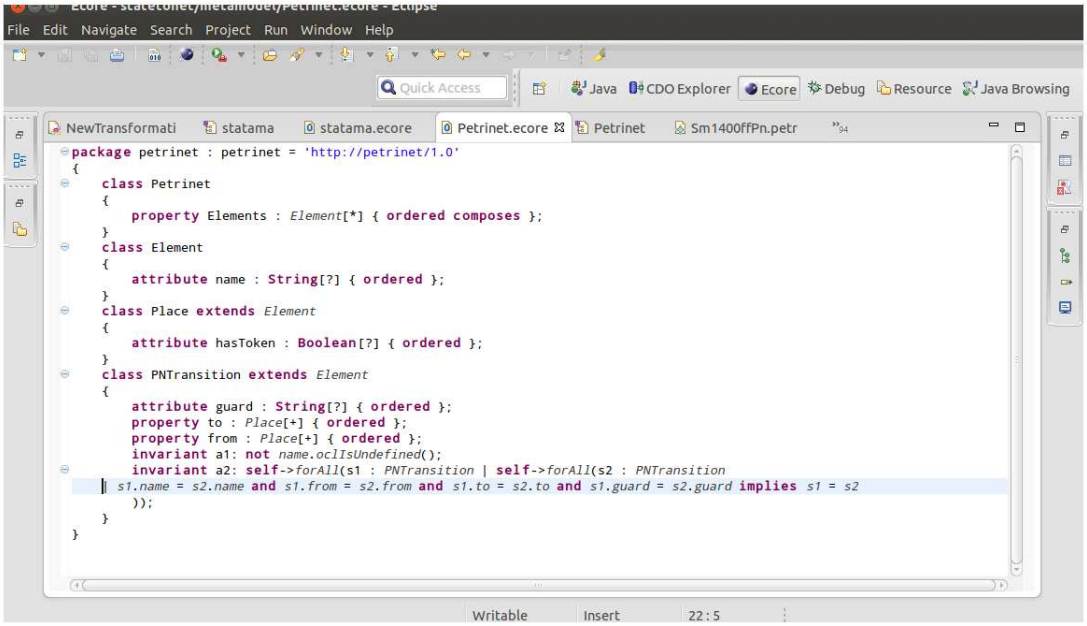

Figure 4.3: Présentation textuelle du métamodèle de Réseau de Petri.

4.2.2 Langage OCL

Le langage OCL (object Constraint Language) [74] a été développé en 1997 par Jos Warmer (IBM). Il a été officiellement incorporé dans UML 1.1 en 1999. OCL est un langage formel qui est basé sur la notion de contraintes. Une contrainte est une expression booléenne qui peut être attachée à tout élément d'UML. Ce langage indique généralement une restriction ou donne des informations sur un modèle. Les contraintes sont notamment utilisées pour décrire la sémantique d'UML et ses différentes extensions, en participant à la définition des profils UML. Le choix de ce langage est justifié par :

- OCL est un langage formel volontairement simple d'accès.
- OCL représente en fait un bon milieu entre un langage naturel et un langage très technique (langage mathématique, informatique, ...).
- Il permet ainsi de limiter les ambiguïtés, tout en restant accessible.
- Langage OCL est intégré dans le langage Ecore, donc il est facile à utiliser pour implémenter les contraintes.

Dans ce qui suit, nous présentons deux contraintes en langage OCL. La première exprime que les noms sont obligatoire. Pour la deuxième, explique le nom ambiguïté des

transitions c-à-d on ne peut pas trouver deux transitions avec le même nom, la même source et destination et la même condition.

```
invariant a1: not name.oclIsUndefined();
invariant a2: self->forAll(s1 : PNTransition | self->forAll(s2 : PNTransition | s1.name =
s2.name and s1.from = s2.from and s1.to = s2.to and s1.guard = s2.guard implies s1 = s2));
```

4.2.3 Langage QVT

Le langage QVT (Query/View/Transformation) [70] est un standard pour les transformations de modèles défini par l'OMG dans le cadre de l'architecture MDA. Il offre un moyen pour transformer les modèles sources vers les modèles cibles. Les modèles sources et cibles doivent être conformes au métamodèle MOF. Ainsi le langage QVT possède deux parties principales : une partie déclarative regroupant les langages qui possèdent une structure déclarative telles que QVT-Relation et QVT-Core, et une partie impérative qui définit la structure impérative comme QVT-Operational (cf. Figure 4.4).

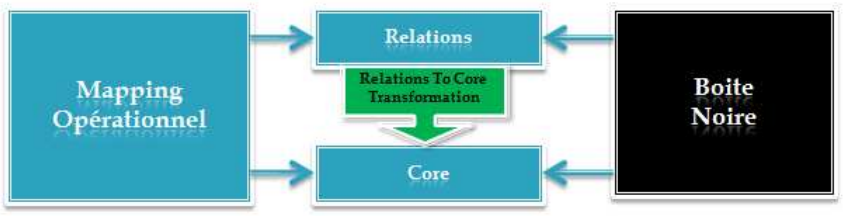

Figure 4.4: Notions de base en ingénierie des modèles

4.2.3.1 La partie déclarative

La partie déclarative de la spécification QVT est définit dans une architecture à deux niveaux : le langage de relations (Relations) et le langage noyau (Core).

1. *Le langage de relations :* permet de spécifier les règles de transformation par des relations entre les métamodèles. Généralement il se base sur l'utilisation de Patterns d'Objets. Un pattern est un sous métamodèle qui représente un ensemble d'éléments satisfaisant certaines caractéristiques tout en restant conformes au métamodèle du modèle de départ [1]. Ce langage supporte la création et la suppression des objets, ainsi que la gestion des informations de trace des transformations d'une façon automatique.

2. *Le langage noyau :* Le langage Core est une extension minimale de EMOF (Essential MOF) et OCL. Ce langage ne supporte pas l'utilisation de Patterns d'Objets. Dans ce cas, l'utilisateur définit ses règles de transformations et gère la création de la trace des transformations à l'aide d'un métamodèle MOF. L'absence de mécanisme d'automatisation de trace et la définition des éléments rend ce langage plus simple, mais difficile à exploiter en pratique.

4.2.3.2 La partie impérative

La norme QVT propose une partie impérative connue par "Operational Mappings" ou "Operational QVT (QVT-op)". Ce dernier permet d'écrire des transformations dans un style plus traditionnel comme tous les langages de programmation Java ou C. Il intègre aussi le langage OCL afin de spécifier des requêtes et faciliter l'implémentation des transformations.

Par conséquent, une transformation peut être implantée soit par un langage déclaratif comme le langage de relations ou le langage de noyau ou par un langage impératif comme le langage Operational Mappings.
QVT propose deux façons d'utiliser le langage Operational Mappings. Premièrement, il peut être utilisé pour spécifier une transformation uniquement dans lui-même. Deuxièmement, il peut être utilisé de façon hybride [7]. Dans ce cas, l'utilisateur peut spécifier certains aspects de la transformation dans un des langages déclaratifs (Relations ou Core) et implémenter, en boîte noire, des règles dans le langage Operational Mappings. Le choix de ce langage justifié par ces caractéristiques suivantes:

- QVT-op peut être utilisé pour spécifier une transformation complète dans lui-même. Comme il peut être utilisé d'une façon hybride, où il définit une partie d'une transformation.

- Il utilise un langage formel comme OCL pour faciliter l'implémentation des requêtes.

Le code suivant présente un exemple de transformation d'une machine d'état vers un réseau de Pétri en langage QVT-op. Cet exemple de transformation utilise les deux métamodèles précédemment.

```
modeltype pn uses 'petrinet_1';
modeltype sm uses 'statemachine_1';
transformation testTrafo(in inModel : sm, out outModel : pn);
main() { inModel.rootObjects() [Statemachine] -> map Statemachine2PetriNet();
} mapping Statemachine::Statemachine2PetriNet() : PetriNet {
elements := self.elements[State] -> map State2Place();
elements += self.elements[ModelElem] -> map ModelElem2Element();
} mapping ModelElem::ModelElem2Element () : Element
when{ self.name ! = null } {
name := self.name;
} mapping State::State2Place() : Place inherits ModelElem::ModelElem2Element
when{self.kind! =?initial? } {
result.hasToken := self.isActive; }
```

4.3 Outil formel : Assistant de preuve Coq

Le système Coq est développé à l'INRIA dans le cadre du projet TypiCal. Coq est un assistant de preuve interactif dans lequel des spécifications sont implémentées dans le langage Gallina. Ce dernier se base sur le calcul des constructions inductives [30]. Coq n'est pas seulement un assistant de preuve, mais également un atelier de construction de programmes corrects par extraction depuis la spécification vérifiée. Les preuves sont construites impérativement à l'aide des tactiques, qui sont des outils préalablement existants dans l'assistant de preuve. COQ fait partie des assistants de preuve les plus utilisés. Il est similaire à HOL [39], Agda [29], LEGO [63], Isabelle [69] et d'autres assistants de preuves interactifs.

4.3.1 Les termes de base

L'assistant de preuve Coq utilise le langage Gallina dans lequel les définitions sont des termes qui sont eux-mêmes typés; leurs types sont appelés Sortes. Coq fait la distinction entre deux catégories de types : le type Prop pour les propositions (formules logiques) et le type Set pour les types de données. Ces deux types sont aussi typés et leurs type commun est nommé Type. Le tableau 4.1 illustre un résumé de la syntaxe de Coq pour les propositions logiques et les quantificateurs. Comme il existe des fonctions prédéfinis on cite: la fonction `"if else"` pour définir une condition, la fonction `"match with"` pour définir les conditions multiples, la fonction `"case"` et la fonction `while`.

$\bot$	$\top$	T=U	$T \neq U$	P	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$	$\forall x, p$	$\exists x, p$
True	False	T=U	t<>u	P	P/ Q	P -> Q	P<->Q	forall x, p	exists x, p

Tableau 4.1: La syntaxe de Coq pour les propositions et les quantificateurs.

4.3.2 Types inductifs

Dans le système de preuve Coq, les types de données peuvent être spécifiés par les types inductifs qui sont spécifiés par le nom et le type de la famille des données définies, les noms et les types de ses constructeurs. Typiquement, on peut définir les listes à l'aide de deux constructeurs *nil* et *cons* :

```
Inductive list (A : Type) : Type :=
|nil : list A
| cons : A -> list A -> list A.
```

4.3.3 Fonctions et Fonctions récursives

Le système de preuve Coq fournit une commande qui permet de définir des valeurs, des propositions ou des fonctions. Ces dernières sont généralement définies en donnant un nom et le type de la valeur qu'elle retourne. En pratique, ce procédé de construction est fourni dans le système de preuve Coq par la commande *Function*. Ce qui suit présente une fonction qui définit l'entête d'une liste :

```
Definition head (default:A) (l:list) := match l with
| nil => default
| x :: => x end.
```

Dans un langage de spécification formel comme le langage Gallina, il est important d'avoir une commande qui permet de définir les fonctions récursives. Ces dernières peuvent être définies par la commande suivante *Fixpoint*. L'exemple suivant illustre une utilisation de cette fonction.

```
Fixpoint Concat (l m: list) :struct l : list := match l with
| nil => m
| a :: l1 => a :: Concat l1 m end.
```

4.3.4 Théorème et Preuve

Pour démontrer qu'une proposition soit vraie, nous devons produire une preuve. Une preuve avec l'assistant de preuve Coq se définit par l'application d'un ensemble de tactiques. Si une tactique appliquée a un but donné, ce dernier sera traduit en un ensemble de sous-buts tel que la preuve de ces sous-buts suffit pour prouver le but.
Avant d'introduire le théorème qui prouve que *∀ l l' : list A, length (concat l l') = length l + length l.* nous essayons de présenter le sens d'un ensemble de tactiques utilisées dans le système Coq pour faciliter la preuve.

- **La tactique intro(s) :** Généralement elle est la première tactique employée lors des preuves et permet à introduire les hypothèses du calcul. Elle correspond au passage de `"Prouvons que A implique B"` à `"Supposons A, prouvons B."` ainsi qu'au passage `"Prouvons que pour tout x de A, P(x)"`, à `"Soit x dans A, prouvons P(x)"`.

- **La tactique apply :** Cette tactique consiste à appliquer une hypothèse afin de modifier le but. Coq fonctionne à l'inverse de ce qu'on pratique souvent : au lieu de modifier nos hypothèses pour arriver à notre but, Coq modifie le but jusqu'à remonter à nos hypothèses. Ainsi "apply hypothèse" où hypothèse est un terme de type A -> B permet de transformer le but actuel de B en A. Donc le but doit être compatible avec le terme passé en argument.

- **La tactique induction :** La tactique induction `id` permet de faire un raisonnement par induction sur `id`. Dans le cas de la définition des entiers naturels par exemple, elle applique le principe de récurrence.

- **La tactique simpl :** Son but est d'appliquer des définitions pour simplifier des calculs. Par exemple simpl permet de transformer un but de type : `plus (Succ foo) bar = beuh` par `Succ (plus foo bar) = beuh` qui est une application simple de la définition Succ.

Un théorème intéressant et simple à démontrer est *∀ l l' : list A, length (concat l l') = length l + length l.* La preuve de cette formule est comme suit :

```
Lemma concat length : forall :l l' : list A, length (concat l l') = length l + length
l'.
Proof.
intros.
induction l; simpl; auto.
Qed.
```

Pour prendre en charge les concepts de validation, nous rappelons qu'il existe différents outils formels. Notre choix s'est porté sur l'assistant de preuve Coq pour les raisons suivantes :

- ***Translation:*** La translation représente la traduction d'un langage vers un autre langage. Au Niveau de Coq, il est facile de traduire les programmes écrits en Gallina vers des programmes exécutables comme Ocaml.
- ***Domaine d'application:*** Généralement Coq a été utilisé dans différents domaines d'application. Comme exemple, il a été utilisé pour certifier le compilateur du langage C et aussi pour vérifier [19, 18] et valider [9] différents programmes de transformation.
- ***Flexibilité:*** elle représente le niveau de base de chaque outil de preuve c-à-d s'il est construit autour d'une variété de calcul formel ou un seul calcul formel. Pour Coq, on peut dire qu'il est flexible puisqu'il se base sur le lamda calul et le calcul d'ordre supérieur.
- ***Niveau d'automatisation:*** Coq est un assistant de preuve interactif semi-automatique. En général, il se base sur la preuve manuelles des utilisateurs et permet d'ajouter des tactiques afin de faciliter l'automatisation du preuve avec une interaction continue avec l'utilisateur.
- ***Niveau d'intégration:*** L'assistant de preuve Coq fait partie des assistants de preuve les plus utilisés comme Isabelle. En plus il est similaire à HOL, Agda et Isabelle.

