REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

N° :……..

DOMAINE : MATHEMATIQUES ET INFORMATIQUE

FILIERE : INFORMATIQUE

OPTION : Systèmes d'Informations Avancés

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par: Boudrouaz Baha Eddine

Intitulé

Une approche MDA pour la transformation des statecharts vers B en utilisant TGG

Soutenu devant le jury composé de :

Mr. Amri Said	Université de M'sila	Président
Mme. Boudia Malika	Université de M'sila	Rapporteur
Dr. Bounif Mohamed	Université de M'sila	Examinateur

Année universitaire : 2016 /2017

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE DES MATHEMATIQUES ET DE L'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

N° :……..

DOMAINE : MATHEMATIQUES ET INFORMATIQUE

FILIERE : INFORMATIQUE

OPTION : Systèmes d'Informations Avancés

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par: Boudrouaz Baha Eddine

Intitulé

Une approche MDA pour la transformation des statecharts vers B en utilisant TGG

Soutenu devant le jury composé de :

Mr. Amri Said	Université de M'sila	Président
Mme. Boudia Malika	Université de M'sila	Rapporteur
Dr. Bounif Mohamed	Université de M'sila	Examinateur

Année universitaire : 2016 /2017

Remerciements

Avant tout on tient notre remerciement à notre dieu tout puissant de nous avoir donné la foi, la force et le courage.

Je remercie mon encadreur Mme Boudia Malika, de sa

Disponibilité, sa générosité professionnelle et ses précieux conseils.

Je remercie les membres du jury qui m'ont honoré de leur présence et d'avoir accepté de juger mon travail.

Merci à tous.

Boudrouaz Baha Eddine

TABLE DES MATIERES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS

[AGG] Attributed **G**raph **G**rammar

[API] Application **P**rogramming **I**nterface

[ATOM^3] A TOol for **M**ulti-formalism and Meta-Modeling

[CIM] Computation **I**ndependent **M**odel

[EMF] Eclipse **M**odeling **F**ramework

[EMF Tiger] Eclipse **M**odeling **F**ramework **T**ransformation **GEN**eration

[FUJABA] From **U**ml to **JA**va and **BA**ck

[GMF] Graphical **M**odeling **P**roject

[GrGen] Graph Rewrite **Gen**erator

[GROOVE] GRaph for **O**bejct-**O**riented **VE**rification

[IDM] Ingénierie **D**irigée par les **M**odèles

[M2M] Model **to M**odel

[M2T] Model **to T**ext

[MDA] Model **D**riven **A**rchitecture

[MOF] Model **O**bject **F**acility

[OMG] Object **M**anagement **G**roup

[PDM] Platform **D**escription **M**odel

[PIM] Platform **I**ndependent **M**odel

[PSM] Platform **S**pecific **M**odel

[QVT] Query/**V**iew/**T**ransformation

[TG] Triple **G**raph

[TGG] Triple **G**raph **G**rammar

[UML] Unified **M**odeling **L**anguage

[VITRA] VIsual automated model **TRA**nsformation

[XMI] XML **M**etadata **I**nterchange

[XML] e**X**tended **M**arkup **L**anguage

[XSD] XML **S**chéma **D**éfinition

INTRODUCTION GENERALE

INTRODUCTION GENERALE

Aujourd'hui, les systèmes modélisés sont de plus en plus complexe et leur taille ne cesse d'augmenter.

Le langage UML est constitué de diagrammes, il existe treize types des diagrammes. Ces diagrammes sont tous réalisés à partir du besoin des utilisateurs.

Le diagramme d'états-transitions est parmi les diagrammes, de la norme UML, à offrir une vision complète de l'ensemble des comportements de l'élément auquel il est attaché.

Les diagrammes d'états-transitions (statecharts diagram), concept utilisé par David Harel pour son extension de notation de machine d'état à plat comprend des états imbriqués et concurrents. Cette notation a servi de base à la notation de la machine UML.

UML étant un langage à caractère plutôt visuel, il souffre d'un manque de Sémantique formelle. En effet, les notations semi-formelles et visuelles d'UML peuvent provoquer des inconsistances au niveau des modèles développés [1].

L'architecture dirigée par les modèles ou MDA est une démarche de réalisation de logiciel, proposée et soutenue par l'OMG.

Le principe de MDA est l'élaboration de modèle pérennes, indépendants des détails techniques des plates-formes d'exécution (J2EE, .NET, PHP ou autre), afin de permettre la génération automatique de la totalité du code des applications et d'obtenir un gain significatif de productivité [2].

Les méthodes formelles, et parmi elles la méthode B, permettent d'atteindre ce niveau de qualité. Cependant, ces méthodes utilisent des notations et des concepts spécifiques, qui génèrent souvent une faible lisibilité et une difficulté d'intégration dans les processus de développement et de certification. Ainsi, proposer des environnements de spécification, de développement de programmes et de logiciels, combinant des méthodes formelles et des méthodes semi-formelles largement utilisées dans les projets industriels.

La méthode B qui est une méthode formelle utilisée pour modéliser des systèmes et prouver l'exactitude de leur conception par raffinements successifs. Mais les spécifications formelles sont difficiles à lire quand elles ne sont pas accompagnées d'une documentation.

La transformation de modèles est un concept clé pour l'ingénierie dirigée par les modèles. Dans ce contexte, les grammaires TGG (Triple Graph Grammar) ont été étudiées et appliquées à plusieurs études de cas et elles montrent une combinaison commode de spécification formelle et de capacités intuitives. Surtout la dérivation automatique des transformations avant et arrière sur un seul ensemble de règles spécifiques pour le modèle intégré simplifie la spécification et améliore la facilité d'utilisation ainsi que la cohérence [3].

INTRODUCTION GENERALE

La mise en pratique d'UML nécessite un apprentissage et passe par une période d'adaptation. Même si l'Espéranto est une utopie, la nécessité de s'accorder sur des modes d'expression communs est vitale en informatique. Le processus (non couvert par UML) est un autre clé de la réussite d'un projet. Or l'intégration d'UML dans un processus n'est pas triviale et améliorer un processus est une tâche complexe et longue.

La notation des Statecharts d'UML n'est pas purement visuelle. N'importe quelle machine d'état non triviale exige un grand nombre d'information textuelle (par exemple, les spécifications des actions et des gardes).

La majeure partie de la sémantique de statecharts est fortement concentrée vers la notation graphique. Par exemple, les diagrammes d'état représentent mal l'ordre du traitement, que ce soit l'ordre de l'évaluation des gardes ou l'ordre d'expédier des événements aux régions orthogonales.

La formalisation des machines d'états peut être spécifiée en plusieurs méthodes : Mathématiques, non-mathématiques, textuelles, graphiques, Théoriques vs. Application pratique, etc.

Cette formalisation a pour objectif d'enrichir les machines d'états UML par une sémantique précise non ambigüe qui permet par conséquence la vérification formelle des diagrammes d'états-transitions.

L'objectif alors, est d'enrichir la notation graphique par des notations mathématiques.

Plan de travail

Ce mémoire est divisé en quatre chapitres :

- Le chapitre 1 : décrit les diagrammes d'états-transitions, leurs concepts de base et leurs concepts avancés.

- Le chapitre 2 : présente les méthodes formelles, après avoir introduit les méthodes semi-formelles, introduit la méthode B et ses concepts de base.

- Le chapitre 3 : apporte les définitions de modèle, méta- modèle, introduit la définition de la transformation de modèles, discute le principe de transformation avec TGG (triple graph grammar).

- Le chapitre 4 : propose une approche de transformation de diagrammes d'états transitions vers B en utilisant TGG.

1. CHAPITRE 1
STATECHARTS

1. Introduction

Les diagrammes d'états-transitions d'UML décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode).

Le diagramme d'états-transitions est le seul diagramme, de la norme UML, à offrir une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché [4].

2. Statecharts (Diagramme d'états-transitions)

2.1. Présentation

Les diagrammes d'état transition (statecharts diagram), concept utilisé par David Harel. [5] pour son extension de notation de machine d'état à plat comprend des états imbriqués et concurrents. Cette notation a servi de base à la notation de la machine UML.

Les diagrammes d'états-transitions UML représentent en réalité des automates à états finis, mathématiquement parlant, ils représentent des graphes orientés.

Le diagramme d'états-transitions est le seul diagramme en UML qui offre une vision complète et non ambiguë de l'ensemble des composants de l'élément auquel il est attaché [6]. La figure 1.1 exprime le protocole d'états- machine.

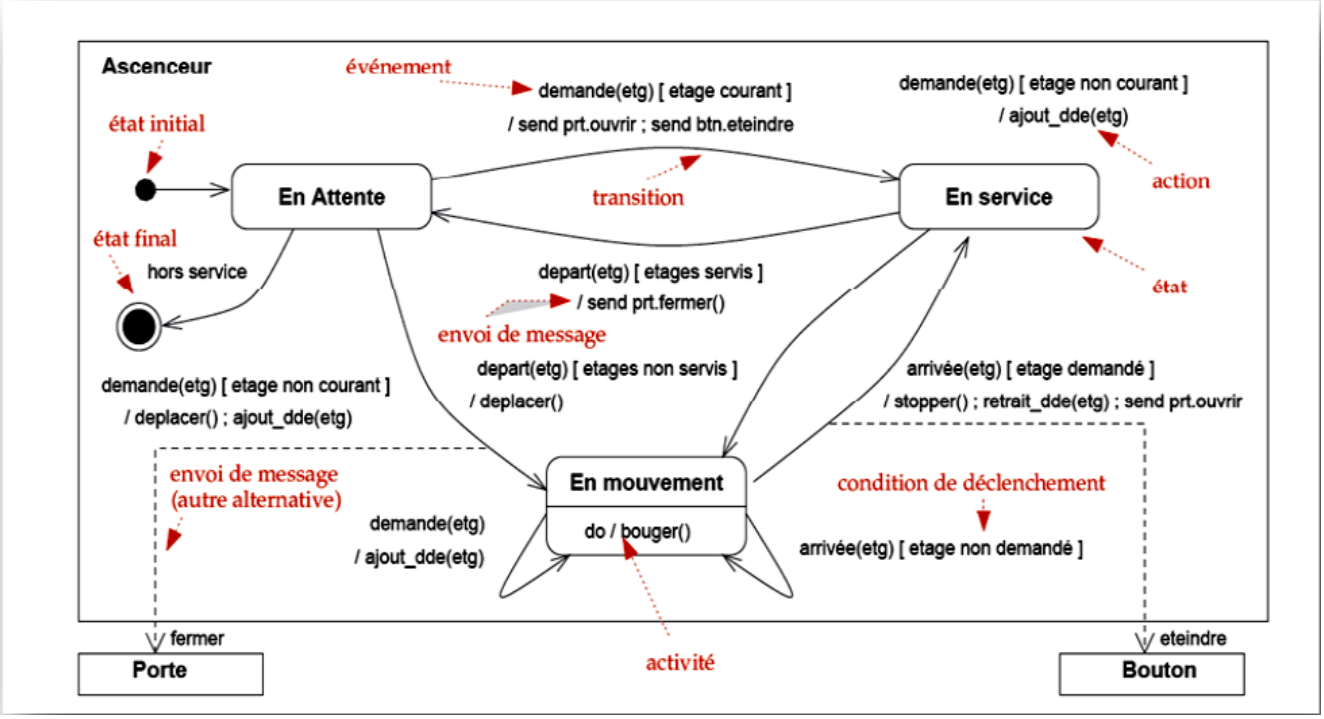


Figure 1.1: diagramme d'états- transitions [7].