4.4 Conclusion

Nous avons introduit dans ce chapitre les outils utilisé dans notre démarche de l'IDM, c'est à dire dans la méta-modélisation d'une part et la transformation de modèles d'autre part. Ces deux concepts constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement. Notre objectif est de proposer une approche qui intègre une technique formelle avec l'IDM dans le but de vérifier et valider la transformation de modèles.
Nous avons également présenté l'outil de preuve formelle utilisé dans le cadre de notre approche. Les outils formels permettent de réduire le coût et aussi faciliter la maintenance des programmes de transformation de modèles.

Afin d'illustrer cette démarche de formalisation de la vérification et de la validation dans le contexte de l'IDM, il est nécessaire d'expliciter ceci à travers la présentation d'un exemple d'étude de cas. Cela fait l'objet du chapitre suivant.

Chapitre 5

Transformation de Modèles : Etude de cas

5.1 Introduction

L' OMG a proposé en 2000 l'approche MDA (Model Driven Architecture) dans le but de bénéficier des avantages de manipulation des modèles. Cette approche vise à la définition d'un cadre normatif pour la définition et l'utilisation des modèles dans le processus de développement des applications informatiques.

La transformation de modèles est l'un des principaux concepts de l'IDM, elle permet de translater des modèles vers d'autres modèles avec différents niveaux d'abstraction dans le but de réduire le coût et le temps de développement. La transformation des modèles n'est bien sûr pas une tâche facile à implémenter. Il est donc nécessaire d'avoir des outils puissants pour la gestion des modèles, et des langages dédiés pour leurs transformations.
Le travail présenté dans ce chapitre illustre la transformation de diagramme d'états-transitions vers le réseau de Petri. C'est une étude de cas qui esquisse l'approche suivie pour la transformation de modèles. Cette transformation ajoute une sémantique formelle aux diagrammes UML afin d'intégrer des notions mathématiques et rigoureuses dans le développement de logiciels.

Dans ce chapitre, nous présentons d'abord la sémantique des diagramme d'états-transitions dans la section 5.2.1. Ensuite, la section 5.2.2 définit les réseaux de Pétri. Notre processus de transformation de modèles est détaillé dans la section 5.3 et nous

terminons par une conclusion.

5.2 Exemple d'étude de cas

L'exemple présenté dans ce chapitre esquissera la démarche de transformation de modèles proposée dans le cadre de notre recherche. Il s'agira de transformer des diagramme d'états-transitions vers des réseau de Pétri. Dans ce qui suivra, nous détaillons les deux espaces : espace des machines d'états-transitions et l'espace des réseaux de Pétri.

5.2.1 Diagramme d'états-transitions

Les diagrammes d'états-transitions [72] fournissent des notions graphiques qui peuvent être utilisées pour modéliser le comportement dynamique des systèmes du point de vue des états et des transitions. Ils décrivent le comportement interne d'un objet à l'aide d'un automate à états finis c'est à dire ils permettent d'écrire le cycle de vie d'un objet sur une séquence d'états. Le diagramme d'états-transitions est le seul diagramme de la norme UML, qui offre une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché.
La figure 5.1 illustre un exemple simple de diagramme d'états-transitions présentant en fait le fonctionnement d'un télé-rupteur dans une maison. Ce diagramme possède deux états (Allumé et Éteint) et deux transitions correspondant au même événement : la pression sur un bouton d'éclairage domestique.

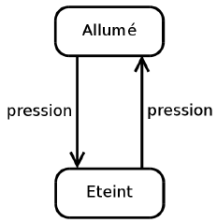

Figure 5.1: Un diagramme d'états-transitions simple.

5.2.1.1 État

Un état (state en anglais) définit un objet à un moment donné pendant lequel cet objet satisfait certaines conditions, exécute une activité ou attend un événement. Il se représente graphiquement dans un diagrammes d'états-transitions par un rectangle. Généralement,

un état peut être simple, composé, final, pseudo ou sous machine. Dans ce qui suit, nous essayons de définir ces types d'état.

5.2.1.2 État Simple

Un état simple (Simple state en anglais) présente un cas général de tous les types d'état ne possèdent pas de caractéristiques spécifiques. La figure 5.1 définit l'utilisation des états simples.

- **Règle de cohérence :**
  - R1 : Un état simple est un élément non décomposable.

5.2.1.3 État Final

Un état final (final state en anglais) (Figure 5.2) présente un cas particulier de l'état simple qui indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé. Graphiquement, il se définit par un cercle coloré en noire et blanc.

Figure 5.2: État final.

- **Règles de cohérence :**
  - R2 : Un état final ne peut pas avoir des transitions sortantes.
  - R3 : Un état final non décomposable.

5.2.1.4 État Composite

Un état composé (Composite state en anglais) possède des régions contenant chacune un ou plusieurs états et transitions. Ces états regroupent un ensemble de caractéristiques qui définissent sa sémantique. La figure 5.3 montre un exemple d'état composite.

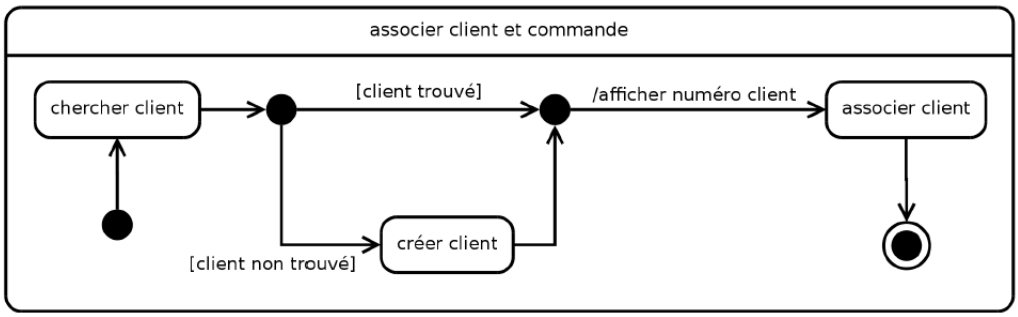

Figure 5.3: État composite.

- **Règles de cohérence :**
  - R4 : Un état composite contient au moins une région.
  - R5 : Un état composite orthogonal doit contenir au moins deux régions.

5.2.1.5 Pseudo-État

Un pseudo-état (Pseudo-state en anglais) est une abstraction qui regroupe différents types de sommets transitoires où chacun possède un ensemble de propriétés (Figure 5.4). Ils sont typiquement utilisés pour relier des transitions multiples.

1. **Initial :** Le pseudo-état initial (pareil en anglais) est un état qui définit le point de départ par défaut pour un diagramme d'états-transitions.
2. **Terminal :** Le pseudo-état terminal (terminate en anglais) est un état qui indique que l'exécution du diagramme d'états-transitions est terminé.
3. **Point d'entré :** Le pseudo-état de point d'entrée (entry point en anglais) permet de relier les transitions entrantes pour les sous-machines d'états.
4. **Point de sortie :** Le pseudo-état de point de sortie (exit point en anglais) permet de relier les transitions sortantes entre les sous machines d'états.
5. **Choix :** Le pseudo-état de choix (choice en anglais) permet de définir un choix de transitions.
6. **Union :** Le pseudo-état d'union ou de Jointure (join en anglais)permet de joindre les transitions.

7. **Bifurcation :** Le pseudo-état de bifurcation (fork en anglais)permet de bifurquer les transitions

8. **jonction :** Le pseudo-état de jonction (junction en anglais)permet de ponctionner les transitions.

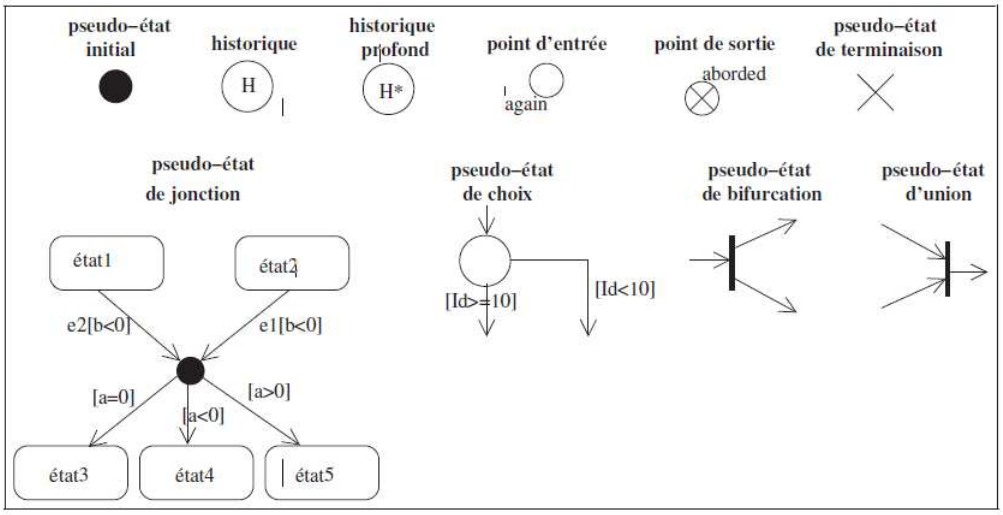

Figure 5.4: Les différents types de pseudo-état.

- **Règles de cohérence :**
  - R6 : Un pseudo-état initial ne doit pas être la cible d'une transition.
  - R7 : Un pseudo-état initial doit avoir au minimum une transition sortante.
  - R8 : Un pseudo-état initial ne doit pas avoir des conditions ni des événements.
  - R9 : Un pseudo-état terminal ne doit pas être la source d'une transition.
  - R10 : Un pseudo-état terminal doit avoir au minimum une transition entrante.
  - R11 : Dans une machine à états complète, un sommet de jonction (junction) doit posséder au moins une transition entrante et au moins une transition sortante.
  - R12 : Dans une machine à états complète, un sommet de choix (choice) doit posséder au moins une transition entrante et au moins une transition sortante.

- R13 : Un pseudo-état de bifurcation (fork) doit avoir au minimum une transition entrante et au moins deux transitions sortantes.
- R14 : Un pseudo-état d'union (join) doit avoir au minimum deux transitions entrantes et au moins une transition sortante.
- R15 : Les point de connexion de type entrée peuvent être définis par des pseudo-états de type point d'entrée.
- R16 : Les point de connexion de type sortie peuvent être définis par des pseudo-états de type point de sortie.

5.2.1.6 Sous-machine à états

Une sous-machine d'états (sub-state machine) est une machine à états définit l'insertion de la spécification d'une sous machine à états (une autre machine à états). Elle illustre l'idée de décomposition qui permet de factoriser des comportements communs et de les réutiliser. Sémantiquement, Une sous-machine à états est équivalente à un état composite.

- **Règles de cohérence :**
  - R17 : Seulement une sous-machine à états peut posséder des références de points de connexion (connection point reference en anglais).
  - R18 : Ces références de points de connexion peuvent utiliser deux types de pseudo-états : soit les points d'entrée ou les points de sortie.

5.2.1.7 Transition

La définition du changement d'un état à un autre se fait par l'intermédiaire de transitions. Une transition est toujours un lien entre deux sommets. Généralement, elle peut être définit par la syntaxe suivante :

[<événement>] ['[' <garde> ']'] ['/' <activité>]

1. **événement :** Un événement est une activité qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé.
2. **Condition de garde :** Une transition peut contenir une condition de garde (spécifiée par '[' <garde> ']' dans la syntaxe). Cette condition de garde est défini par une expression logique sur les attributs de l'objet, associé au diagramme d'états-transitions, ainsi que sur les paramètres de l'événement.

3. **Activité :** Si une transition se déclenche, alors son effet s'exécute (définit par '/' <activité> dans la syntaxe).

Les transitions peuvent être classifiées en trois catégories: des transitions internes, des transitions locales et des transitions externes.

- **Transitions internes :** Les transitions internes se produisent sans sortir ou sans entrer dans un état. Elles peuvent être définies dans un compartiment de leur état associé.
- **Transitions locales :** Les transitions locales se produisent entre deux états en respectant leur syntaxe. Graphiquement, elles sont définies par des flèches où elles ne sortent pas de l'état composite source.
- **Transitions externes :** Les transition externes se produisent aussi entre deux états en respectant leur syntaxe. Graphiquement, elles sont définies par des flèches, où elles sortent de l'état composite source.

Ensuite, nous définissons les règles de cohérence d'une transition.

- **Règles de cohérence :**
  - R19 : Les transitions sortantes d'un sommet de bifurcation (fork) ne doivent ni contenir le champ d'événement ni le champ de garde.
  - R20 : Les transitions entrantes du sommet d'union (Join) ne doivent ni contenir le champ d'événement ni le champ garde.
  - R21 : Les transitions sortantes d'un sommet de bifurcation (fork) doivent cibler des états et non des pseudo-états.
  - R22 : Les transitions entrantes du sommet d'union (Join) doivent cibler des états et non des pseudo-états.
  - R23 : Les transitions sortantes du sommet pseudo-état ne peuvent pas contenir des événements.

5.2.1.8 Région

Une région est une partie orthogonale d'un état composé, d'une sous-machine à états ou d'une machine à états. Elle contient des états et des transitions.

- **Règles de cohérence :**
  - R24 : Une région contient au maximum un pseudo-état initial.

– R25 : Si une machine à états contient une région donc l'attribut *IsHorthogonal* est faux sinon il est vrai.

– R26 : Une machine à états ne contient pas deux régions avec le même nom.

5.2.2 Réseau de Petri

Les Réseaux de Petri (ou RdP) ont été introduits pour la première fois par Carl Adam Petri dans sa thèse intitulée «Communication with Automata» en 1962 [77]. Ils sont des outils graphiques et mathématiques permettant de modéliser le comportement dynamique des systèmes réactifs et concurrents.

Un réseau de Petri est un graphe dirigé, connecté et biparti, où il est composé de deux types de nœuds (cf. Figure 5.5). Un nœud représente soit une place, soit une transition. Une place peut contenir des jetons, qui correspondent généralement aux ressources disponibles. Si un jeton est présent dans une place en entrée d'une transition, alors la transition est activée et peut donc s'exécuter. Une exécution d'une transition consomme un jeton dans chaque place en entrée et met un jeton dans chaque place en sortie. De nombreuses définitions s'accordent sur cette approche, nous proposons de présenter une définition formelle du réseau de Petri fournie par Peterson dans [76].

Réseau de Petri : *Un réseau de Petri est un 3-uplet 1 où : <S, T, W>*

- *S est un ensemble fini de places.*
- *T est un ensemble fini de transitions.*
- *S et T sont distincts, aucun objet ne peut être à la fois une place et une transition.*
- *W : N est un multi-ensemble (ou sac) d'arcs.*

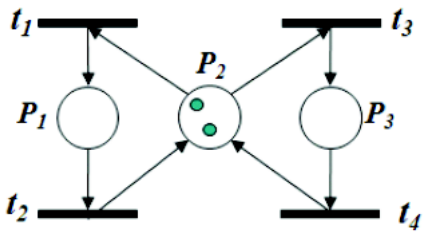

Figure 5.5: Exemple de réseau de Petri.

- **Les propriétés d'un réseau de Petri :**
  - R27 : Un arc ne peut pas connecter deux places ou deux transitions.
  - R28 : Un réseau de Petri ne peut pas contenir deux places avec le même nom.
  - R29 : Un réseau de Petri ne peut pas contenir deux transitions avec le même nom.

5.3 Approche IDM

Comme nous l'avons déjà mentionné, l'objectif principal de ce chapitre est d'expliciter le processus (cf. Figure 5.6) exploité dans le cadre de l'IDM comprenant les deux aspects principaux : la métamodélisation et la transformation de modèles.
La métamodélisation s'occupera de la définition des métamodèles et la génération des modèles conformes à leurs métamodèles. Tandis que le processus de transformation permettra de générer automatiquement un modèle à partir d'un programme de transformation codifié avec un langage de transformation.

Figure 5.6: Notre processus de transformation.

5.3.1 Métamodélisation

Généralement, la pratique de la méta-modélisation consiste à définir les métamodèles qui reflètent une partie de la sémantique des modèles. Les langages de métamodélisation offrent les concepts et relations élémentaires en termes desquels il est possible de définir les métamodèles qui peuvent être utilisés dans la spécification des règles de transformation de modèles. Dans ce qui suit (les sous sections 5.3.1.1 et 5.3.1.2) nous présentons de manière graphique les métamodèles source et cible de notre exemple d'étude de cas.

5.3.1.1 Spécification du métamodèle source

Dans cette partie, nous spécifions notre métamodèle de diagrammes d'états-transitions d'UML. La définition des métamodèles a été réalisée sous l'environnement EMF (cf. Figure 5.7).

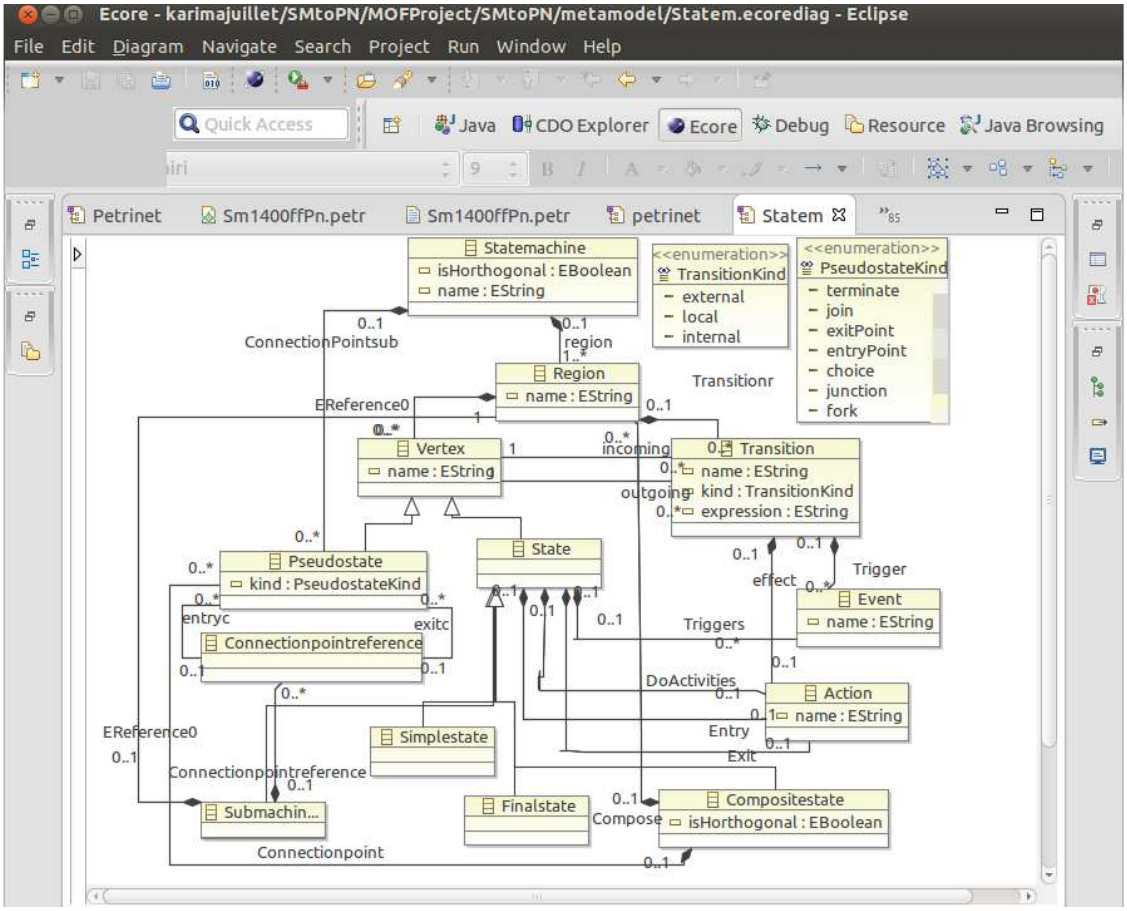

Figure 5.7: Métamodèle du diagramme d'états-transitions.

5.3.1.2 Spécification du métamodèle cible

Nous présentons maintenant le métamodèle cible qui définit la structure statique des modèles de réseau de Petri qui complète la sémantique des diagrammes d'UML.

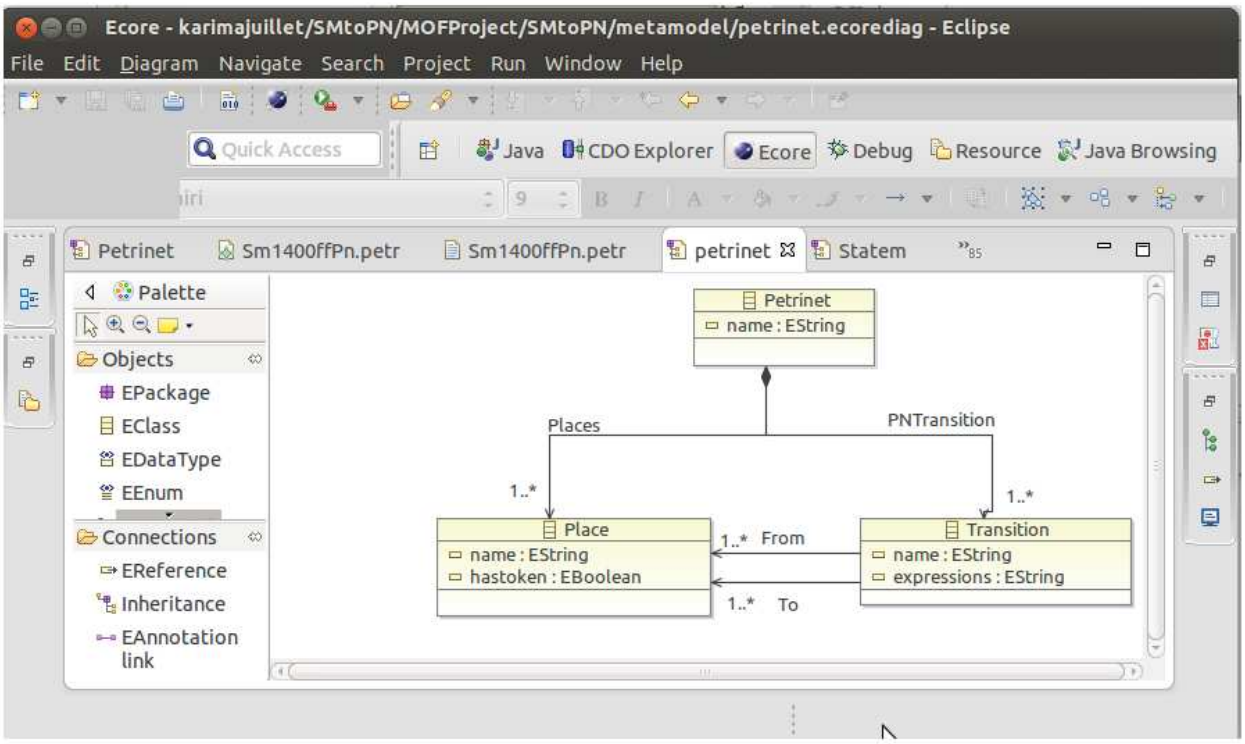

Figure 5.8: Métamodèle du réseau de Petri.

5.3.2 Processus de transformation de modèles

La transformation de modèles est également un paradigme important afin de permettre la génération de code, l'ingénierie inverse, la migration de modèles ou la translation de modèles. La transformation de modèles se base sur la définition des règles de transformation où une bonne transformation de modèles dépend d'une bonne définition des règles de transformation. Dans cette sous section, nous présentons un ensemble de règles de transformation de Choppy [24] que nous avons étendu et amélioré. Ces règles de correspondance exprime la sémantique suivante :

1. **État simple :** La transformation de l'état simple se fait directement vers une place. La figure suivante présente graphiquement cette transformation.

Figure 5.9: Transformation d'un état simple.

2. **État final :** La même chose se réalise pour l'état final, donc il est translaté en une place. Nous présentons cette figure qui traduise cette transformation.

Figure 5.10: Transformation d'un état final.

3. **État composite :** L'état composite se traduit vers les réseaux de Petri par la transformation de ces sous états et de ces sous transitions. cette figure, présente les possibilités de transformer l'état composite dans le réseau de Petri correspondant.

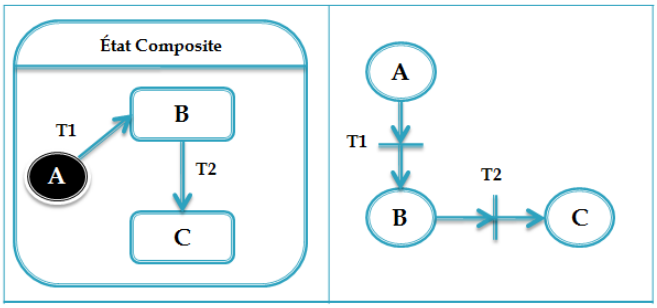

Figure 5.11: Transformation d'un état composite.

4. **Sous machine d'états :** Puisque les sous machine d'états sont équivalente aux états composite sémantiquement, donc nous proposons d'appliquer le même principe de transformation des états composites aux sous machines d'états.

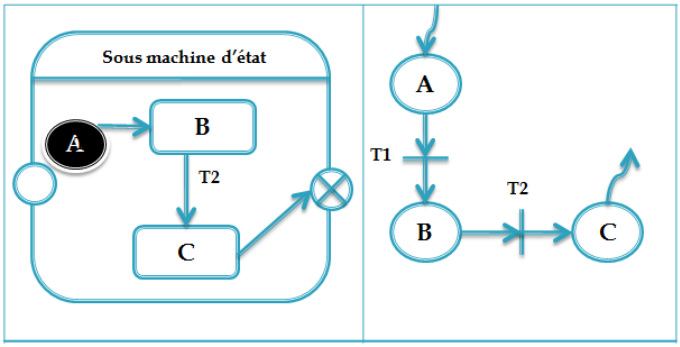

Figure 5.12: Transformation d'une Sous machine d'états.

5. **Pseudo état :** La transformation d'un pseudo état se change selon son propre type.

   (a) ***Pseudo-état initial :*** Un pseudo état initial est transformé de la même manière que l'état simple, mais sans avoir des transitions internes.

Figure 5.13: Transformation d'un Pseudo-état initial.

   (b) ***Pseudo-état terminal :*** Un pseudo état terminal est translaté en une place dans le réseau de Petri correspondant.

Figure 5.14: Transformation d'un pseudo-état terminal.

   Les pseudo états de type jointure, choix, point d'entrée, point de sortie, bifurcation et jonction sont supprimés en gardant leurs sens par la transformation de leurs transitions.

6. **Transition simple :** Tout d'abord, une transition en UML simple possède comme une source et une destination des état simples. Donc, elle peut être transformée

vers une transition de réseau de Petri en gardant les mêmes sources et les mêmes destinations.

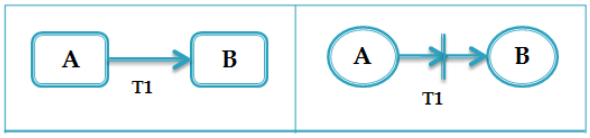

Figure 5.15: Transformation d'une Transition simple.

7. **Transition de bifurcation :** Une transition est dite de bifurcation si elle possède une destination de type pseudo-état de bifurcation. Cette transition est transformée vers une transition dans le réseau de Petri en gardant ses sources et en changeant ses destinations par les destinations de ses transitions sortantes qui possèdent comme source le pseudo état de bifurcation.