2.2. Les états (state)

Un état est la condition d'un objet à un moment donné [8], c'est une situation donnée durant la vie de l'élément qui satisfait à des conditions, réalise des actions ou est en attente d'événement. Cet état dépend des états précédents et des événements survenus. Graphiquement, les états sont représentés sous forme de rectangles arrondis ; chaque état peut posséder un nom qui le distingue des autres (le nom de l'état est optionnel). Les états sans nom sont anonymes et distincts les uns des autres [9].

Les états sont caractérisés par deux choses différentes : la valeur des attributs de l'objet et la valeur des liens avec les autres objets à un instant donné.

Un diagramme d'états-transitions est habituellement constitué d'un état initial, éventuellement des états intermédiaires (zéro ou plusieurs), et un ou plusieurs états finaux [10].

- ❖ Un état peut être simple, ou composite ou de sous machine. Par définition un état simple n'a pas de sous état imbriqué, un état composite contient un ou plusieurs régions, chaque (région) contient un ou plusieurs sous états. Si un état composite est actif, chacune de ces régions est active.
- ❖ Un état composite peut être non orthogonal ou orthogonal. Un état non orthogonal possède une seule région. L'état orthogonal modélise la concurrence.

La figure 1.2, Présente le protocole d'états machine.

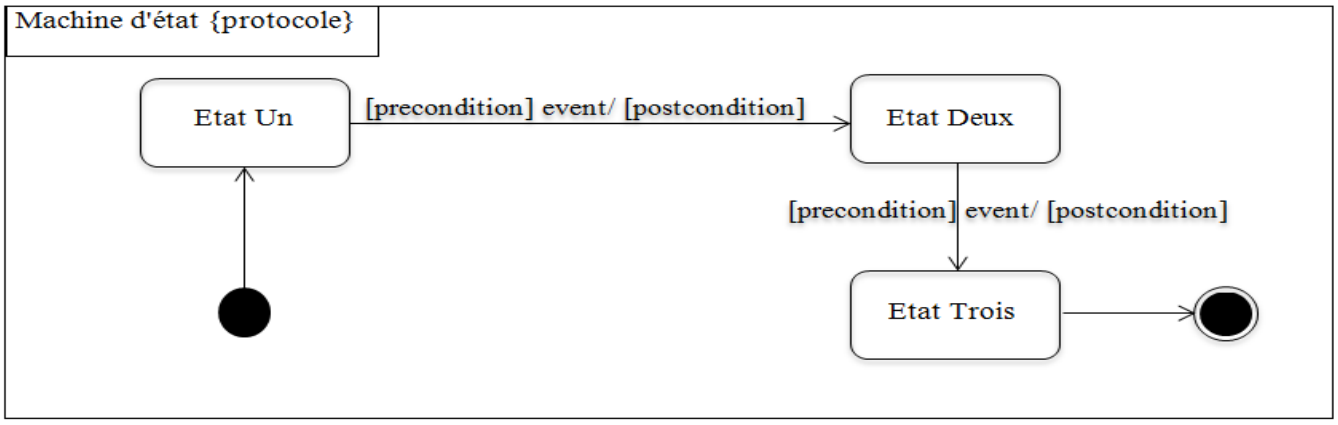


Figure 1.2: Protocole d'états machine [11].

Généralement un état est constitué de : [4]

a. Nom : une chaine textuelle qui distingue l'état d'autres états (un état peut être anonyme).

b. Sous état : si la machine d'états à une sous structure imbriquée, on l'appelle état composite. Un état composite contient une ou plusieurs régions, chacune contenant un ou plusieurs sous états directs. Un état sans structure (à l'exception des éventuelles actions internes) est un état simple.

c. Activités entry et exit : un état peut avoir une activité « entry » et une activité « exit », l'objectif de ces activités est d'encapsuler l'état de façon à ce qu'il soit utilisable en externe sans connaissance de sa structure interne. Elles représentent des actions à exécuter à l'entrée et à la sortie de l'état.

d. Transition internes : un état peut avoir une liste de transitions internes (qui sont comme des transitions normales mais qui n'ont pas d'état cible et ne provoquent pas de changement d'état).

e. Activité do interne : un état peut contenir une activité do interne décrite par une expression. Lors de l'entrée dans l'état, l'activité do commence après que l'activité entry est terminée.

f. Evénements rapportés : (déférés) c'est une liste d'événements qui ne sont pas traités dans cet état mais qui sont sauvegardés dans une file d'attente pour être traiter par l'objet dans d'autres états.

g. Sous machine : le corps d'un état peut représenter une copie d'une machine d'états distincte référencée par son nom, appelée sous machine d'état car elle est imbriquée dans la machine d'état étendu. Une sous machine peut être rattachée à une classe qui fournit le contexte des actions qui s'y trouvent comme les attributs qui peuvent être lus ou écrits. Une sous machine peut être réutilisée dans plusieurs machines d'états afin d'éviter la répétition du même fragment de machine d'états. Une sous machine est une sorte de sous routine de machine d'état.

h. Etat initial / état final :

Etat initial (initial state) : pseudo état qui indique l'état de départ par défaut de la région enveloppée.

Etat final (final state) : pseudo état qui indique que l'exécution du diagramme ou de l'état composite est terminé. La figure 1.3 désigne le formalisme de représentation des états initial et final.

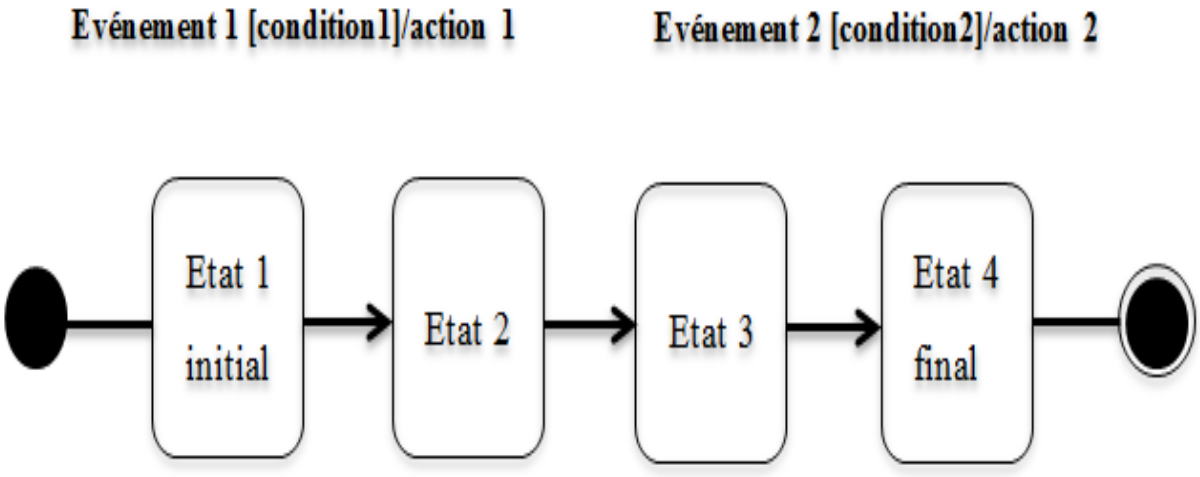


Figure 1.3: Formalisme de représentation des états initial et final [12]

2.3. Les transitions (transition)

C'est une relation entre deux états qui indique que l'objet change d'état lorsqu'un événement se produit. Sémantiquement, les transitions représentent les chemins potentiels entre les états dans l'historique de vie de l'objet, ainsi que les actions réalisés dans le changement d'état.

Les diagrammes d'états-transitions sont des graphes dirigés, les états sont reliés par des connexions unidirectionnelles, appelées transitions.

Pour la structure, une transition possède un état source, un déclencheur d'événement, une condition de garde, une action et un état cible, certains éléments peuvent ne pas figurer. Généralement, une transition est constituée de [4]:

a. Etat source : l'état source est l'état affecté par la transition. Si un objet se trouve dans l'état source, une transition sortante de l'état peut se déclencher si l'objet reçoit l'événement déclencheur de la transition de garde (soit satisfaite), l'état source devient inactif après le déclenchement de la transition.

b. Etat cible : l'état cible est l'état qui devient actif après l'achèvement de la transition.

c. Déclencheur d'événement (évent trigger): c'est un événement reconnu par l'état source et avec lequel la transition est franchissable une fois que la garde de la transition est vérifiée. Un événement peut être un signal, un appel de méthode, un passage de temps ou un changement d'état.

d. Condition de garde : la condition de garde est une expression booléenne qui est évaluée lorsque la gestion d'un événement déclenche une transition spécifiée par '[‘<garde>’]', syntaxiquement il s'agit d'une expression logique sur les attributs de l'objet, la transition ne se déclenche qu'une fois que la condition de la garde est évaluée « true ».

e. Effet : une transition peut contenir un effet, c'est-à-dire une action ou une activité qui s'exécute lorsque la transition se déclenche. Le comportement peut accéder à l'objet détenteur de la machine d'états et le modifier (ainsi qu'indirectement d'autres peuvent utiliser d'autres objets qu'il peut atteindre). L'effet peut utiliser des paramètres de l'événement déclencheur, ainsi que des attributs et des associations de l'objet détenteur. On peut accéder à l'événement déclencheur comme étant l'événement en cours pendant toute la durée de l'étape run-to-completion, qu'il a initié, y compris les segments dépourvus de déclencheurs et actions entry et exit .

On suppose qu'un effet a une durée de vie assez brève car la machine d'états ne peut pas traiter d'autres événements tant que son exécution n'est pas achevée. Tout comportement destiné à poursuivre pour une durée étendue doit être modélisé comme une activité do associé à un état plutôt qu'à une transition. Le tableau suivant illustre les types et les effets implicites.

Le tableau 1.1 résume les différents types de transition et effets implicites.

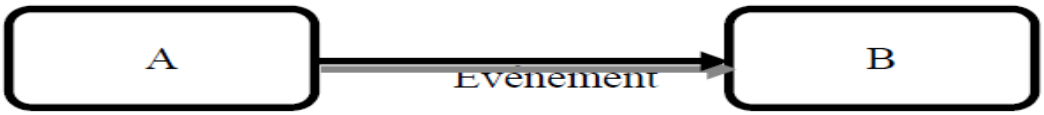
Type de transition	description	Syntaxe
Transition entry	Spécification d'une activité d'entrée qui s'exécute lorsqu'on saisit un état	**Entry/activity**
Transition exit	Spécification d'une activité de sortie qui s'exécute lorsqu'on quitte un état	**Exit /activity**
Transition externe	Réponse à un événement qui engendre un changement d'état ou une auto transition, ainsi qu'un effet spécifié. Elle peut également entrainer l'exécution d'une activité exit et/ou d'une activité entry pour les états dont l'on sort ou dans lesquels on entre.	**(a :T)[guard]/activity**
Transition interne	Réponse à un événement qui entraine l'exécution d'un effet mais pas d'un changement d'état, ou l'exécution d'activités exit ou entry	**e(a:T)[guard]/activity**

Tableau 1.1: Type de transition et effets implicites [4].