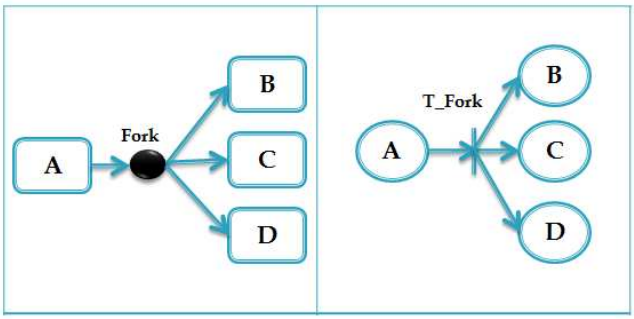

Figure 5.16: Transformation d'une Transition de bifurcation.

8. **Transition de jointure :** Une transition est dite de type jointure si elle contient une source de type pseudo-état de jointure. Cette transition est donc transformée vers une transition dans le réseau de Petri correspondant en gardant ses destinations et en changeant ses sources par les sources de ses transitions qui ont comme une destination le pseudo état de jointure.

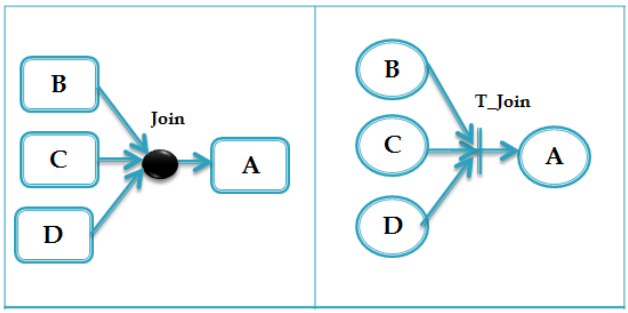

Figure 5.17: Transformation d'une Transition de jointure.

9. **Transition de jonction :** Une transition est dite de jonction si elle a une source de type pseudo-état de jonction. Cette transition peut être transformée vers une transition dans le réseau de Petri en gardant ses destinations et en changeant ses sources par les sources de ses transitions qui contiennent comme destination le pseudo état de jonction.

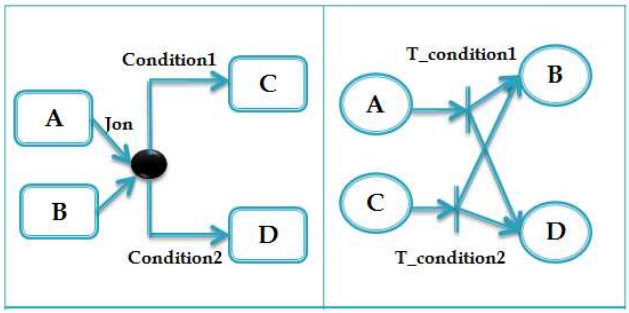

Figure 5.18: Transformation d'une Transition de jonction.

10. **Transition de choix :** Une transition est dite de choix si elle possède une source de type pseudo-état de choix. Cette transition est donc transformée vers une transition de réseau de Petri en gardant ses destinations et en changeant ces sources par les sources des transitions qui ont comme destination le pseudo état de choix.

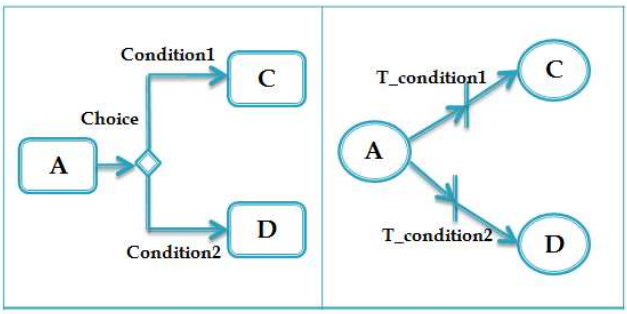

Figure 5.19: Transformation d'une Transition de choix.

11. **Transition de point de sortie :** Une transition est dite de point de sortie si elle a une source de type pseudo-état de point de sortie. Cette transition est transformée en une transition dans le réseau de Petri en gardant ses destinations et en changeant ses sources par les sources de ses transitions qui possèdent comme destination le pseudo état de point de sortie.

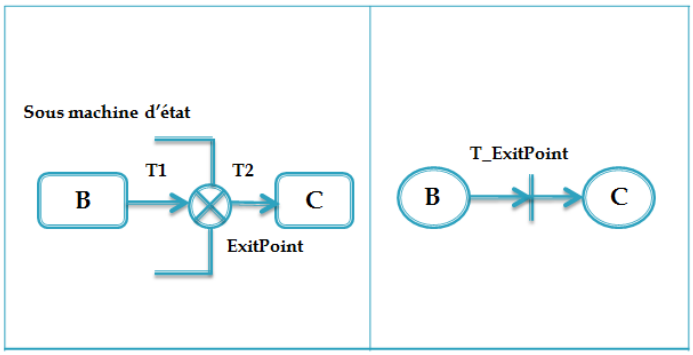

Figure 5.20: Transformation d'une Transition de point de sortie.

12. **Transition de point d'entrée :** Une transition est dite de point d'entrée si elle possède une destination de type pseudo-état de point d'entrée. Cette transition est transformée vers une transition dans le réseau de Petri en gardant ses sources et en changeant ses destinations par les destinations des transitions qui ont comme source le pseudo état de point d'entrée.

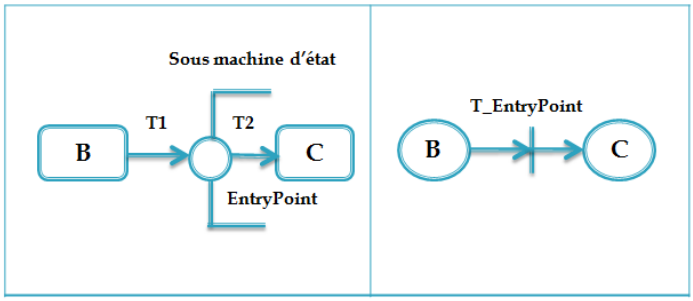

Figure 5.21: Transformation d'une Transition de point d'entrée.

13. **Transition avec une source composite :** Chaque transition qui possède comme source un état composite et un état simple comme destination peut être transformée soit par une transition de réseau de Petri en gardant sa destination et en changent sa source par l'état initial de cet état composite, ou par des transitions qui possèdent la même destination (la transformation de l'état simple) et leur sources sont les transformations des sous états de l'état composite.

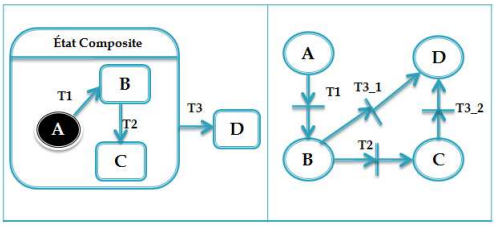

Figure 5.22: Transformation d'une transition avec une source composite : cas 1.

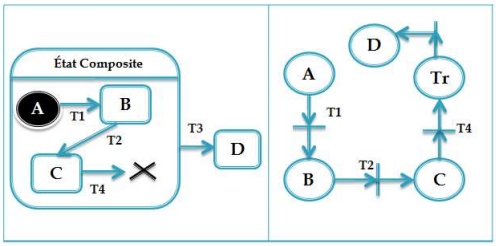

Figure 5.23: Transformation d'une transition avec une source composite : cas 2.

14. **Transition avec une destination composite :** Chaque transition qui possède comme destination un état composite et un état simple comme source peut être transformée soit par une transition de réseau de Petri en gardant sa source et en changeant sa destination par l'état initial de cet état composite, ou par des transitions qui possèdent la même source (la transformation de l'état simple) et leur destinations sont les transformations des sous états de l'état composite.

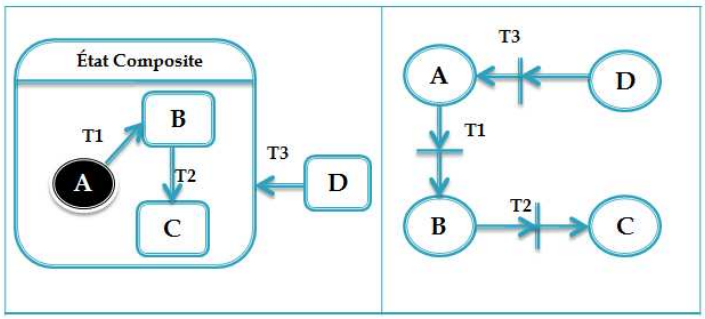

Figure 5.24: Transformation d'une transition avec une destination composite.

15. **Transition avec une source de type sous machine d'état :** Chaque transition qui possède comme une source une sous machine d'état et un état simple comme une destination peut être transformée soit par une transition de réseau de Petri en gardant sa destination et en changeant sa source par l'état initial de cette sous machine d'état, ou par des transitions qui possèdent la même destination (la transformation de l'état simple) et leur sources sont les transformations des sous états de la sous machine d'états.

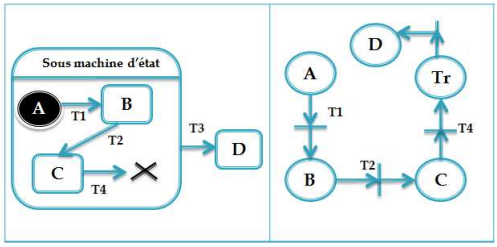

Figure 5.25: Transformation d'une Transition avec une source de type sous machine d'état : cas 1.

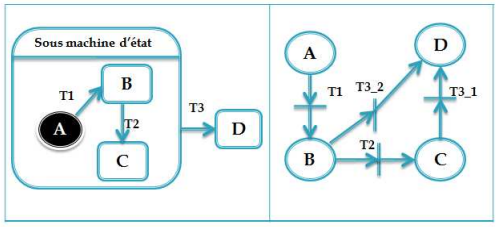

Figure 5.26: Transformation d'une transition avec une source de type sous machine d'état : cas 2.

16. **Transition avec une destination de type sous machine d'états :** Chaque transition qui possède comme destination une sous machine d'états et un état simple comme source peut être transformée soit par une transition de réseau de Petri en gardant sa source et en changeant sa destination par l'état initial de cette sous machine d'états, ou par des transitions qui possèdent la même source (la transformation de l'état simple) et leur destinations sont les transformations des sous états de la sous machine d'états.

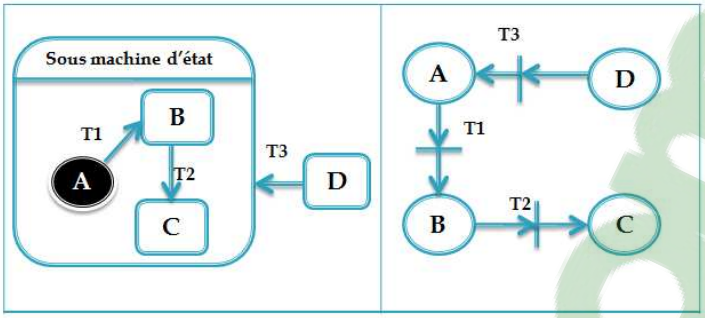

Figure 5.27: Transformation d'une transition avec une destination de type sous machine d'état.

17. **Transition d'un état initial :** Une transition est dite d'un état initial si elle possède une source de type pseudo-état initial. Cette transition est donc transformée en une transition dans le réseau de Petri en gardant ses destinations et ses sources.

Figure 5.28: Transformation d'une Transition d'un état initial.

18. **Transition d'un état final :** Une transition est dite d'un état final si elle possède une destination de type pseudo-état final. Cette transition est donc transformé en une transition dans le réseau de Petri en gardant ses destinations et ses sources.

Figure 5.29: Transformation d'une transition d'un état final.

19. **Transition interne :** Une transition est dite interne si un état simple possède une activité interne (Do, Entry, Exit). Donc ces activités peuvent être transformés en des transitions dans le réseau de Petri qui possèdent la même source et la même destination (état simple).

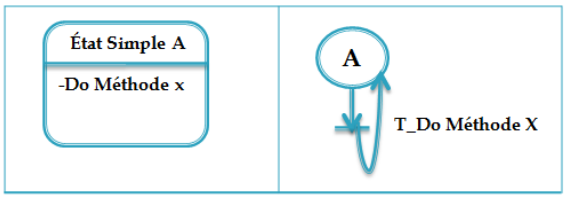

Figure 5.30: Transformation d'une transition interne.

Les règles de transformation sont traduites ensuite dans un langage de transformation tel que le langage QVT-op. Nous introduisons ci-dessous deux règles de transformation dans le formalisme du langage QVT-op. Une de ces règles présente la spécification d'une transition de point d'entrée. Nous rappelons que cette règle peut être transformée dans un modèle de réseau de Petri vers une transition qui porte les mêmes sources de la transition en UML et les destinations de la transition sortante de ce point de sortie. La deuxième règle présente la spécification d'une transition de point de sortie. Nous rappelons que le principe de cette règle peut être définit comme la règle précédente sauf que la modification se fait pour les sources (comme les destinations de la règle précédente).

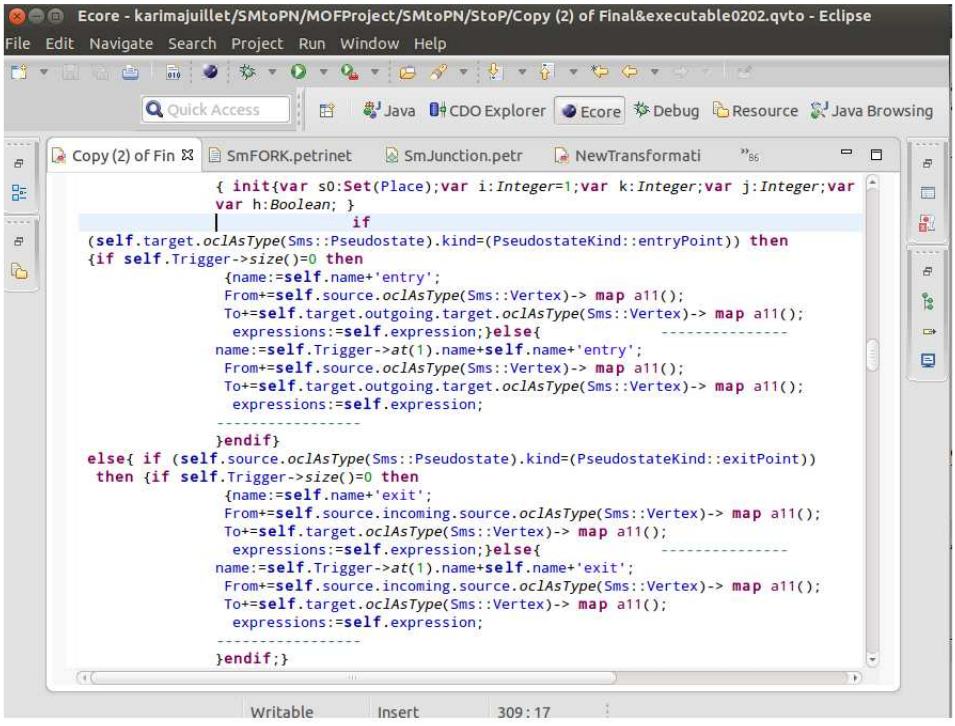

Figure 5.31: règles de transformation en QVT-op.

5.4 Exécution de la transformation

Dans cette section, nous présentons deux cas pour définir l'exécution de la transformation de diagramme d'états-transitions vers les réseaux de Petri. Dans le premier cas, nous utilisons le diagramme d'état-transitions du système ATM (Automated Teller Machine) qui illustre l'utilisation d'un point de sortie au niveau d'une sous machine d'états. Le deuxième cas présente le diagrammes d'états-transitions de la montre en prenant en compte tous ses comportements (la gestion de l'affichage, la gestion de l'alarme et la gestion de l'éclairage).

5.4.1 Exemple : Système ATM

ATM [72] (Automated Teller Machine) est un système permettant aux clients d'effectuer différentes transactions bancaires en libre-service. Il permet de faire des retraits, accepte des dépôts en liquide ou par chèque, ordonne des transferts de fonds, imprime des mises à jour de carnets, augmente le montant d'une carte d'appel téléphonique et même vend des timbres-poste.
Dans cette sous section, nous illustrons la définition du système ATM (Automated Teller Machine) avec le formalisme UML par un diagramme d'état-transitions (Figure 5.32).

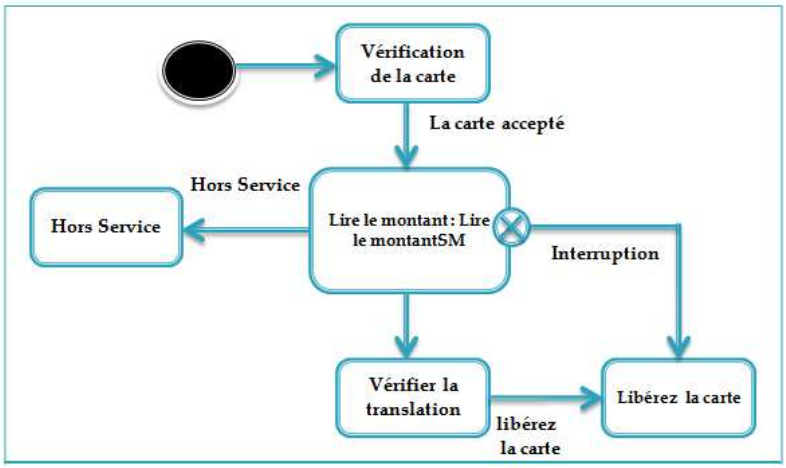

Figure 5.32: Diagramme d'état-transitions du systèmr ATM.

Ensuite nous présentons la transformation de cet exemple vers un modèle dans le réseau de Petri (la figure 5.33).

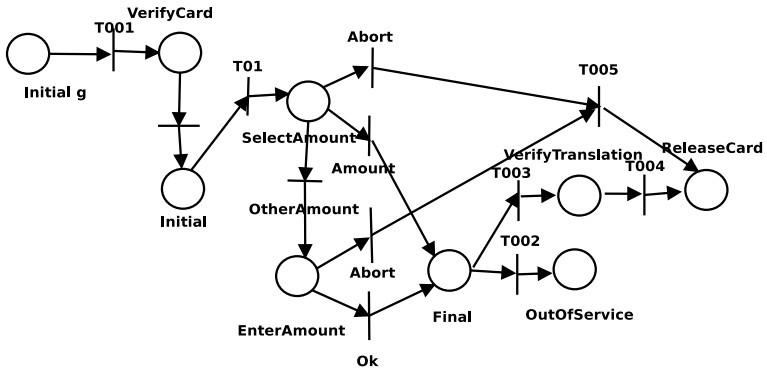

Figure 5.33: Réseau de Petri du systèmr ATM.

5.4.2 Exemple : Comportements de la montre

Nous présentons ici la transformation de diagrammes d'états-transitions de la montre [80] avec trois régions concurrentes vers le réseau de Petri. La figure 5.34 définit les comportements de la montre dans le formalisme UML.

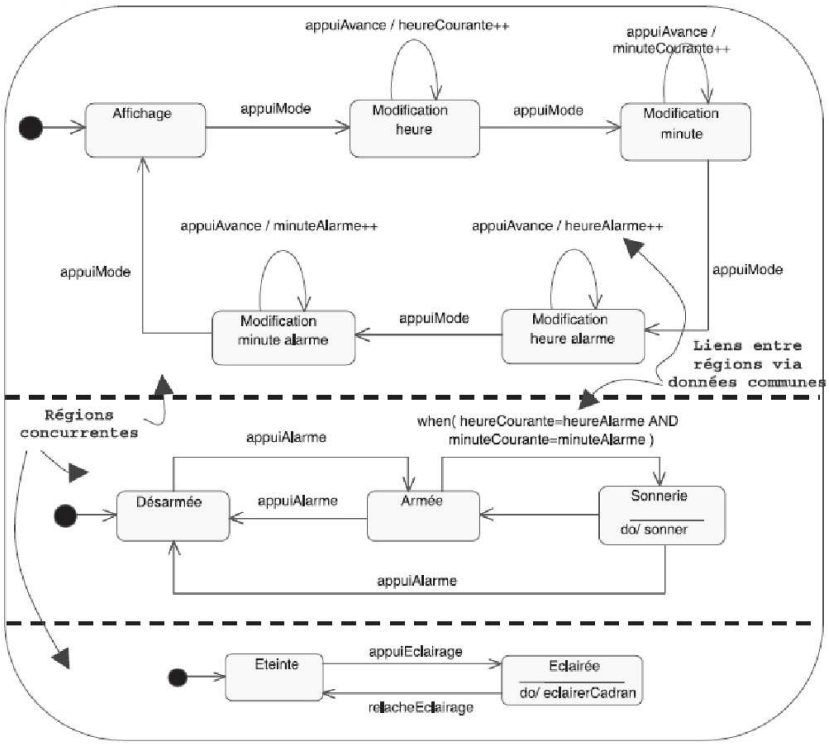

Figure 5.34: Diagramme d'état-transitions de la montre.

Après l'exécution des règles de transformation, on obtient un modèle de type réseau de Petri (Figure 5.35).

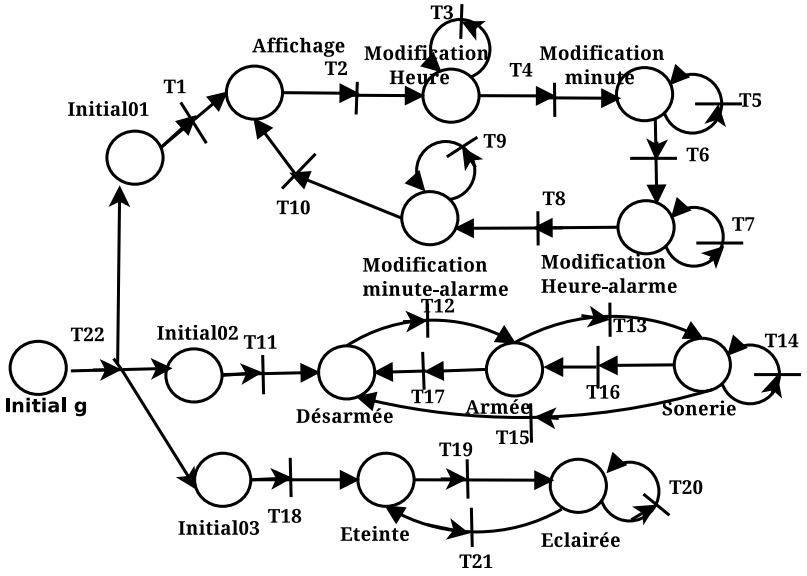

Figure 5.35: Réseau de Petri de la montre.

5.5 Conclusion

Dans le cadre de ce chapitre, nous proposons de transformer les diagrammes d'états-transitions vers les réseaux de Petri en utilisant des outils de l'IDM comme Ecore et QVT-op.
Le langage Ecore permet de définir des métamodèles et des modèles. Le le langage QVT-op qui combine la structure impératif et la structure déclaratif assure la spécification des transformations dont l'exécution générera un modèle conforme à son métamodèle.
La problématique clé de l'IDM consiste à pouvoir vérifier et valider la transformation de modèles en utilisant des outils formels. L'objectif du chapitre suivant est de décrire notre méthode de résolution pour cette problématique.

Chapitre 6

Mise en œuvre de la vérification et de la validation

6.1 Introduction

L'objectif de ce chapitre consiste à appliquer des techniques de vérification et de validation (V&V) pour les transformations de modèles. Dans le chapitre précédent, nous avons détaillé le processus de transformation à travers un exemple d'étude de cas en mettant l'accent sur les outils et les langages utilisés.

Donc, nous proposons dans ce cadre, une approche qui prend en charge la vérification et la validation de transformation de modèles en exploitant pleinement les avantages de techniques formelles.
Ce chapitre est structuré comme suit : la section 6.2 présente une vue générale de notre approche. Les sections 6.3 et 6.4 , les plus importantes en taille, présentent notre proposition en détaillant le processus de vérification et de validation dans notre approche de transformation de modèles. Enfin, une conclusion termine ce chapitre.

6.2 Vue générale sur notre approche

La transformation de modèles occupe une place très importante dans toute démarche d'ingénierie dirigé par les modèles et porte un ensemble d'outils et des langages qui facilitent son cycle de développement mais elle n'intègre pas des outils de vérification ou de validation formelle. Pour cela, nous nous sommes intéressées à la technique formelle de preuve pour assurer une bonne production des modèles finaux et une analyse statique pour vérifier certains propriétés des modèles selon leurs métamodèles par le

langage formel OCL. Donc, notre approche est subdivisé en trois étapes principales : ***La première étape*** concerne l'implémentation du processus de transformation qui offre une sémantique formelle aux diagrammes d'UML. Pour ***la deuxième étape***, nous utilisons le langage OCL dans le but de vérifier la conformité des modèles à leurs métamodèles. ***La troisième étape*** propose d'appliquer une technique formelle comme l'assistant de preuve Coq afin de valider notre transformation de modèles.

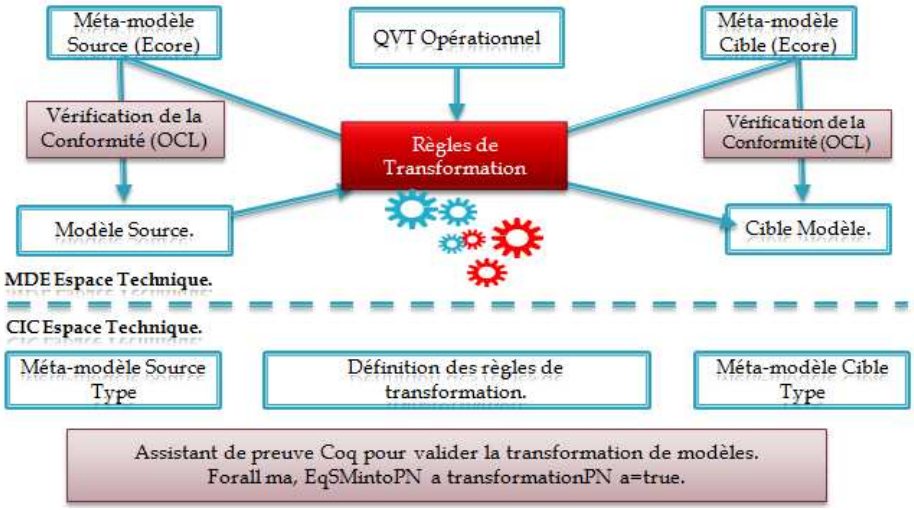

Figure 6.1: Processu de Transformation.

La figure 6.1 illustre d'une façon générale l'idée de notre approche [9]. Dans ce qui suit, nous détaillons les étapes de mise en exploitation de la transformation de modèles, de la vérification et de la validation de cette transformation.

6.3 Mise en œuvre de la vérification

Dans cette section, nous nous intéressons à la vérification de la conformité des modèles. Ce type de vérification a été appliqué dans de nombreux travaux et dans des contextes différents. Comme nous l'avons déjà rappelé auparavant, la mise en œuvre de la vérification concerne l'étude d'un ensemble de propriétés qui décrit la sémantique des modèles et l'implémentation d'un outil permettant de vérifier automatiquement la conformité des modèles à leurs métamodèles. A cet effet, notre choix s'est porté sur l'utilisation du langage OCL pour spécifier formellement les propriétés décrivant la sémantique des métamodèles. Dans le chapitre précédent, nous avons présenté un échantillon de propriétés pour les métamodèles de l'exemple d'étude de cas. L'implémentation de ces propriétés

sous forme de contraintes OCL et leur exécution dans l'environnement EMF/Eclipse permettent la prise en charge de la vérification de la conformité des modèles par rapport à leurs métamodèles.
Pour illustrer ceci, nous proposons les spécifications de quelquels propriétés du chapitre précédent avec le formalisme du langage OCL. Ceci concernera certaines proriétés du métamodèles source et cible de l'exemple d'étude de cas.subdivisé se travail en deux sous sections, la première s'intéresse la formalisation de propriétés de modèle source en langage OCL. Tandis que la deuxième, concerne l'écriture des propriétés de modèle cible aussi en langage OCL.