2.4. Les événements

Un événement est une occurrence d'un fait significatif ou remarquable. Un événement correspond à l'occurrence d'une situation donnée dans le domaine du problème. Contrairement aux états qui durent, un événement est par nature une information instantanée qui doit être traitée sans délai. Un événement peut déclencher le passage d'un état à un autre.

Les transitions indiquent les chemins dans le graphe des états, par contre les événements déterminent quels chemins doivent être suivis. Les événements, les transitions et les états sont indissociables dans la description du comportement dynamique. Fonctionnellement, un objet placé dans un état donné attend l'occurrence d'un événement pour passer dans un autre état. La réception de l'événement par l'objet conduit au déclenchement de la transition. L'objet qui était dans un état source passe dans l'état destination de la transition. De ce point de vue, les objets se comportent comme des éléments passifs, contrôlés par les événements en provenance de système. L'exemple représente un événement qui déclenche une transition [4].



La définition compète d'un événement comprend :

- Le nom d'événement.
- La liste des paramètres éventuels.
- L'objet expéditeur.
- L'objet destinataire.
- La description de la signification de l'événement.

Différents types d'évènements sont définis dans UML.

Type d'événement	Description	Syntaxe
Appel	Réception d'une demande d'appel explicite synchrone par un objet	**Op (a : T)**
Changement	Changement dans la valeur d'une expression booléenne	**When (exp)**
Signal	Réception d'une communication explicite, nommé asynchrone entre des objets	**sname (a :T)**
Temps	Arrivée d'un temps absolu ou passage d'un laps de temps relatif	**after (time)**

Tableau 1.2: Types d'événements [10].

La figure 1.4 extraite de [13] illustre les éléments de statechart diagram (diagramme d'états-transition).

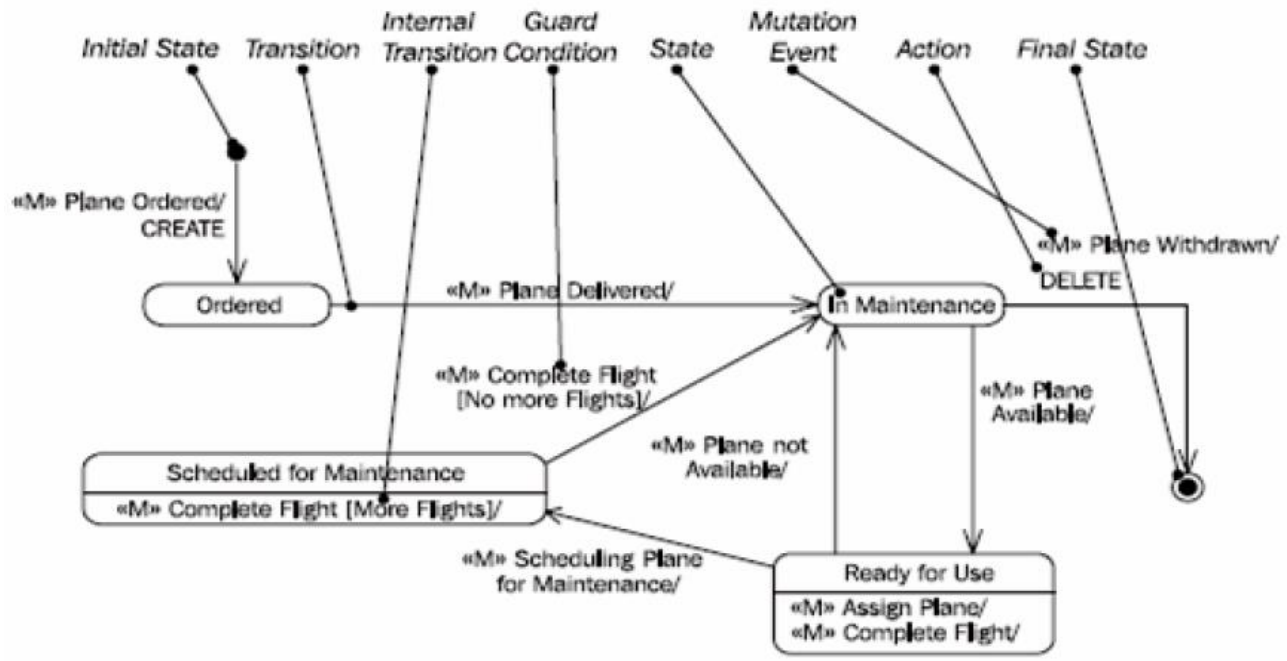


Figure 1.4: les éléments de statechart diagram (diagramme d'états-transitions) [13].

2.5. Les concepts avancés

Les diagrammes d'états transitions utilisent plusieurs concepts avancés afin de modéliser les comportements complexes, afin de diminuer le nombre des états et des transitions et afin de réduire la complexité des diagrammes d'états transitions [4].

- Etat historique (history state) : un état historique permet à un état composite de se souvenir de son dernier sous état actif avant son exit le plus récent. Un état historique permet de conserver un historique plat (shallow history) ou un historique profond (deep history). L'historique plat se souvient et réactive un état situé à la même profondeur d'imbrication que l'état historique lui-même. L'historique plat est représenté par un petit cercle contenant la lettre H. L'état historique profond se souvient d'un état qui a pu être imbriqué à une certaine profondeur de l'état composite. Il est représenté par la lettre H* [4].

Tous les formalismes d'états machine, y compris les statecharts d'UML, universellement, assument que les machines d'états complètent le traitement de chaque événement avant de commencer le traitement d'événement suivant. Ce modèle d'exécution s'appelle run-to-completion ou RTC [14].

- Point de choix : pour représenter des alternatives pour franchir d'une transition. UML2 offre des pseudos états particuliers : les points de jonction (graphiquement représentés par un petit cercle plein), et les points de décision (représentés graphiquement par losange, voir figure 1.5) [15].

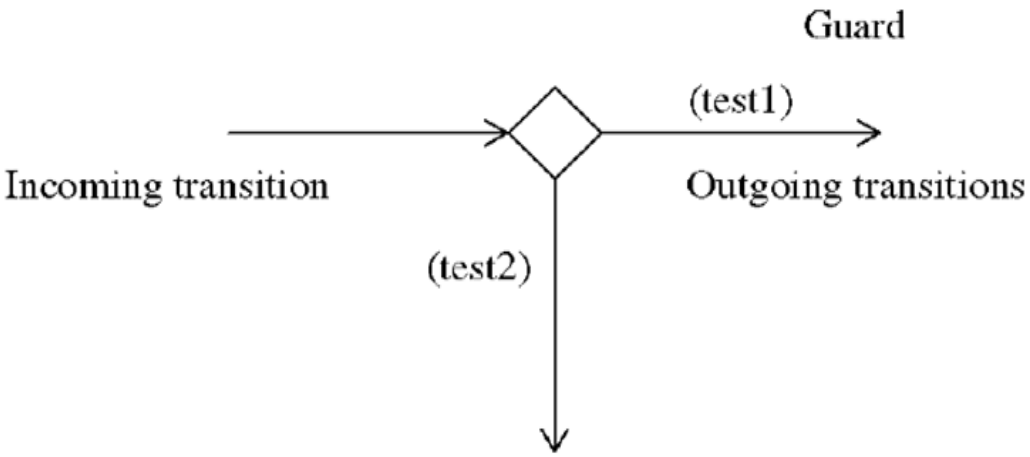


Figure 1.5: point de choix [15].

- Point de jonction : les points de jonction sont des pseudos états permettant de partager des segments de transition, afin d'aboutir à une notation plus compacte ou plus lisible des chemins alternatifs. Un point de jonction peut avoir plusieurs segments de transition entrante et plusieurs segments de transition sortante (voir la figure 1.6) [12].

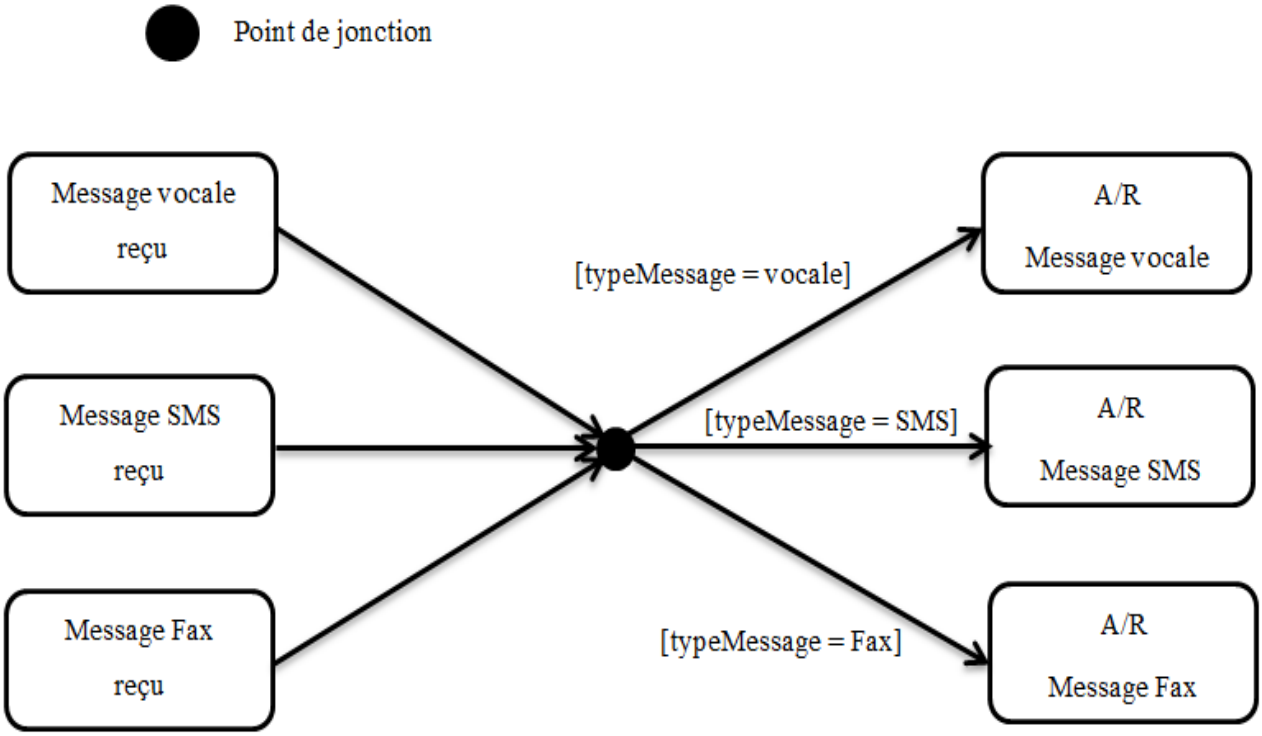


Figure 1.6: exemple d'état-transition avec point de jonction [12].

- Point de décision : un point de décision est un pseudo état qui possède une entrée et au moins deux sorties. Dans un point de jonction les gardes localisées après le point de décision sont évaluées au moment où il est atteint. Le choix est basé sur les résultats obtenus en franchissant le segment avant le point de choix dans une point de décision on peut utiliser une garde particulière, notée [else], sur un segment en aval d'un point de choix. Ce segment n'est franchissable que si les gardes des autres segments sont toutes fausses. Afin de garantir un modèle bien formé, il est recommandé d'utiliser la classe [else].
- La transition complexe (complex transition) : une transition complexe modélise la synchronisation de contrôle et/ou le débranchement de contrôles selon le nombre de sources et de cibles. Une transition complexe est une transition à partir ou vers un état orthogonal. Une transition complexe possède plusieurs états source et/ou plusieurs états cibles [4].

3. Conclusion

Dans ce chapitre nous avons exposés le diagramme d'états-transition (statechart diagram).

Statechart est un graphique des états reliés par les transitions, qui représentent la machine d'état. Typiquement, Statecharts décrivent le comportement d'une classe lorsqu'elle reçoit un événement.

Les diagrammes d'états transitions appartiennent au langage UML, qui est un langage semi-formel caractérisé par une syntaxe riche et une sémantique ambigüe et non précise.

L'alternative réside dans les méthodes formelles qui représentent une solution intéressante à ce problème pour effet d'éliminer les ambigüités au niveau de l'interprétation des modèles comme la méthode formelle B.

2. CHAPITRE 2
LANGAGE B

1. Introduction

Les méthodes formelles contribuent à la construction de spécification dont l'interprétation tient moins à une intuition humaine qu'à l'utilisation de méthodes liées aux mathématiques. L'utilisation des méthodes formelles au sien du développement de logiciel permet de concevoir des produits sûrs.

L'objectif, dans ce chapitre, est d'introduire d'une manière générale les méthodes formelles pour décrire plus précisément la méthode B. Nous commençons par définir ce qu'est une méthode formelle, nous discutons de son utilisation et présentons les différentes classes de méthodes existantes. Nous abordons ensuite la méthode B dont nous détaillons les idées et concepts de base nécessaires.

2. Introduction aux méthodes formelles

2.1. Définition

Une méthode est dite formelle si elle est fondée sur un langage avec une syntaxe et une sémantique précises, construit sur des bases théoriques. Ces bases sont généralement mathématiques, des raisonnements sur la spécification sont alors possibles : syntaxe et sémantique sont accompagnées de règles de déduction qui permettent de démontrer des propriétés d'une spécification [16].

2.2. Classification

On distingue différentes classifications pour les méthodes formelles dans la littérature, toutefois J. M. Wing [17], divise ces dernières selon trois approches (figure 2.1) :

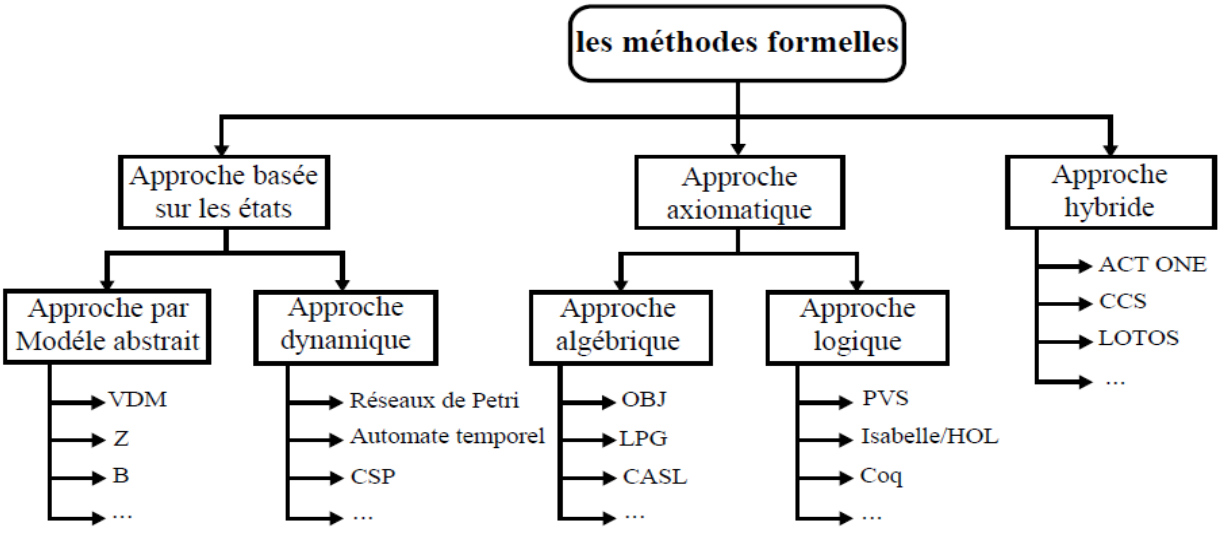


Figure 2.1: La classification des méthodes formelles [17].

3. La méthode B

La méthode B a été présentée au milieu des années 80 par J.R. Abrial [18] auteur à l'origine de la notation Z [19]. C'est une technique de développement formel du logiciel dont tous les niveaux de construction sont pris en charge, de la phase de spécification jusqu'à la génération de code, en passant par la conception. Toute partie ou élément constitue une activité qui doit être approuvée par des preuves mathématiques. Ces preuves ont pour rôle ainsi de divulguer les erreurs et de guider la conception et la maintenance. La méthode B a été conçue au départ comme une méthode de développement de logiciel prouvé, toutefois tend actuellement à être plus généralement une méthode d'analyse et de construction système [20].

3.1. Fondements théoriques

Le formalisme de la machine abstraite décrit toutes les phases de la méthode B, allant de la spécification à l'implémentation, en passant par la conception (raffinement). Afin de montrer le principe de fonctionnement de cette méthode, il nous faut mettre l'accent sur les fondements théoriques de son formalisme. Nous commençons par la description des notations mathématiques élémentaires, et allons jusqu'aux concepts fondamentaux que sont les machines abstraites. Afin d'étudier profondément et plus rigoureusement la méthode B, nous incitons de s'orienter vers le livre de référence de celle-ci : [18].

3.1.1. Notations mathématiques

Le langage B s'appuie sur les trois notations mathématiques : la logique prédicative du premier ordre, la théorie des ensembles et le langage des substitutions généralisées.

La théorie des ensembles et les types : La théorie des ensembles se sert comme la base des modèles mathématiques en B. Les relations, fonctions et suites sont décrites en tant que des couples d'ensembles. Quoique les variables en langage B soient strictement typées, leur type n'est pas défini explicitement à la déclaration comme pour la majorité des langages de programmation mais par des prédicats au sein de l'invariant, les pré-conditions ou les contraintes des paramètres formels. Ces prédicats indiquent le typage des variables par une condition d'appartenance à un ensemble donné.

Le langage des substitutions généralisées : L'ensemble composé de modules, appelés ainsi machines abstraites, constitue des spécifications B. Chaque machine comprend un état et des opérations qui permettent d'accéder à celui-ci ou de le changer. Les substitutions constituent les processus déterminant et changeant l'état dont leur sémantique est donnée par les substitutions généralisées [21].

Le concept de substitution est proche au mécanisme d'affectation des langages de programmation. Ce concept a été étendu en B afin de modéliser la transformation de prédicats. Un transformateur de prédicats est défini par une substitution S qui associe à toute post-condition R la plus faible pré-condition, notée [S] R, permettant de garantir que R est vraie juste après l'exécution de S. Si P désigne un prédicat qui doit être respecté par l'état de la machine et si S modélise une des opérations de la machine, alors [S] R décrit la plus faible condition pour que l'exécution S se termine et établisse R. Les transformateurs de prédicats définissent la sémantique des substitutions généralisées [22]. Concernant le concept de plus faible pré-condition (weakest precondition) a été défini par E.W. Dijkstra [23].

3.1.2. Machine abstraite B

La notion de machine abstraite est similaire aux notions de classe, module, paquetage ou autres objets des langages de programmation. Elle constitue l'unité de structuration d'une spécification B. Celle-ci vise à réaliser les logiciels de façon modulaire et structurée. Elle peut être réalisée ainsi, à partir d'autres machines. L'encapsulation des données et les opérations qui les manipulent est un concept très important pour une machine abstraite. Plus précisément, cette dernière est un module, constitué d'un côté d'un état : représenté par les données et l'invariant que ces données doivent respecter ; et d'un autre côté, de la description des transformateurs de cet état : la dynamique du système est modélisée par les opérations. La notion d'encapsulation est un moyen de protéger l'état d'une machine abstraite. De ce fait, il est crucial d'introduire non seulement des opérations de modification mais aussi des opérations permettant aux utilisateurs extérieurs de consulter l'état d'une machine [16].

Structure générale d'une machine abstraite. Une machine abstraite est généralement composée clauses illustrées dans la Figure 2.2 [16].

Nous distinguons des clauses déterminant les données et d'autres les opérations. Ci-dessous, une brève description des principales clauses d'une machine abstraite :

```
MACHINE M(X,x)              //en-tête de la machine : nom et paramètres formels
CONSTRAINTS                 //définition du type et des propriétés des paramètres formels
      C
SETS                        //déclaration d'ensembles abstraits et définition d'ensembles énumérés
      S;
      T = {a,b}
CONSTANTS                   // déclaration des constantes
      C
PROPERTIES                  //définition du type et des propriétés des constantes et ensembles
      P
VARIABLES                   //déclaration des variables
      V
INVARIANT                   //définition du type et des propriétés des variables
      I
INITIALISATION              //initialisation des variables
      U
OPERATIONS                  //déclaration et définition des opérations
      u ← Op(w) =
        PRE
          Q
        THEN
          V
        END
DEFINITIONS                 //alias utilisables dans le texte des autres clauses
      D(z)=X
END
```

Figure 2.2: Structure générale d'une machine abstraite [21].

- La clause MACHINE définit le nom identifiant la machine abstraite M, suivi d'une liste de paramètres (X,x) ; où X décrit des ensembles abstraits et x des scalaires.
- La clause CONSTRAINTS définit un prédicat C permettant de typer les scalaires des paramètres et de spécifier des contraintes sur ces scalaires ainsi que sur les ensembles des paramètres.
- La clause SETS décrit les ensembles définis par la machine. Ce sont soit des ensembles abstraits (S), soit des ensembles énumérés (T) déterminés par la liste de leurs éléments. L'utilisation des ensembles abstraits a pour but de désigner des objets dont on ne veut pas définir la structure au niveau de la spécification. Tout ensemble abstrait est déterminé non vide et fini. Les ensembles énumérés sont définis par les éléments de leur énumération (a,b).
- La clause CONSTANTS déclare une liste d'éléments dont la valeur ne peut être modifiée (c).
- La clause PROPERTIES désigne un prédicat P permettant, d'un côté, de typer les constantes, et d'un autre côté de spécifier des conditions sur les ensembles et les constantes.
- La clause VARIABLES définit la liste des variables de la machine (v). Celles-ci décrivent l'état de la machine. Ce sont les seuls objets qui peuvent être modifiés au sein de l'initialisation et les opérations de la machine.