6.3.1 Spécification des propriétés du métamodèle source

Les règles de cohérence présentées dans le chapitre précédent sont traduites en contraintes du langage OCL. Les contraintes suivantes sont des exemples.
Afin de vérifier la conformité des modèles à leurs métamodèles, les contraintes OCL sont implémentées avec le formalisme OclInEcore équivalent du langage OCL dans l'environnement EMF/Eclipse. Le code suivant illustre une représentation des règles R 21, R 19 et R 13.

```
invariant a1: source.oclIsKindOf(Pseudostate) and source.kind = PseudostateKind::fork implies target.oclIsKindOf(State);
invariant a2: source.oclIsKindOf(Pseudostate) and source.kind = PseudostateKind::fork implies guard->isEmpty() and Trigger->isEmpty();
invariant a3: self.kind = PseudostateKind::fork implies self.incoming->size() = 1 and self.outgoing->size() >= 2;
```

6.3.2 Spécification des propriétés du métamodèle cible

La vérification permet d'assurer que le modèle généré est conforme syntaxiquement et sémantiquement à son métamodèles. L'implémentation des règles de cohérence de la section 5.2 permettent d'atteindre cet objectif. Ce qui suit, en présente quelques règles de cohérence du métamodèle cible de notre exemple d'étude de cas.

```
invariant a1: self.PNTransition->forAll(s1 : Transition | self.PNTransition->forAll(s2 : Transition | s1.name = s2.name and s1.From = s2.From and s1.To = s2.To and s1.expressions = s2.expressions implies s1 = s2));
invariant a2: self.Places->forAll(s1 : Place | self.Places->forAll(s2 : Place | s1.name = s2.name implies s1 = s2));
```

6.4 Mise en œuvre de la validation

Plusieurs techniques de validation sont applicables sur les transformations de modèles. Elles peuvent être classifiées en deux grandes catégories: les techniques formelles et les techniques semi-formelles. Dans cette section, nous proposons d'appliquer une technique de validation formelle qui s'étale tout au long du cycle de vie d'un produit. Elle se base sur des aspects mathématiques afin d'assurer la satisfaction des exigences particulières. Cette technique *"assistant de preuve Coq"* se fonde sur l'utilisation de la preuve et d'un langage formel qui s'appelle Gallina. Pour valider une transformation de modèles en utilisant cet assistant, nous suivons les trois étapes suivantes : *La première étape* concerne la spécification de processus de transformation qui englobe la définition des modèles et leurs métamodèles, plus la spécification de règles de transformation. Alors que *la deuxième étape* s'intéresse la formalisation de théorème qui exprime les exigences particulières de la transformation. *La troisième étape* porte sur la preuve de théorème qui se base sur l'utilisation d'un ensemble de tactiques.

6.4.1 Spécification en langage Gallina

Gallina est un langage de spécification de l'assistant de preuve Coq. Il permet de définir: des types de données structurés, des fonctions qui peuvent être non récursives ou récursives sur la structure des données et des formules de calcul des prédicats d'ordre supérieur. Dans les sous sections suivantes, nous spécifions les modèles et leurs métamodèles de l'exemple d'étude cas en langage Gallina en se basant sur cet ensemble de règles qui facilitent le passage à partir du langage Ecore vers ce langage :

1. ***Type de données et Énumérations :*** Ecore supporte les types de données primitifs par exemple: String, Boolean, Integer et Double. Dans le système Coq, il est possible d'avoir ces types de données primitifs via les librairies.

2. ***Classes, Attributs et Relations :*** Dans Ecore une classe contient des attributs et des relations. Chaque attribut et relation possède un nom, une multiplicité et un type. Chaque multiplicité a une valeur inférieure et une valeur supérieure. la présentation suivante illustre la spécification de la multiplicité dans le système Coq selon Calegarie en changeant au la présentation des métamodèles et la définition de modèles (une correction de la définition de métamodèles). Nous présentons les relation de Calegari [19] que nous avons étendu et amélioré par des fonctions dans le système Coq sauf quelques relations.
   [1-1]— attribut : string.
   [1-0]— relation 01 : option Element.

[0-*]— relation 02 : list Element.

Pour présenter une classe dans le système Coq, nous utilisons les définitions inductives. Ce code illustre comment on peut définir une classe sous le système Coq.

```
Inductive Petrinet : Set := |build_Petrinet (name:string) (Places: list Place)
with Place: Set:= |build_Place (name:string) (tokenNb:nat).
```

6.4.1.1 Spécification des métamodèles source et cible

D'après les règles de passage vers le système Coq qui sont présentées ci-dessus, nous définissons maintenant le métamodèle source en langage Gallina pour faciliter la preuve dans ce système.

```
Inductive Statemachine : Set := |build_Statemachine (name:string) (isHorthogonal:bool) (region : list Region)
with Region : Set:= |build_Region (name:string) (Transitionr: list Transition) (Container:list Vertex)
with Transition : Set:= |build_Transition (name:string) (source: Vertex) (target: Vertex) (Condition:string) (Trigger: list Event) (effect: option Action) (kind:TransitionKind)
with Vertex : Set:= |build_Vertex (SubPseudostate: option Pseudostate) (SubState: option State) (abstract:bool:=true)
with Compositestate : Set:= |build_Compositestate (isHorthogonal: bool) (abstract1:bool:=false) (Connectionpoint: list Pseudostate) (Compose: list Region)(name: string)
with TransitionKind: Set:= |local:TransitionKind
|internal: TransitionKind
|external:TransitionKind
with Pseudostate: Set:= |Build_Pseudostate (kind:PseudostateKind) (abstract:bool:=false) (name: string)
with Simplestate: Set:= |build_Simplestate (abstract1:bool:=false) (name: string)
with Finalstate: Set:= build_Finalstate (abstract1:bool:=false) (name: string)
with Submachinestate : Set:= |build_Submachinestate (Connectionpointreference: list Connectionpointreference) (abstract1:bool:=false) (containers: list Region) (name: string)
with State :Set:= |build_State (subsimplestate: option Simplestate) (subfinalstate: option Finalstate) (subcompositestate: option Compositestate) (subsubmachinestate: option Submachinestate) (abstract:bool:=false) (abstract1:bool:=true) (Entry: Action) (Exit: Action) (DoActives: Action) (Triggers:list Event)
```

```
with Action : Set:= |build_Action (name:string)
with Event : Set:= |build_Event (name:string)
with PseudostateKind : Set:= |terminate : PseudostateKind |join : PseudostateKind
|exitPoint: PseudostateKind |entryPoint:PseudostateKind |fork: PseudostateKind |choice:
PseudostateKind |junction :PseudostateKind |initial : PseudostateKind
with Connectionpointreference:Set:= |build_Connectionpointreference (entryc: list
Pseudostate)(exitc: list Pseudostate).
```

Le même principe reste valable pour la spécification de métamodèle cible (réseau Petri) en langage Gallina. Le code suivant présente la définition du métamodèle cible dans le système Coq.

```
Inductive Petrinet : Set := |build_Petrinet (name:string) (PNTransition: list Transi-
tionq) (Places: list Place)
with Place: Set:= |build_Place (name:string) (tokenNb:Z)
with Transitionq : Set:= |build_Transitionq (name:string) (Condition:string) (Fron: list
Place) (To:list Place).
```

6.4.1.2 Spécification des modèles source et cible

Nous présentons ici, les modèles en langage Gallina. Le Code suivant illustre le modèle en entrée dans le système Coq où nous utilisons `Definition` pour définir l'instanciation d'un métamodèle.

```
Definition Event1 : Event:= build_Event "amount".
Definition Event2 : Event:= build_Event "otheramount".
Definition Event3 : Event:= build_Event "abort".
Definition Event4 : Event:= build_Event "ok".
Definition Event5 : Event:= build_Event "accepltcard".
Definition Event6 : Event:= build_Event "outservice".
Definition Event7 : Event:= build_Event "releasecard".
Definition Sim1: Simplestate := build_Simplestate "selectamount".
Definition S1: State:= build_State (Some Sim1) (None) (None) None None None None .
Definition t1 : Vertex:=build_Vertex None (Some S1).
Definition Sim2: Simplestate := build_Simplestate "enteramount".
Definition S2: State:= build_State (Some Sim2) (None) (None) None None None None .
Definition t2 : Vertex:=build_Vertex None (Some S2).
```

Definition Sim3: Finalstate := build_Finalstate "Final01".
Definition S3: State:= build_State (None)(Some Sim3) (None) None None None None .
Definition t3 : Vertex:=build_Vertex None (Some S3).
Definition PSKin: PseudostateKind := initial.
Definition S4: Pseudostate := Build_Pseudostate (PSKin) "4".
Definition t4 : Vertex:=build_Vertex (Some S4) None .
Definition S5: Pseudostate := Build_Pseudostate (PSKin) "5".
Definition t5 : Vertex:=build_Vertex (Some S5) None .
Definition Sim6: Simplestate := build_Simplestate "verifycard".
Definition S6: State:= build_State (Some Sim6) (None) (None) None None None None .
Definition t6 : Vertex:=build_Vertex None (Some S6).
Definition Sim7: Simplestate := build_Simplestate "outservice".
Definition S7: State:= build_State (Some Sim7) (None) (None) None None None None .
Definition t7 : Vertex:=build_Vertex None (Some S7).
Definition Sim8: Simplestate := build_Simplestate "verifytransaction".
Definition S8: State:= build_State (Some Sim8) (None) (None) None None None None .
Definition t8 : Vertex:=build_Vertex None (Some S8).
Definition Sim9: Simplestate := build_Simplestate "releasecard".
Definition S9: State:= build_State (Some Sim9) (None) (None) None None None None .
Definition t9 : Vertex:=build_Vertex None (Some S9).
Definition Exit1: PseudostateKind := exitPoint.
Definition S10: Pseudostate := Build_Pseudostate (Exit1) "10".
Definition t10 : Vertex:=build_Vertex (Some S10) None .
Definition Tr1 : Transition:=build_Transition "01" t4 t1 " " (nil) None local.
DefinitionTr2 : Transition:=build_Transition "02" t1 t3 " " (Event1::nil) None local.
Definition Tr3 : Transition:=build_Transition "03" t2 t3 " " (Event4::nil) None local.
Definition Tr4 : Transition:=build_Transition "04" t1 t2 " " (Event2::nil) None local.
Definition Tr5 : Transition:=build_Transition "05" t1 t10 " " (Event3::nil) None local.
Definition Tr6 : Transition:=build_Transition "06" t2 t10 " " (Event3::nil) None local.
Definition Tr7 : Transition:=build_Transition "07" t5 t6 " " (nil) None local.
Definition r1 : Region:= build_Region "regio01" (Tr1::Tr2::Tr3::Tr4::Tr5::Tr6::nil) (t1::t2::t3::t4::nil).
Definition Interface: Connectionpointreference:=build_Connectionpointreference (S10::nil) nil.
Definition Sim11: Submachinestate := build_Submachinestate (Interface::nil) (r1::nil) "Sub".
Definition S11: State:= build_State (None) (None) None (Some Sim11)None None None (nil).

```
Definition t11 : Vertex:=build_Vertex None (Some S11).
Definition Tr8 : Transition:=build_Transition "08" t6 t11 " " (Event5::nil) None local.
Definition Tr9 : Transition:=build_Transition "09" t11 t7 " " (Event6::nil) None local.
Definition Tr10 : Transition:=build_Transition "10" t11 t8 " " nil None local.
Definition Tr11 : Transition:=build_Transition "11" t10 t9 " " (Event3::nil) None local.
Definition Tr12 : Transition:=build_Transition "12" t8 t9 " " (Event7::nil) None local.
Definition r2 : Region:= build_Region "region02" (Tr7::Tr11::Tr12::Tr10::Tr9::Tr8::nil)
(t5::t6::t7::t8::t9::t10::t11::nil).
Definition Statemachine1 : Statemachine:= build_Statemachine "Machine01" false
(r2::nil).
```

Maintenant, nous illustrons le code du modèle cible avec le langage Gallina.

```
= build_Petrinet "Machine01"
(build_Transitionq "01" " " (build_Place "4" 0 :: nil) (build_Place "selectamount" 0 ::
nil)
:: build_Transitionq "02" " " (build_Place "selectamount" 0 :: nil)(build_Place "Final01"
0 :: nil)
:: build_Transitionq "03" " "(build_Place "enteramount" 0 :: nil)(build_Place "Final01"
0 :: nil)
:: build_Transitionq "04" " " (build_Place "selectamount" 0 :: nil) (build_Place "enter-
amount" 0 :: nil)
:: build_Transitionq "07" " " (build_Place "5" 0 :: nil) (build_Place "verifycard" 0 :: nil)
:: build_Transitionq "08" " " (build_Place "verifycard" 0 :: nil) (build_Place "04" 0 ::
nil)
:: build_Transitionq "09" " " (build_Place "Final01" 0 :: nil) (build_Place "outservice" 0
:: nil)
:: build_Transitionq "10" " " (build_Place "Final01" 0 :: nil) (build_Place "verifytrans-
action" 0 :: nil)
:: build_Transitionq "11" " " (build_Place "selectamount" 0 ::build_Place "enteramount"
0:: nil) (build_Place "releasecard" 0 :: nil)
:: build_Transitionq "12" " " (build_Place "verifytransaction" 0 :: nil) (build_Place "re-
leasecard" 0 :: nil) :: nil)
(build_Place "5" 0 :: build_Place "verifycard" 0 :: build_Place "outservice" 0 ::
build_Place "verifytransaction" 0 :: build_Place "releasecard" 0 :: build_Place "selec-
tamount" 0 :: build_Place "enteramount" 0 :: build_Place "Final01" 0 :: build_Place "4"
0 :: nil) : Petrinet
```

6.4.1.3 Spécification des règles de transformation

L'exemple d'étude de cas traite La translation d'un diagramme d'états-transitions vers un réseau de Petri en se basant sur un ensemble de règles de transformation établies entre leurs métamodèles source et cible. Dans le cas de nos expérimentations, nous avons défini ces règles de transformation dans le système de preuve Coq afin de spécifier, vérifier et prouver complètement le processus de transformation de modèles. Le code suivant présente une règle de transformation qui translate une transition permettant de relier deux états simples en langage Gallina.

```
Definition TransitionToTransitionqSTS (a:Transition) : Transitionq :=
build_Transitionq (Transition_name (a)) (Transition_condition (a)) ( match (Ver-
tex_SubState(Transition_source(a))) with
|None ⇒ nil
|Some q ⇒ match State_Subsimplestate (q) with
|None ⇒ nil
|Some s ⇒ SimpleSToPlace(s)::nil end end)
( match (Vertex_SubState(Transition_target(a))) with
|None ⇒ nil
|Some q ⇒ match State_Subsimplestate (q) with
|None ⇒ nil
|Some s ⇒ SimpleSToPlace(s)::nil end end).
```