- La clause INVARIANT collecte un ensemble de prédicats (I). Ces prédicats déterminent les propriétés mathématiques des variables et permettent de typer celles-ci. Ils définissent aussi les contraintes (autre que le typage) que doit vérifier l'état de la machine à tout moment. Cette clause est inévitable si la clause VARIABLES est présente.
- La clause INITIALISATION est constituée des substitutions (U) définissant les valeurs initiales pour chaque variable propre à la machine. Toute variable propre à la machine doit être initialisée. L'initialisation doit satisfaire l'invariant de la machine. Cette clause est inévitable si la clause VARIABLES est présente.
- La clause OPERATIONS définit une liste d'opération. Ces opérations peuvent être paramétrées en entrée (w) et en sortie (u) par une liste de scalaires. Lorsqu'une opération est paramétrée en entrée, une pré-condition Q est employée pour typer les paramètres. La substitution V définit le comportement de cette opération par rapport à l'état de la machine.
- Finalement, la clause DEFINITION définit des alias (D), éventuellement paramétrés (z), sous la forme d'une expression X qui peut être utilisée dans le corps de toutes les autres clauses.

On peut utiliser les constituants des différentes clauses dans d'autres, à l'intérieur d'une même machine abstraite. La visibilité entre les constituants et les clauses est illustrée dans le Tableau2.1. Nous pouvons noter que les opérations d'une machine ne sont pas visibles au sein des autres clauses de la même machine. Des contraintes sont imposées par la méthode et alors imposées également au niveau des machines abstraites :

- Le séquencement des substitutions n'est pas permis (ceci est réservé au raffinement), seule la substitution parallèle doit être employée.
- La même chose pour les substitutions de boucle, elles ne sont pas autorisées au niveau de la spécification abstraite.

Constituants	CONSTRIANTS	PROPERTIES	INVARIANT	OPERATIONS
Paramètres	oui		oui	oui
Ensembles		oui	oui	oui
Constantes		oui	oui	oui
Variables			oui	oui
Opérations				

Tableau 2.1: Visibilité des constituants d'une machine abstraite [16].

3.1.3. Développement B par raffinement successif

Dans la méthode B, le raffinement est le processus transformateur graduel des spécifications vers des implémentations qui peuvent être traduites en langage de programmation. C'est le mécanisme qui lève le non-déterminisme et précise des choix de réalisation. Son principe est montré dans la Figure 2.3.

Soit M une machine abstraite, son raffinement R est un nouveau composant plus déterministe que M, il modifie éventuellement l'état ou les opérations de M, cependant, perçu de

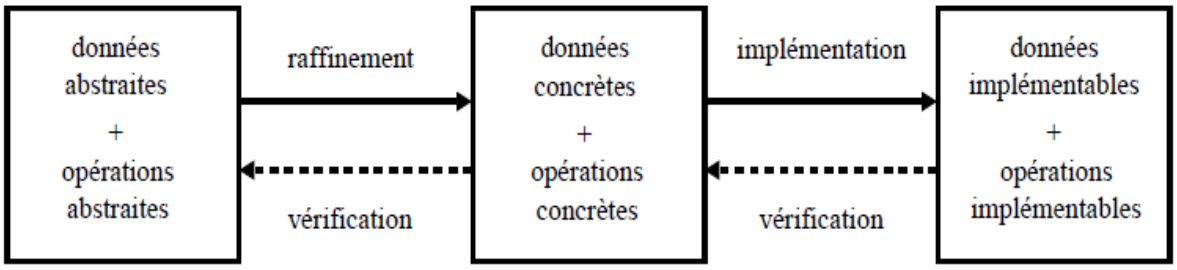


Figure 2.3: Principe général du développement B par raffinement successif [21].

3.2. Extensions de la méthode B

Primitivement, la méthode B n'a pas été introduite pour la construction des systèmes distribués. Abrial a proposé une extension [24] pour la spécification et la conception des systèmes où la distribution des exécutions occupe une place primordiale. Cette extension sert à garantir les phases amont du développement permettant de simuler mathématiquement le système global. Nous allons voir ci-dessous que ce but ne requiert pas une modification profonde de la méthode B, mais uniquement un changement au niveau des termes utilisés ainsi que de l'interprétation des opérations.

Modèle global versus modèle local. Jusqu'à maintenant, une machine abstraite n'est considérée qu'en tant que modèle du futur système informatique, ceci est appelé modèle local [20]. Afin de prendre en compte les systèmes distribués dans l'extension de B, on doit tenir principalement à la sémantique octroyée aux machines abstraites et à leurs opérations. Dans un modèle local, les opérations décrivent des fonctionnalités (ou des services) que le système fournit à ses utilisateurs. Généralement, toute opération est pré-conditionnée par une contrainte qui détermine une condition initiale nécessaire à l'exécution de cette fonctionnalité, faute de quoi, le bon fonctionnement de l'ensemble n'est pas assuré. Au fil de la réalisation du système, ces services sont raffinés successivement.

Les données des machines ainsi que le contenu de ces services peuvent être modifiés mais demeurent identiques dans leurs formes extérieures (même signature/interface).

Lorsqu'on se focalise sur le traitement de la modélisation entière d'un ensemble dynamique dans lequel divers agents sont actifs et coopèrent, dans ce cas-là, les machines abstraites sont considérées comme des objets plus importants que des systèmes informatiques (modules fonctionnels). Les machines abstraites sont devenues ici, des systèmes abstraits décrivant des composants informatiques, des composants matériels ou des éléments de communication entre les deux. On parle donc d'un autre type de modèle qui est appelé modèle global. Ceci décrit la simulation d'une certaine réalité observable dont les opérations ne jouent plus le rôle de services ou fonctionnalités, mais d'événements [21].

Evènement versus opération. La différence entre une opération et un évènement est qu'une opération est convoquée par un "agent" externe tandis qu'un évènement apparaît de façon dynamique et spontanée lorsqu'une condition de déclenchement est confirmée. A un instant donné, un seul évènement peut se dérouler dans ce modèle, dont son exécution est supposée être immédiate [21].

Nous montrons par la figure 2.4, le principe général d'un évènement défini par une opération B. Dans un évènement, la pré-condition est remplacée par une condition de garde. Chaque opération est alors gardée, et non plus pré-conditionnée, par une contrainte G qui décrit la condition de déclenchement de l'évènement E. Lorsque cette contrainte n'est pas valide, l'évènement est en instance (suspendu). Quand cette garde est vraie, l'évènement E est à nouveau activé et son action W exécutée.

La justification d'utiliser une garde au lieu d'une pré-condition est le fait qu'une opération gardée s'exécute toujours lorsque sa garde est valide, et, qu'il n'est pas obligatoire de vérifier à l'avance que celle-ci est toujours valide. En revanche, une pré-condition doit toujours être vraie. C'est ce qui doit être vérifié lorsque l'opération est appelée (avant son exécution) [16].

$u \leftarrow E(w) \;\widehat{=}$

```
SELECT
   G
THEN
   W
END
```

Figure 2.4: Structure générale d'un événement [21].

Raffinement. Nous avons supposé ci-dessus, qu'il n'y a pas de simultanéité pour l'exécution de deux événements. La granularité du temps d'exécution de l'évènement est "si infime" qu'il ne peut y avoir d'interférences entre deux évènements qui modifient les mêmes données. Cependant, on considère aussi que le temps n'est pas un médium à unité constante : il peut être abstrait et par conséquent raffiné. Son degré de raffinement est directement lié à la taille d'une unité de temps. Il est considéré qu'un seul évènement peut avoir lieu dans cette unité.

Au cours des raffinements successifs, des évènements sont affinés et de nouveaux imperceptibles au niveau abstrait sont introduits également. Généralement, quand un événement est affiné, sa garde est renforcée. Cela est dû au fait que l'observation de l'évènement est plus fine, c'est à dire que l'évènement abstrait était décrit de manière primitive (imprécise), sa description est précisée au moyen du raffinement et des évènements nouvellement créés [21].

Preuve. La méthode B a été étendue pour le but de spécifier les systèmes distribués. Cette extension appelée également B-évènementiel, n'offre pas de nouvelles obligations de preuve mais consiste à utiliser celles vues précédemment. Nous étudions ci-dessous, plus en détail les différences qui résultent de cette extension : nous distinguons entre les obligations de preuve générées par une opération (B-classique) et par un évènement (B-évènementiel).

Soit M une machine abstraite dont C représente les contraintes sur les paramètres formels, P représente les conditions les constantes et ensembles et *I* représente l'invariant. Soient O et E respectivement une opération (service), dans un développement B-classique, et un événement, dans un développement B évènementiel, d'une même machine M. L'opération O est décrite par la substitution pré-conditionnée Q | V et l'évènement E est décrit par la substitution gardée G⇒W. Soit R un raffinement de M dont C_r définit les contraintes sur les paramètres formels, P_r définit les conditions sur les constantes et ensembles et I_r définit l'invariant. Si R affine respectivement O et E par Q_r | V_r et $G_r \Rightarrow W_r$, alors les obligations de preuve générées par la méthode B sont montrées dans le Tableau 2.2 [21].

	Machine Système Abstrait(e)	Raffinement
O	$C \wedge P \wedge I \wedge Q \Rightarrow [V] I$	$C \wedge P \wedge C_r \wedge P_r \wedge I \wedge I_r \wedge Q \Rightarrow Q_r \wedge [V_r] \neg[V] \neg I_r$
E	$C \wedge P \wedge I \wedge G \Rightarrow [W] I$	$C \wedge P \wedge C_r \wedge P_r \wedge I \wedge I_r \wedge G_r \Rightarrow G \wedge [W_r] \neg[W] \neg I_r$

Tableau 2.2: Comparaison entre les obligations de preuve d'une opération et d'un évènement

On peut remarquer qu'au niveau abstrait, les obligations de preuve générées sont identiques.
Il n'existe pas de différence entre une substitution pré-conditionnée et une substitution gardée.
La vraie distinction se traduit au niveau des raffinements où l'on remarque que : [21].

- La pré-condition abstraite Q, doit être « plus forte » que celle du raffinement (concrète) Q_r, pour l'opération O.
- La garde abstraite G, doit être « plus faible » que celle du raffinement G_r, pour l'événement E.

Propriétés dynamiques. Il est possible en addition aux propriétés statiques (invariants), d'exprimer des propriétés sur la dynamique du système. On parle en général de propriétés temporelles [25] qui décrivent de quelle façon un système est autorisé à évoluer. Souvent, ces propriétés sont apparues dans la littérature sous différents noms tels que : l'interblocage, la sûreté, la vivacité, etc. Afin d'exprimer des propriétés dynamiques lors de l'extension de la méthode B, Abrial [26] propose d'étendre la notation B par des :

- invariants dynamiques décrivant des contraintes sur les transitions (entre deux états successifs)
- modalités décrivant des propriétés sur des séquences de transitions.

Les clauses représentant ces deux nouvelles propriétés sont respectivement définies par les mots clés : DYNAMICS et MODALITIES. Pour décrire des propriétés de la logique temporelle, le langage B est aussi étendu par des opérateurs (connecteurs) de cette logique. Du côté vérification, l'approche réservée consiste toujours à prouver des obligations de preuve. Ces dernières ont été particulièrement identifiées pour les clauses nouvellement crées. Julliand a proposé une autre approche d'intégration et de vérification de propriétés dynamiques [27]. Celle-ci vise à étendre le langage B par la logique temporelle linéaire et de réaliser la vérification des propriétés dynamiques du système par les « model checking ».

4. Conclusion

Dans ce deuxième chapitre nous avons définis les méthodes formelles qui sont basées sur un langage doté d'une syntaxe et d'une sémantique précises appuyées sur des fondements mathématiques, celles-ci sont utilisées pour spécifier des systèmes.

La méthode formelle B développe de logiciels prouvés, c'est un outil performant et industriel pour sa mise en œuvre dans la modularité du développement, la génération automatique du code. La transformation des modèles est définie comme étant le processus de conversion d'un modèle d'un système vers un autre modèle.

3. CHAPITRE 3
META-MODELISATION ET TRANSFORMATION DES MODELES

1. Introduction

Dans ce chapitre, nous présentons les fondements de base de l'IDM ainsi que ses deux principaux concepts : la métamodélisation et la transformation des modèles qui sont toujours des domaines de recherche. Nous examinons les concepts de base des grammaires de graphes qui ont été choisies comme formalisme sous-jacent pour exprimer la transformation de modèles.

2. Méta modèle

2.1. Définition (modèle, méta-modèle, méta-méta-modèle)

2.1.1. Le modèle

Est une description abstraite d'un système ou d'un processus, une représentation simplifiée qui permet de comprendre et simuler. La modélisation n'est pas un problème à solution unique, bien souvent, le même problème analysé par des personnes différentes conduit à des modèles différents. D'où, il n'y a donc pas de mauvais modèles, mais en revanche des modèles plus élégants que d'autre. Un modèle capture la sémantique d'un problème et contient des données exploitées par les outils pour l'échange d'information, la génération de code, la navigation, etc. [9].

- **La motivation de modélisation** : descriptive ou spécification.

Un modèle peut être utilisé pour décrire un système existant ou pour spécifier un système à construire. Dans [28], Seidewitz différencie entre les modèles descriptifs et les spécifications. La différence entre ces deux types est que dans le premier cas, le modèle est conçu par l'observation d'un système existant alors que dans le deuxième cas, le système est conçu par l'observation d'un modèle [29].

- **L'évolution du modèle** : statique ou dynamique.

Bézivin classifie les modèles en statique et dynamique selon le changement d'état du modèle au cours du temps [29]. Un modèle est statique si son état est constant ; il est dynamique si son état évolue. Il utilise cette distinction pour remarquer l'usage répandu en informatique de modèles statiques et de modèles dynamiques : le modèle en soi ne change pas, mais il représente l'évolution du système modélisé dans le temps [30].

- **Le niveau d'abstraction du modèle :** PIM (Platform Independent Model), ou PSM (Platform Specific Model)

- **Les modèles peuvent modéliser les systèmes à des degrés d'abstraction différents.**

Ils peuvent être très abstraits ou très détaillés ; très conceptuels ou très techniques. Pour distinguer les modèles selon les différences de niveaux d'abstraction, le standard MDA introduit les concepts de modèles indépendants de la plate-forme (PIM) et dépendants de la plate-forme (PSM) [4].

Cette distinction est basé sur le principe du MDA selon laquelle le cycle de développement du logiciel est un processus de transformation progressive des modèles métiers, au niveau d'abstraction haut et qui ne dépendent d'aucune technologie d'implémentation, vers des modèles plus spécifiques aux plates-formes techniques [4].

Bien que le principe de MDA soit facile à concevoir, sa réalisation n'est pas évidente.

Ceci vient de l'ambiguïté de la définition du concept de plate-forme. Cette notion est assez vague et très dépendante du contexte, ce qui rend les notions de PIM et PSM contestées [4].

[31] suggère d'autres critères de classification. Notons que, parmi les critères que nous avons abordés, les trois premiers concernent tous les modèles, alors que les trois derniers sont attribués aux modèles informatiques.

2.1.2. Le méta-modèle

Décrit de manière formelle les éléments de modélisation ainsi que la syntaxe et la sémantique de la notation qui permet de les manipuler. Le gain d'abstraction induit par la construction d'un méta-modèle facilite l'identification d'éventuelles incohérences et encourage la généricité [9].

Un méta-modèle est donc une sorte de diagramme de classes qui définit la structure d'un ensemble de modèles [4].

2.1.3. Le langage de modélisation :

Que ce soit en linguistique (langage naturel) ou en informatique (langage de programmation ou de modélisation), le langage est caractérisé par sa syntaxe et sa sémantique.

La syntaxe : décrit les différentes constructions du langage et les règles d'agencement de ces constructions (également appelées context condition).

La sémantique : désigne le lien entre un signifiant (un programme, un modèle, etc.), et un signifié (p. ex. un objet mathématique) afin de donner un sens à chacune des constructions du langage. [32].

Définition (Langage de modélisation) : Langage de modélisation est un ensemble de modèles. "A modelling language is a set of model." [33].La figure 3.1 exprime le concept de langage de modélisation en l'intégrant avec d'autres concepts définis précédemment.

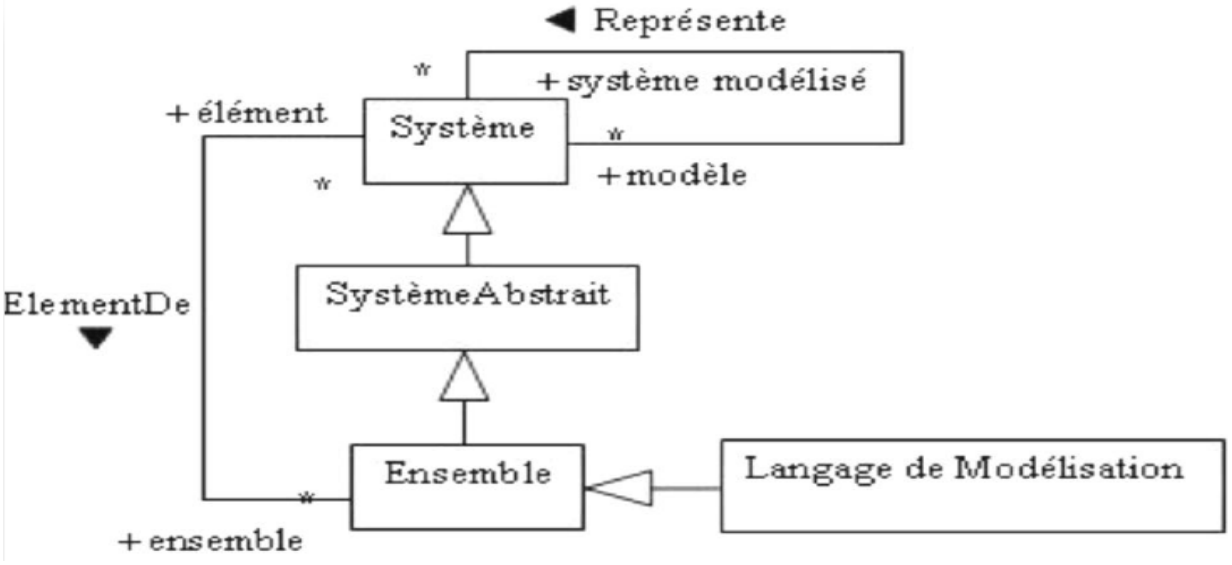


Figure 3.1: Langage de modélisation [33].

Dans sa version originale, l'approche MDA ne proposait aucun mécanisme pour ces transformations. Pour autant, pour obtenir un logiciel cohérent, il est nécessaire de s'assurer du bon déroulement de ces transformations et donc d'avoir un outil sûr pour exécuter ces constructions. Plusieurs approches ont été proposées pour cela :

- QVT (Query/View/Transformation) est aujourd'hui le standard défini par l'OMG. C'est une spécification de langage de transformations de modèles qui permet d'adapter un modèle à de nouvelles contraintes et de transformer n'importe quel langage dédié vers un autre [21].
- Les systèmes de récriture de graphes basés sur des grammaires de graphes. Plusieurs méthodes ont été développées pour faire de manière efficace des transformations de modèles à l'aide des grammaires de graphes ; on peut par exemple citer l'approche TGG ((pour triple graph grammars) introduite par Schürr) [21].

2.1.4. méta-méta-modèle

Modèle qui définit le langage pour exprimer un méta-modèle. La relation entre un méta-méta-modèle et un méta-modèle est un analogue à celle qui existe entre un méta- modèle et un modèle. [10] Les définitions précédentes définissent également le moyen utilisé pour décrire un modèle. [34] et [35] décrivent un modèle comme un ensemble d'entité et de relations. Cette définition correspond à la description de modèles par un méta-modèle réalisé avec le formalisme entité association.

2.2. La méta-modélisation

2.2.1. La définition de méta-modélisation

La méta-modélisation est l'activité consistant à définir le méta-modèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser [4].

Les langages de méta-modélisation qui servent à décrire la structure de différents formalismes tels que Méta Object Facility (MOF), Protégé, et PIR3 sont conçus pour faciliter la définition des formalismes. [36].

2.2.2. L'objectif de méta-modélisation:

La modélisation est une technique de conception de systèmes en utilisant un certain nombre de concepts prédéfinis. La méta-modélisation est une technique de définitions des concepts à utiliser pour modéliser des systèmes. Deux expressions clés récapitulant cette approche sont [4]:

-«Une tentative pour décrire le monde » :

- «Dans un objectif bien particulier». De cette définition on peut déduire les aspects suivants d'un méta-modèle :

- Il ne peut pas y avoir un méta-modèle universel utilisable pour décrire tous les systèmes informatiques.

- Un méta-modèle doit être défini pour un objectif précis. Et donc, qu'il existe une multitude de méta-modèle.

- Le dernier aspect d'un méta-modèle, en plus des concepts et des relations du domaine d'application, est la sémantique. Il est en effet important de donner un sens aux éléments définis dans un méta-modèle.

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

L'approche de méta-modélisation adoptée par l'OMG est connue comme une hiérarchie à quatre niveaux [37] (voir la Figure 3.2) :

- Niveau méta-méta-modèle (M3). M3 : est le niveau méta-méta-modèle, il définit le langage de spécification du méta-modèle. Le MOF (Meta Object Facility) [38] est un exemple d'un méta-méta-modèle.

- Niveau méta-modèle (M2). M2 : est le niveau méta-modèle. Le méta-modèle d'UML se situe à ce niveau et il est spécifié en utilisant le MOF, c'est-à-dire que les concepts du méta-modèle d'UML sont des instances des concepts de MOF. La Figure 3.2 montre deux méta-classes du méta-modèle UML : Class et Association.

- Niveau modèle (M1). M1 : correspond au niveau des modèles UML des utilisateurs.

Les concepts d'un modèle UML sont des instances des concepts du méta-modèle UML.

La Figure 3.2 montre un extrait de diagramme de classes pour une application bancaire contenant deux classes Account et Customer liées par une association UML. Les deux classes sont des instances de la méta-classe Class et le lien est une instance de la méta-classe Association du méta-modèle UML.

- Niveau objets (M0). M0 correspond au niveau des objets à l'exécution. Il s'agit ici de deux objets must et acc des instances des deux classes Customer et Account respectivement.