Le code suivant présente une autre transformation qui peut être utilisé au niveau de la règle de transformation précédente comme il peut être classifié parmi les règles de transformation qui translate un état simple dans le diagramme d'états-transitions en une place dans le réseau de Petri.

```
Definition SimpleSToPlace (a:Simplestate) : Place := build_Place (Simplestate_name
(a)) (0).
```

6.4.2 Formalisation de théorème

Dans la vérification basée sur la preuve de théorème, nous supposons que les spécifications des règles de transformation sont correctes et nous essayons de démontrer que les propriétés de notre transformation sont vérifiées. Dans notre transformation de modèles, nous utilisons cette vérification dans le but de vérifier si notre transformation est valide ou non en extrairant quatre propriétés qui traduisent les exigences particulières du

système à prouver.
La première propriété consiste à calculer le nombre de transitions de diagramme d'états-transitions qui peuvent être transformées vers des transitions dans le réseau de Petri. Cette propriété se base sur la manipulation du modèle en entrée, plus les règles de transformation.

```
Fixpoint calcul1 (a:list Transition):nat:= match a with
|nil⇒ 0
|x::nil⇒ match (IstypeofSimpleS(Transition_source(x))) with
|false⇒ match (IstypeofPseudojunction (Transition_source(x))) with
|false⇒ match IstypeofPseudochoice (Transition_source(x)) bluewith
|false⇒ match IstypeofPseudoinitial (Transition_source(x) ) with
|false⇒ match IstypeofPseudoexitPoint (Transition_source(x)) with
|false⇒ match IstypeofPseudojoin (Transition_source(x) ) with
|false⇒ 0 |true ⇒ match (IstypeofSimpleS(Transition_target(x))) with
|true⇒ 1
|false⇒ 0.....
|x::l ⇒ match (IstypeofSimpleS(Transition_source(x))) with
|false⇒ match (IstypeofPseudojunction (Transition_source(x))) with
|false⇒ match IstypeofPseudochoice (Transition_source(x)) with
|false⇒ match IstypeofPseudoinitial (Transition_source(x) ) with
|false⇒ match IstypeofPseudoexitPoint (Transition_source(x)) with
|false⇒ match IstypeofPseudojoin (Transition_source(x) ) with
|false⇒ calcul1(l)
|true ⇒ match (IstypeofSimpleS(Transition_target(x))) with
|true ⇒ (1 +calcul1(l))
|false⇒ calcul1(l)
end end .....
```

Dans la deuxième propriété, nous calculons le nombre de transitions du modèle en sortie directement.

```
Fixpoint calcul2 (a:list Transition):nat:= match a with
|nil ⇒ 0
|m::nil ⇒ 1
|m::l ⇒ ( (1) + (calcul2(l)))end.
```

La troisième propriété concerne le calcul du nombre des états qui peuvent être translatés en des places dans le réseau de Petri. Cette propriété se base aussi sur l'utilisation du modèle en entrée, plus les règles de transformation.

```
Fixpoint calcul3 (a:list Vertex):nat:= match a with
|nil ⇒ 0
|x::nil ⇒ match IstypeofFinalS (x) with
|true ⇒ 1
|false ⇒ match IstypeofSimpleS (x) with
|true ⇒ 1
|false ⇒ match IstypeofPseudoterminate (x) with
|true ⇒ 1
|false ⇒ match IstypeofPseudoinitial (x) with
|true ⇒ 1
.....
|false ⇒ 0 end end end end
|x::l ⇒ match IstypeofFinalS (x) with
|true ⇒ (1+calcul3 (l))
|false ⇒ match IstypeofSimpleS (x) with
|true ⇒ (1+calcul3 (l))
|false ⇒ match IstypeofPseudoterminate (x) with
|true ⇒ (1+calcul3 (l))
|false ⇒ match IstypeofPseudoinitial (x) with
|true ⇒ (1+calcul3 (l))
.....
|false ⇒ (calcul3 (l)) end end end end end.
```

La quatrième propriété consiste à calculer le nombre de places du modèle en sortie directement.

```
Fixpoint calcul4 (a:list Place):nat:= match a with
|nil ⇒ 0
|m::nil ⇒ 1
|m::l ⇒ ( (1) + (calcul4(l)))end.
```

Notre théorème traduit la satisfaction de ces propriétés dans notre processus de transformation (empruntée de [56]). C'est à dire nous devons atteindre le résultat suivant :

Theorem preuve :forall a :Statemachine, EqStatemachineintoPetrinet a Transformation-StoP a =true.

6.4.3 Preuve de théorème

Après avoir spécifié le théorème en Coq, sa preuve peut se réaliser interactivement sous Coq grâce au langage de tactiques et de commandes.

Theorem preuve :forall a :Statemachine, EqStatemachineintoPetrinet a TransformationStoP a =true.
Proof.
intros.
unfold TransformationStoP.
induction a.....
Qed.

6.5 Conclusion

Dans ce chapitre, nous avons expérimenté à travers une étude de cas, une approche permettant de vérifier et valider la transformation de modèles qui permet de fournir une sémantique formelle aux diagrammes d'UML dans le but d'automatiser une partie de processus de développement dans le contexte de l'IDM. En effet, nous avons défini la transformation des diagrammes d'états-transitions vers les réseaux de Petri et avons vérifié la conformité des modèles à leurs métamodèles.
Par conséquent, cette transformation de modèles peut être classifiée premièrement comme une transformation hétérogène et deuxièmement elle peut être vue comme une transformation générique. Pour notre approche de validation, elle peut être appliquée avec différentes transformations de modèles en ajoutant des modifications partielles au niveau de propriétés qui traduisent la validation.

Conclusion Générale

L'ingénierie dirigée par des modèles permet le développement logiciel et elle a apporté plusieurs avantages dans la production et la maintenance des systèmes. L'IDM n'est intéressant que si la transformation de modèles qui occupe une place très importante, est partiellement ou complètement automatisée dans le processus de développement.
La notion de transformation se base sur un ensemble de techniques, langages et d'outils dont l'objectif final est de faciliter le concept de programmation et de minimiser le coût du développement.

Notre recherche de magister s'inscrit dans la problématique de la transformation de modèles et l'intégration de la vérification et de la validation dans ce processu. Dans nos travaux, nous nous sommes intéressées à la transformation de modèles avec le langage QVT avec la plateforme EMF/Eclipse. L'étude de la conformité des modèles utilisés à leurs méta-modèles en utilisant le langage formel OCL. L'utilisation de ce langage permet d'ajouter un aspect formel à la sémantique des métamodèles. L'intégration de l'outil Coq dans ce processus a pour objectif de valider le processus de transformation. L'expérimentation a été réalisée à travers une étude de cas assez complète comprenant la transformation des diagrammes d'états-transitions d'UML vers des réseaux de Petri.

Ce projet de recherche nous a conduit à explorer différents domaines :

- Découvrir le domaine de l'IDM avec ses différents formalismes et outils utilisés ddans le processus de développement d'application.
- La proposition d'une étude cas nous a permis de proposer des métamodèles assez complexes et de définir une sémantique formelle associée à ces derniers. La formalisation d'un ensemble de propriétés pour chaque méta-modèle a permis d'exprimer et d'exploiter la vérification de la conformité des modèles.
- L'utilisation de l'assistant de preuve Coq s'inscrit globalement dans l'activité de validation formelle en spécifiant complètement la spécification de notre processus de

transformation en langage Gallina. En plus, la preuve interactive de notre théorème assure la validité de notre transformation de modèles.

L'utilisation d'un langage dédié de transformation Comme QVT-op diminue temps de développement pour la création des règles de transformations et réduit le nombre des erreurs qui peuvent être produisent dans le codage manuelle. L'inexistance de présentation graphique pour les modèles dans le langage QVT-op est un inconvénient majeur pour la compréhension de la structure des modèles.
L'intégration des techniques formelles de preuve dans le contexte de l'IDM offre un cadre plus rigoureux dans le développement des systèmes afin d'assurer la vérification et la validation mais l'utilisation de ces techniques reste actuellement accessibles seulement aux spécialistes (les mathématiciens ou les informaticiens) puisqu'il manque une méthodologie permettant sa mise en œuvre.

Les travaux accomplis durant ce projet de recherche ouvrent un certain nombre de perspectives à la fois liées à la transformation de modèles et aux concepts de vérification et de validation de transformation de modèles. Comme suite, nous pouvons proposer quelques pistes de développement :

- Notre principe de validation expérimenté avec le système de preuve Coq peut être appliqué à différentes types de transformation de modèles en utilisant le même principe de théorème.
- Notre proposition de validation dans la transformation de modèles pourrait être semi-automatique permettant à l'utilisateur de remonter plus rapidement dans la chaîne de validation.
- Expérimenter autres technique formelles de preuve pour l'étude de cas proposé et déduire une comparison des techniques utilisées.
- Finalement, l'utilisation des autres langages de transformation de modèle présentées dans ce document et autres, peuvent faire l'objet d'expérimentation dans le contexte de transformation des diagramme d'états-transitions afin de répondre à la question suivante : quel type de langage doit-on choisir pour la transformation de modèles?

Bibliographie

[1] Jamal Abd-Ali. *Métamodélisation et transformation automatique de PSM dans une approche MDA*. PhD thesis, Université du Québec en Outaouais, 2006.

[2] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[3] Alcatel, Softeam, Thales, and TNI-Valiosys. Response to the mof 2.0 query/views/-transformations rfp (ad/2003-04-10), 2003.

[4] Flemming Andersen. *A theorem prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark Lyngby Denmark, 1992.

[5] ATOM3. Web site. http://atom3.cs.mcgill.ca/. MSDL : Modelling, Simulation and Design Lab.

[6] Atelier B. Web site. http://www.atelierb.eu/, 2012. Clearsy System Engineering.

[7] Mariano Belaunde and Gregoire Dupe. SmartQVT, an implementation of the MOF QVT Operational language in top of EMF. http://www.eclipsecon.org/2007/index4c44.html?page=sub/&id=3995, March 2007. EclipseCON.

[8] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *UPPAAL a tool suite for automatic verification of real-time systems*. Springer, 1996.

[9] Karima Berramla, El Abbassia Deba, and Mohamed Senouci. Experiment on verification and validation of model transformation with coq proof assistant. In *International Symposium on Informatics and its Applications (ISIA)*, M'sila-Algeria, February 2014.

[10] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[11] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.

[12] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14(1):25–59, 1987.

[13] Jonathan Peter Bowen. *Formal specification and documentation using Z: A case study approach*, volume 66. International Thomson Computer Press London, 1996.

[14] Peter Braun, Frank Marschall, et al. Botl–the bidirectional object oriented transformation language. 2003.

[15] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, pages 85–94. IEEE, 2006.

[16] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[17] Fabian Büttner, Marina Egea, and Jordi Cabot. On verifying ATL transformations using ?off-the-shelf?SMT solvers. In *Model Driven Engineering Languages and Systems*, pages 432–448. Springer, 2012.

[18] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. In *Proc. 2nd Chilean Workshop on Formal Methods*, pages 29–36, 2009.

[19] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. A type-theoretic framework for certified model transformations. In *Formal Methods: Foundations and Applications*, pages 112–127. Springer, 2011.

[20] Daniel Calegari and Nora Szasz. Verification of model transformations: a survey of the state-of-the-art. *Electronic Notes In Theoretical Computer Science*, 292:5–25, 2013.

[21] Eric Cariou. Ingénierie des Modèles Méta-modélisation, 2010.

[22] CBOP, DSTC, and IBM. MOF Query/Views/Transformations, Revised Submission. 2003.

[23] Jean Benoîst T. Chabrol and Ludwig von Bertalanffy. *Théorie générale des systèmes*. Dunod, 1973.

[24] Christine Choppy, Kais Klai, and Hacene Zidani. Formal verification of UML state diagrams: a petri net based approach. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[25] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[26] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[27] Codagen. Architect 3.0. http://www.codagen.com/products/architect/default.html.

[28] Compuware. Optimalj 3.0, users guide. http://www.compuware.com/products/optimalj.

[29] Catarina Coquand. The interactive theorem prover Agda, 2001.

[30] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[31] Dumoulin, Cédric. Modtransf v3. http://www.lifl.fr/~dumoulin/modTransf/, 2003.

[32] Jacky Estublier, Jean Marie Favre, Jean Bézivin, Laurence Duchien, Raphael Marvie, and Others. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Technical report, CNRS, Janvier 2005.

[33] Anne Etien, Cedric Dumoulin, and Emmanuel Renaux. Towards a Unified Notation to Represent Model Transformation. Research Report RR-6187, INRIA, 2007.

[34] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In *MoDELS'07*, Nashville, TN, USA, États-Unis, 2007.

[35] Kokichi Futatsugi, Joseph A Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 52–66. ACM, 1985.

[36] Miguel Garcia and Ralf Möller. Certification of transformation algorithms in model-driven software development. *Software Engineering*, 105:107–118, 2007.

[37] Bilel Gargouri, Mohamed Jmaiel, and Abdelmajid Ben Hamadou. Vers l'utilisation des méthodes formelles pour le développement de linguiciels. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, pages 438–443. Association for Computational Linguistics, 1998.

[38] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, and Robert Wagner. Towards verified model transformations. In *Proc. of the 3rd International Workshop on Model Development, Validation and Verification (MoDeV 2a), Genova, Italy*, pages 78–93. Citeseer, 2006.

[39] Michael JC Gordon and Tom F Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[40] GReAT. Web site. http://www.isis.vanderbilt.edu/tools/GReAT. ISIS - Institute for Software Integrated Systems.

[41] Jean-Charles Grégoire, Gerard J Holzmann, and Doron Peled. *The SPIN Verification System: The Second Workshop on the SPIN Verification System: Proceedings of a DIMACS Workshop, August 5, 1996*, volume 32. American Mathematical Soc., 1997.

[42] GrGen. Web site. http://www.info.uni-karlsruhe.de/software/grgen/.