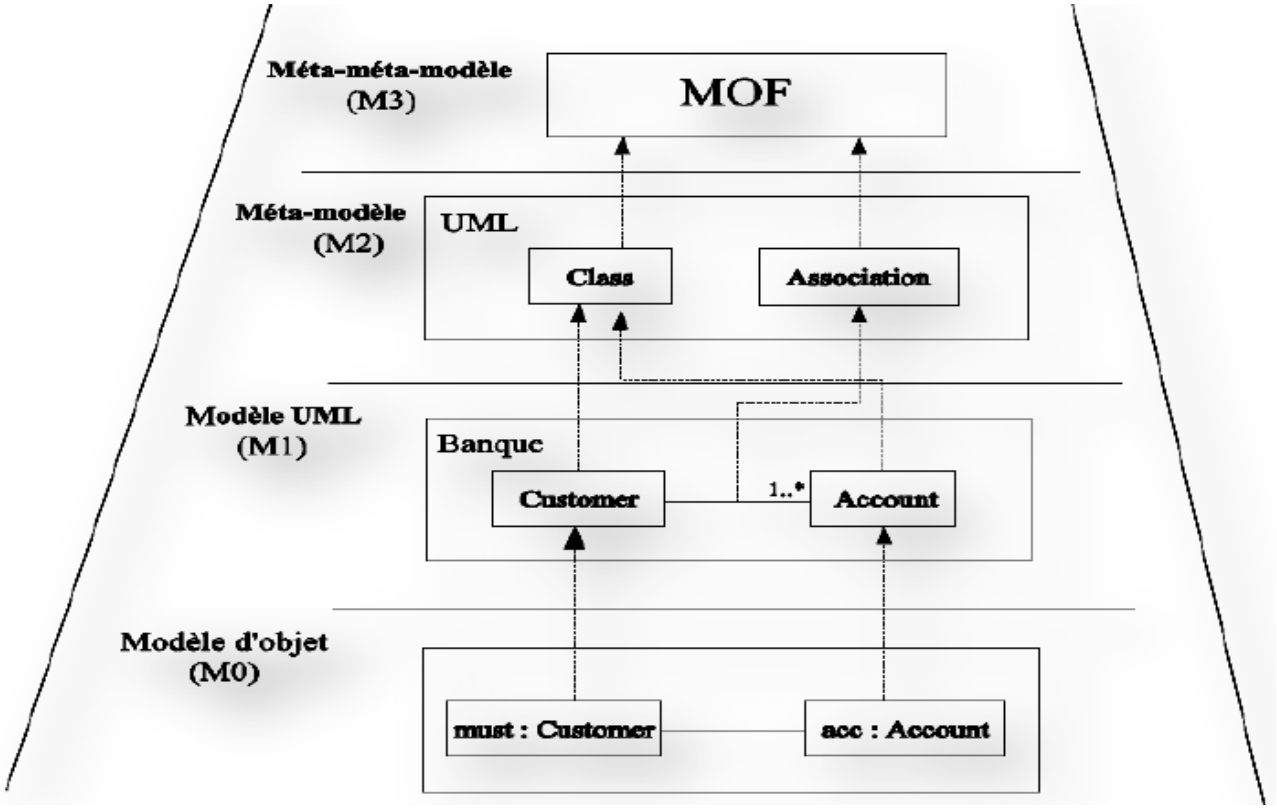


Figure 3.2: L'architecture à quatre niveaux de la méta-modélisation adoptée par l'OMG [37].

3. La transformation des modèles :

La technologie MDA (Model Driven Architecture) est un standard issu de l'OMG qui permet d'exploiter et de transformer les modèles UML.

Ceci permet notamment de :

- Produire d'autres modèles (d'analyse vers conception, par exemple),
- Produire automatiquement des documentations issues du modèle,
- Générer automatiquement du code à partir du modèle.

Une fonctionnalité essentielle de la MDA est la notion de transformation. Une transformation regroupe un ensemble de règles et de techniques pour transformer un modèle en un autre [4].

3.1. La définition transformation des modèles :

Une transformation de modèle est une fonction, t : S→ T, qui à partir d'un ensemble de modèles sources dans S crée un ensemble de modèles cibles dans T.

Les ensembles S et T sont des ensembles de modèles conformes à deux ensembles de méta-modèles. Si ces deux ensembles de méta-modèles sont identiques alors la fonction est endogène, sinon elle est exogène.

- Selon [39], la transformation des modèles est définie comme étant le processus de conversion d'un modèle d'un système vers un autre modèle.

- Une transformation de modèles peut également avoir plusieurs modèles source et plusieurs modèles cibles. Une caractéristique de transformation de modèles est qu'elle est un modèle puisque elle doit être conforme à un méta-modèle donné [39]. La figure 3.3 extraite de [39] décrit l'ensemble des concepts de la transformation des modèles.

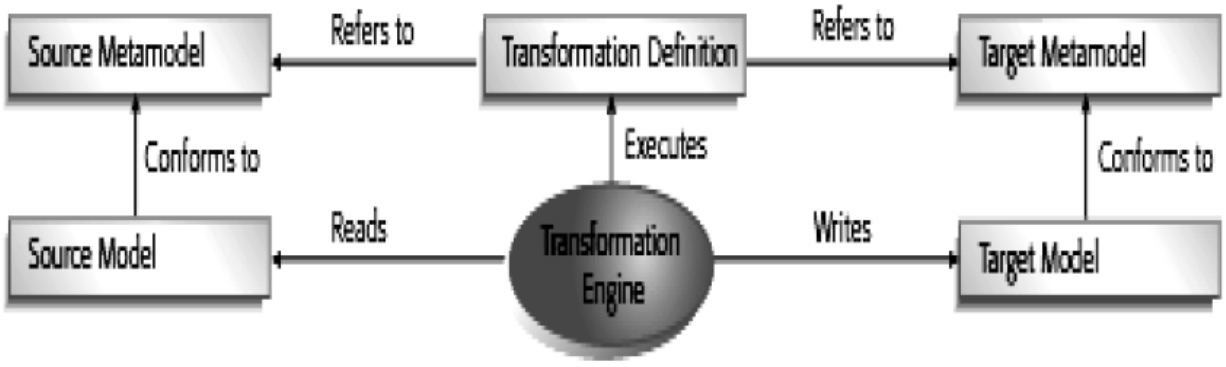


Figure 3.3: Concepts de base de la transformation de modèles [39].

[40], proposent la définition suivante pour la transformation de modèles. La transformation : est une génération automatique d'un modèle cible (Target model) à partir d'un modèle source (source Model).

D'après la définition de la transformation, La transformation est un ensemble de règles de transformation qui décrivent ensemble comment un modèle dans un langage source est transformé en modèle dans un langage cible.

Une règle de transformation est une description de comment un concept ou plus dans un langage source peut être transformé en un concept ou plus d'un langage cible.

Une transformation de modèles est une opération qui crée automatiquement un ensemble de modèles cible à partir d'un ensemble de modèles source.

La figure 3.4 représente le contexte opérationnel de la transformation de modèles [41]. Le modèle MB, conforme au méta-modèle MMB, est obtenu par l'application de la transformation MMA à MMB au modèle MA, conforme au méta-modèle MMA. De plus, en suivant le principe « tout est modèle ». La transformation elle-même est un modèle dont le méta-modèle est MMT. On dit alors que MMA à MMB est un modèle de transformation. Le langage de transformation, ou plus précisément ses concepts et leurs relations, est donc capturé par ce méta-modèle. Les trois méta-modèles MMA, MMB et MMT sont conformes au méta-méta-modèle qui est conforme à lui-même.

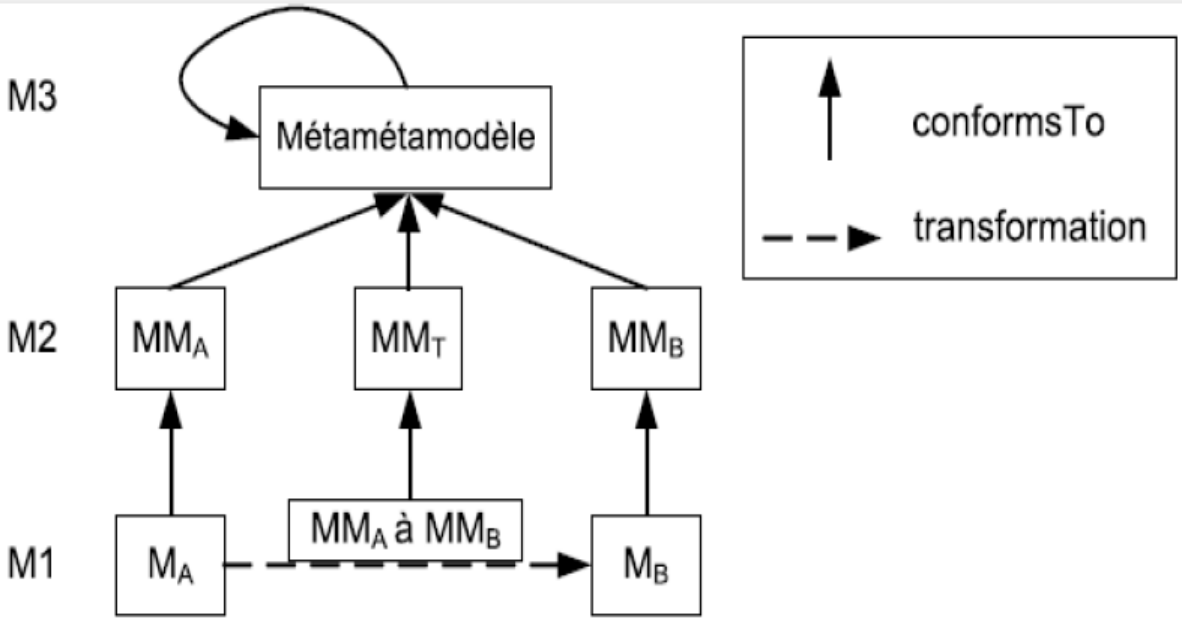


Figure 3.4: La transformation de modèles [41]

3.2. Le principe des transformations de modèles :

Le principe général présenté ici pour les transformations de modèles est indépendant de la syntaxe concrète et de la sémantique associées aux méta-modèles source et cible. Concernant les différences de syntaxe concrète, il est à noter que les transformations modèle-à-modèle et modèle-à-texte sont souvent distinguées l'une de l'autre dans la littérature. Le principe des transformations de modèles est illustré aussi par la figure 3.4.

Les transformations de modèle peuvent s'appuyer sur différentes techniques sous-jacentes telles que les « templates » (sortes de gabarits), les transformations de graphes, les grammaires de graphes triples [42], etc…

3.3. Structure d'une règle de transformation : [43]

Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible. [44]

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

3.4. Les types de transformations des modèles :

Krzystof Czarnecki et Simon Helsen ont publié une classification des différentes approches de transformation de modèles .Cette classification est basée sur un ensemble de critères. [45] [39].

Critère	Notion
Spécification	Certaines approches fournissent un mécanisme dédié pour spécifier les transformations
Règles de transformation	Unités de base de l'expression de transformation
Contrôle de l'application des règles	Stratégies de localisation des modèles et de détermination de l'ordre d'exécution des règles de transformation
Organisation des règles	Structuration, modularité et réutilisation des règles
Relation entre source et cible	Concerne l'identité des modèles sources et cibles ; sont-ils différents ou non ?
Incrémentation	Possibilité de mise à jour des modèles cibles lorsque les modèles sources correspondants changent
Directivité ou réversibilité	Détermine si la transformation est unidirectionnelle ou multidirectionnelle
Traçabilité	Mécanisme d'enregistrement des liens entre éléments de modèles sources et éléments des modèles cibles

Tableau 3.1 Critères de classification des approches de transformations de modèles [45], [39].