[43] ATLAS Group et al. *ATL User Manual Version 0.7*, 2006. LINA & INRIA.

[44] RAISE Language Group. *The RAISE specification language*. Prentice Hall, 1992.

[45] John V Guttag, James J Horning, Stephen J Garland, Kevin D Jones, Andres Modet, and Jeannette M Wing. Larch: languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Citeseer, 1993.

[46] Andrew Harry. *Formal Methods: Fact File: VDM and Z*. John Wiley & Sons, Inc., 1997.

[47] Anne E. Haxthausen. An Introduction to Formal Methods for the Development of Safety-critical Applications. *Technical University of Denmark*, 2010.

[48] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[49] INRIA. The Coq Proof assistant. http://coq.inria.fr/.

[50] ISO. Management de la qualité et assurance de la qualité, 1995.

[51] Jamda. The Java Model Driven Architecture 0.2, 2003.

[52] Manfred A Jeusfeld, Matthias Jarke, and John Mylopoulos. *Metamodeling for method engineering*. the MIT Press, 2009.

[53] Cliff B Jones, Kevin D Jones, Peter Alexander Lindsay, Richard Moore, J Bicarregui, M Elvang-Gøransson, RE Fields, R Kneuper, B Ritchie, and AC Wills. Mural – A Formal Development Support System, 2004.

[54] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[55] Kermeta. Web site. http://www.kermeta.org/. Triskell Team.

[56] Mounira Kezadri, Benoit Combemale, Marc Pantel, and Xavier Thirioux. A proof assistant based formalization of MDE components (regular paper). In *Formal Aspects of Component Software (FACS), Oslo, Norway, 14/09/2011-16/09/2011*, volume 7253 of *Lecture Notes in Computer Science*, page (electronic medium), http://www.springerlink.com, septembre 2011. Springer.

[57] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 285–294. ACM, 2007.

[58] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the Model Driven Architecture: practice and promise*. Addison-Wesley Professional, 2003.

[59] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.

[60] Jochen M Küster. Systematic validation of model transformations. *WiSME'04 (associated to UML'04)*, 2004.

[61] Maher Lamari. Towards an automated test generation for the verification of model transformations. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 998–1005. ACM, 2007.

[62] Kevin Lano and Howard Haughton. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[63] Zhaohui Luo and Robert Pollack. *LEGO proof development system: User's manual*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992.

[64] Petra Malik and Mark Utting. CZT: A framework for Z tools. In *ZB 2005: Formal Specification and Development in Z and B*, pages 65–84. Springer, 2005.

[65] Eric Meyer. *Développement formel par objets : utilisation conjointe de B et UML*. PhD thesis, Université de Nancy, 2001.

[66] Robin Milner. Lectures on a calculus for communicating systems. In *Control Flow and Data Flow: concepts of distributed programming*, pages 205–228. Springer, 1986.

[67] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. Redbooks, February 2004.

[68] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[69] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[70] OMG. MOF Query/Views/Transformations, Adopted Submission. http://www.omg.org/spec/QVT/1.1/PDF/.

[71] OMG. Meta object facility 1.4. http://doc.omg.org/formal/2002-04-03.pdf, 2002.

[72] OMG. Unified Modeling Language: Superstructure. {http://www2.imm.dtu.dk/courses/02291/files/UML2.1.1_superstructure.pdf}, 2007.

[73] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. http://www.omg.org/spec/QVT/1.1/PDF/, January 2011.

[74] OMG. Object Constraint Language, OMG Available Specification, Version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/PDF/, 2012.

[75] Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. PVS Language Reference. SRI International, 2001. Version 2.4. http://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

[76] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[77] Carl Adam Petri. Communication with automata. Technical Report RADC-TR-65-377, 1966.

[78] TIme petri Net Analyzer. Web site. http://projects.laas.fr/tina. LAAS/CNRS.

[79] Remko Popma. JET Tutorial Part 2 (Write Code that Writes Code). http://eclipse.org/articles/Article-JET2/jet_tutorial2.html, May 2004.

[80] P. Roques and F. Vallée. Les Cahiers du Programmeur UML2 Modéliser une application web 4e édition. pages 141–150. EYROLLES, 2008.

[81] Bernhard Schätz. Verification of model transformations. *Electronic Communications of the EASST*, 29, 2010.

[82] Johannes Schönböck. *Testing and Debugging of Model Transformations*. PhD thesis, E188 Institut für Softwaretechnik und Interaktive Systeme, 2012.

[83] Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 151–160. IEEE, 2004.

[84] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.

[85] VIATRA. Web site. http://www.eclipse.org/gmt/VIATRA2/.

[86] Jeannette M Wing. Formal Methods. *Encyclopedia of Software Engineering*, pages 504–517, 1994. Revision in Second Edition.

Annexe

Dans cette annexe, nous présentons la spécification du modèle cible pour chaque exemple d'exécution de la transformation (les exemples de la section 5.4) dans le système EMF Eclipse en langage XMI.
Le code suivant décrit la définition du modèle en sortie de la transformation de diagramme d'état qui modélise le comportement de la montre vers les réseaux de Petri(de la sous section 5.4.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<petrinet:Petrinet xmi:version="2.0" xmlns:xmi="http:
//www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xmlns:petrinet="http://petrinet/1.0" xsi:schemaLocation
="http://petrinet/1.0 SMtoPN/metamodel/petrinet.ecore" name="subm">
  <Places name="1 initial"/>
  <Places name="VerifyTranslation"/>
  <Places name="04 final"/>
  <Places name="11 initial"/>
  <Places name="OutOfService"/>
  <Places name="selectamount"/>
  <Places name="VerifyCart"/>
  <Places name="CartReleased"/>
  <Places name="enteramount"/>
  <PNTransition From="//@Places.0" To="//@Places.5" name="01"/>
  <PNTransition From="//@Places.8" To="//@Places.2" name="03ok"/>
  <PNTransition From="//@Places.5" To="//@Places.2" name="02amount"/>
  <PNTransition From="//@Places.0 //@Places.2 //@Places.5
  //@Places.8" To="//@Places.4" name="OutOfService3Subs"/>
  <PNTransition From="//@Places.5" To="//@Places.8" name="04otheramount"/>
  <PNTransition From="//@Places.1" To="//@Places.7" name="ReleaseCart5"/>
  <PNTransition From="//@Places.8 //@Places.5" To="//@Places.7" name="6
  abortedexit"/>
  <PNTransition From="//@Places.3" To="//@Places.6" name="1inital"/>
  <PNTransition From="//@Places.5 //@Places.8" To="//@Places.1" name="4Subs"/>
  <PNTransition From="//@Places.6" To="//@Places.0" name="AcceptCart2Subs"/>
</petrinet:Petrinet>
```

Dans ce qui suit, nous présentons le code en langage XMI du modèle cible de l'exemple d'exécution qui modélise le système ATM (exemple de la sous section 5.4.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<petrinet:Petrinet xmi:version="2.0" xmlns:xmi="http:
//www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xmlns:petrinet="http://petrinet/1.0"
 xsi:schemaLocation="http://petrinet/1.0 SMtoPN/metamodel/
 petrinet.ecore" name="Statemachine">
  <Places name="Modification heure Alarme"/>
  <Places name="Désarmée"/>
  <Places name="Sonnerie"/>
```

```
  <Places name="eteinte"/>
  <Places name="Armée"/>
  <Places name="Modification Heure"/>
  <Places name="eclairée"/>
  <Places name="c1 initial"/>
  <Places name="Affichage"/>
  <Places name="1 initial"/>
  <Places name="Modification Minute"/>
  <Places name="b1 initial"/>
  <Places name="Modification minute Alarme"/>
  <Places name="Statemachine  initialG"/>
  <PNTransition From="//@Places.13" To="//@Places.9 //@Places.11
   //@Places.7" name="Statemachine initialG"/>
  <PNTransition From="//@Places.11" To="//@Places.1" name="b6inital"/>
  <PNTransition From="//@Places.2" To="//@Places.1" name="appuialarmeb5"/>
  <PNTransition From="//@Places.0" To="//@Places.0" name="6"/>
  <PNTransition From="//@Places.1" To="//@Places.4" name="appuialarmeb1"/>
  <PNTransition From="//@Places.2" To="//@Places.2" name="Sonnerie"/>
  <PNTransition From="//@Places.5" To="//@Places.10" name="appuimode2"/>
  <PNTransition From="//@Places.10" To="//@Places.10" name="appuiavance4"/>
  <PNTransition From="//@Places.12" To="//@Places.12" name="appuiavance8"/>
  <PNTransition From="//@Places.7" To="//@Places.3" name="c1inital"/>
  <PNTransition From="//@Places.9" To="//@Places.8" name="10inital"/>
  <PNTransition From="//@Places.4" To="//@Places.2" name="b4"
  expressions="when(heurecourant=)"/>
  <PNTransition From="//@Places.10" To="//@Places.0" name="apuimode5"/>
  <PNTransition From="//@Places.12" To="//@Places.8" name="appuimode9"/>
  <PNTransition From="//@Places.8" To="//@Places.5" name="appuieMonde1"/>
  <PNTransition From="//@Places.0" To="//@Places.12" name="appuimode7"/>
  <PNTransition From="//@Places.4" To="//@Places.1" name="appuialarmeb2"/>
  <PNTransition From="//@Places.6" To="//@Places.6" name="Eclairée"/>
  <PNTransition From="//@Places.5" To="//@Places.5" name="appuiheure3"/>
  <PNTransition From="//@Places.3" To="//@Places.6" name="appuieclairagec2"/>
  <PNTransition From="//@Places.6" To="//@Places.3" name="relacheeclairagec3"/>
  <PNTransition From="//@Places.2" To="//@Places.4" name="b3"/>
</petrinet:Petrinet>
```

Nous décrivons dans la suite quelques règles de cohérence en langage OCL (pour vérifier la conformité des modèles selon leurs métamodèles). Le reste de ces règles sont exprimées par la structure des métamodèles.

Règle de cohérence	**La spécification en langage OCL**
R26	self.region->forAll(s1 : Region | self.region->forAll(s2 : Region | s1.name = s2.name implies s1 = s2));
R25	if self.region->size() <= 1 then self.isHorthogonal = false else self.isHorthogonal = true endif;
R24	self.container->select(v : Vertex | v.oclIsKindOf(Pseudostate))->select(p : Pseudostate | (p.kind = PseudostateKind::initial))->size() <= 1;
R19	source.oclIsKindOf(Pseudostate) and source.kind = PseudostateKind::fork implies guard->isEmpty() and Trigger->isEmpty();

Règle de cohérence	La spécification en langage OCL
R20	target.oclIsKindOf(Pseudostate) and target.kind = PseudostateKind::join implies guard->isEmpty() and Trigger->isEmpty();
R21	source.oclIsKindOf(Pseudostate) and source.kind = PseudostateKind::fork implies target.oclIsKindOf(State);
R22	target.oclIsKindOf(Pseudostate) and target.kind = PseudostateKind::join implies source.oclIsKindOf(State);
R23	source.oclIsKindOf(Pseudostate) and source.kind <> PseudostateKind::junction and source.kind <> PseudostateKind::join and source.kind <> PseudostateKind::initial implies Trigger->isEmpty();
R7	self.kind = PseudostateKind::initial implies self.outgoing->size() <= 1;
R6	self.kind = PseudostateKind::initial implies self.incoming->size() = 0;
R14	self.kind = PseudostateKind::join implies self.outgoing->size() = 1 and self.incoming->size() >= 2;
R13	self.kind = PseudostateKind::fork implies self.incoming->size() = 1 and self.outgoing->size() >= 2;
R11	self.kind = PseudostateKind::junction implies self.incoming->size() >= 1 and self.outgoing->size() >= 1;
R12	self.kind = PseudostateKind::choice implies self.incoming->size() >= 1 and self.outgoing->size() >= 1;
R8	self.kind = PseudostateKind::initial implies self.outgoing.guard->isEmpty() and self.outgoing.Trigger->isEmpty();
R9	self.kind = PseudostateKind::terminate implies self.outgoing->size() = 0;
R10	self.kind = PseudostateKind::terminate implies self.incoming->size() <= 1;
R2	self.outgoing->size() = 0;
R15, R25 et R26	self.entryc->notEmpty() implies self.entryc->forAll(e : Pseudostate | e.kind = PseudostateKind::entryPoint);
R16, R25 et R26	self.exitc->notEmpty() implies self.exitc->forAll(e : Pseudostate | e.kind = PseudostateKind::exitPoint);
R28	self.PNTransition->forAll(s1 : Transition | self.PNTransition->forAll(s2 : Transition | s1.name = s2.name and s1.From = s2.From and s1.To = s2.To and s1.expressions = s2.expressions implies s1 = s2));
R29	self.Places->forAll(s1 : Place | self.Places->forAll(s2 : Place | s1.name = s2.name implies s1 = s2));

Résume

L'Ingénierie Dirigée par les Modèles (IDM) se base sur l'utilisation intensive et systématique des modèles tout au long du processus de développement de logiciels. Ainsi, l'IDM vise à améliorer l'interopérabilité, la réutilisation et la possibilité de migrer les modèles pour des applications hétérogènes. Dans le contexte de l'IDM, la transformation de modèles est une phase cruciale puisqu'elle définit l'automatisation qui peut être utilisée dans le processus de développement de logiciels. Il est donc nécessaire d'augmenter la sûreté, la vérification et la validation d'une telle transformation de modèles. A cet effet, il est important de proposer des solutions pour intégrer les méthodes formelles dans le processus de transformation où la plupart des langages de transformation n'ont pas une sémantique formelle pour ajouter des spécifications détaillées sur le comportement attendu. L'idée de base de notre travail est de définir une transformation de modèles avec un langage de transformation comme QVT-op, vérifier la conformité des modèles avec le langage OCL et valider le programme de transformation en utilisant un assistant de preuve Coq. Nous illustrons ce travail en spécifiant une transformation de modèle complexe et hétérogène entre le diagramme d'états-transitions UML et les réseaux de Pétri.

Mots Clés :

Transformation de modèles; Langage QVT-op; Vérification; Validation; Assistant de preuve Coq; Langage OCL; Diagramme d'état-transition; Ingénierie Dirigée par les Modèles; Réseaux de Pétri; Méthodes Formelles.

.