Différents types de transformations de modèle sont étudiés :

-La transformation verticale/La transformation horizontale/ La transformation oblique : [46] [47].

-La transformation endogène / La transformation exogène : transformer un modèle Ma en un modèle Mb, si les méta-modèles respectifs MMa et MMb sont identiques (transformation endogène), ou différents (transformation exogène) [48].

La figure 3.5 extraite de [49] récapitule les types de transformation et leurs principales utilisations.

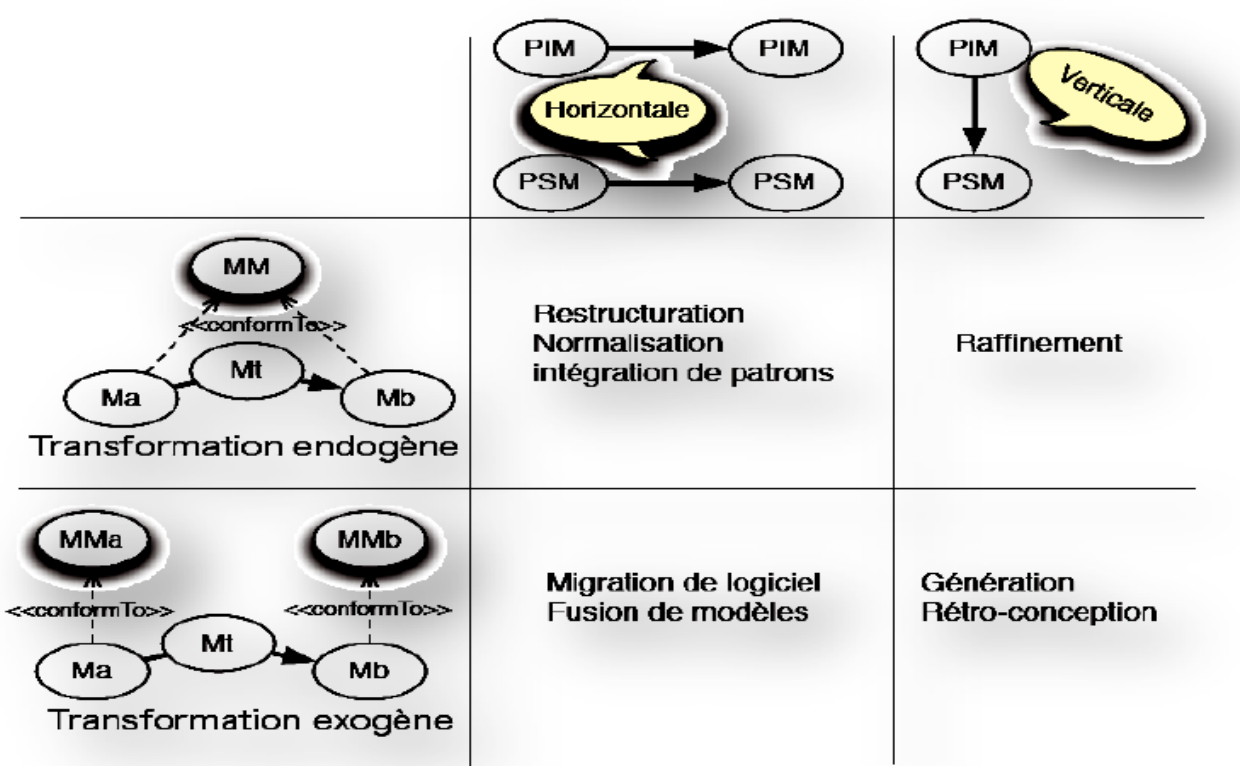


Figure 3.5: Types de transformation et leurs principales utilisations [49]

-Transformations PIM-PIM et PSM-PSM/Transformation PIM-PSM/ Transformation PSM–code/ Transformations inverses PIM-PSM et PSM-code.

-CIM (Computation Independent Model)

-PIM (Platform Independent Model)

-PDM (Platform Description Model)

-PSM (Platform Specific Model)

Plusieurs types de transformations de modèle sont définis dans la MDA.

-Transformations PIM→PIM : PIM vers PIM

-Transformations PIM→PSM : PIM vers PSM

-Transformation PSM→code

-Transformations inverses PIM←PSM et PSM←code

La figure 3.6 réunit les modèles ainsi que les transformations dans l'approche MDA.

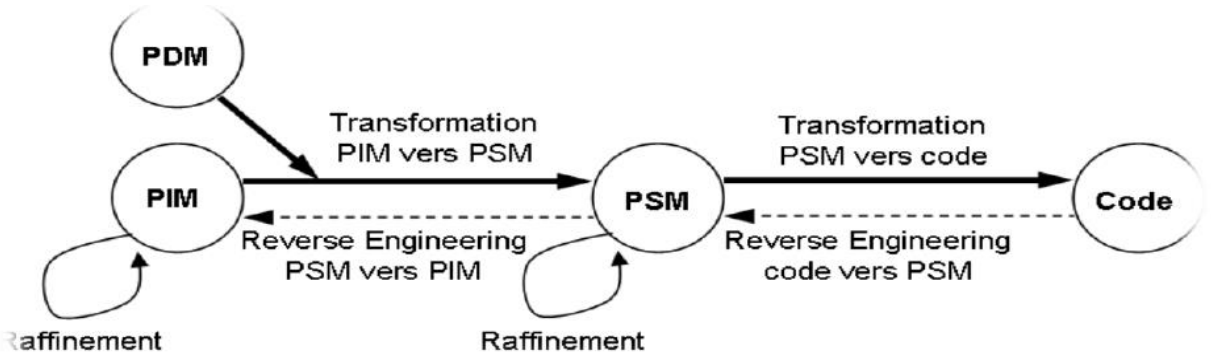


Figure 3.6: Modèles et transformations dans l'approche MDA

-Transformation modèle vers modèle / modèle vers code [45].

-Les transformations relationnelles.

-Les transformations des graphes.

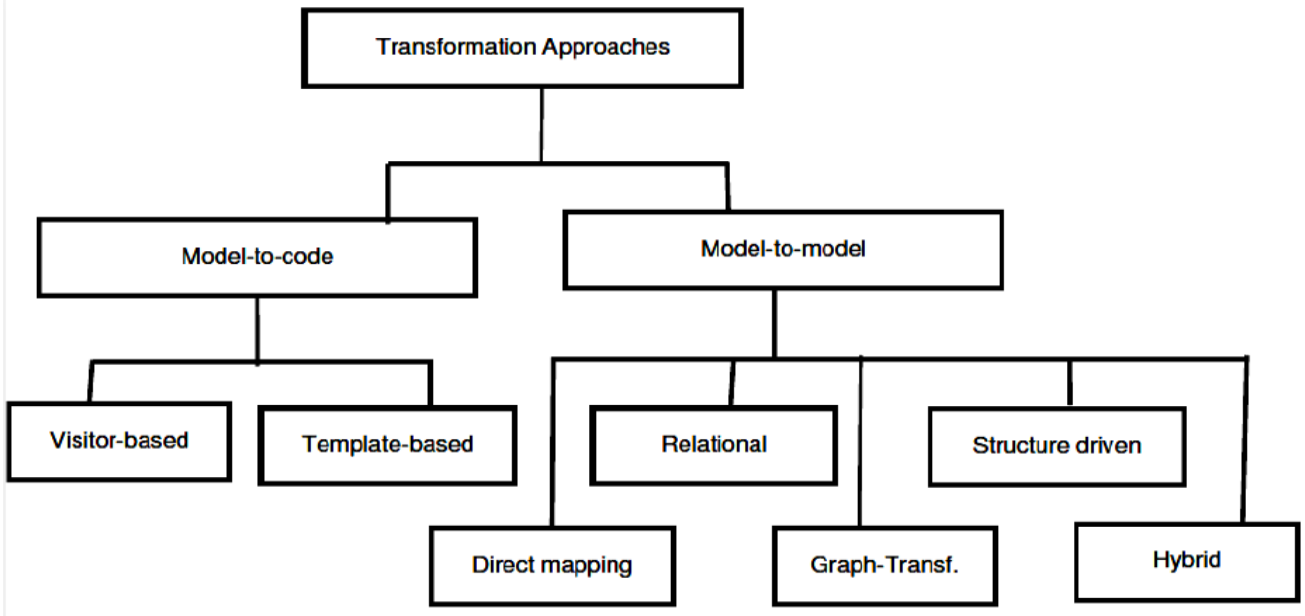


Figure 3.7: Approches de transformations de modèles (modèle vers modèle / modèle vers code) [45].

4. Etude comparative sur les outils de transformation de graphe

4.1. Tableaux de comparaison

Le Tableaux suivant 3.2 présente une étude comparative d'un ensemble d'outils.

Critère	AGG	ATOM3	GROOVE	GrGen	Fujaba	EMF Tiger	Viatra
Nombre de source et cible	One to one	One to one	One to one	One to one	Many to many	One to one	Many to many
Type de transformation	Endogène	Endogène	Endogène	Exogène	Endogène	Exogène	Endogène et exogène
Approche utilisé	SPO et NAC	Grammaire de graphe	SPO et NAC	SPO et DPO	TGG	SPO	TGG
Intégrité de contrainte	Non supporté Présenté par NAC ou Java	Non supporté Présenté par OCL ou Python	Non supporté Présenté par NAC	Non supporté Présenté par NAC	Non supporté Présenté par OCL	Non supporté Présenté par NAC	Supporté Graph pattern
Editeur	Graphique et textuel	Graphique	Graphique	Textuel	Graphique	Graphique	Graphique pour les modèles et textuel pour les règles
Degré d'automatisation	Grammaire de graphe	Grammaire de graphe	Grammaire de graphe	Optimisation automatique de règles de transformation	Histoire axée modélisation	Simulation automatique	Transformation entraînées par la machines d'état abstraites
Processus de transformation	Automatique	Automatique	Automatique	Automatique	Semi-automatique	Automatique	Automatique
Réutilisabilité	Transformation paramétrées				Transformation paramétrées/héritage de transformations	Transformation configurable	Réutilisation des motifs prédéfinis
Vérification	Exprimé par les couches de règles	Priorité de règles	Priorité de règles	(GRS) pour le contrôle des règles de transformation	Analyse automatique et une vérification cohérente	Contrôles de fin de contrat et l'unicité des résultats de la transformation.	Vérification à part entière
Validation	L'analyse des paires critiques, la résiliation, la	Typage correct du modèle cible	Valider avec le model checking de groove	Contrôle les règles de réécriture contenant les tests arbitraires	Techniques de tests unitaires sont utilisées	Confluence, terminaison, analyse des paires critiques	″Typage correct+ Préservation des modèles de graphes″ valider les

	préservation de contraintes graphiques					de règles	transformations avec des techniques de model checking
Composition	Niveaux de priorités (layers) règle	Priorité de règle	Priorité de règle	Graphe Rewrite Séquence (GRS)	Appel aux méthodes de modélisation de l'histoire conduit ¨story diagrams¨	Niveaux de priorité Procédural	Composition non récursive de modèle dans les règles 'machine d'états'
Complexité	Niveaux de grammaires	Des couches avec priorités, le séquençage par priorité. L'exécution en parallèle des matches non chevauchés			Transformation de graphes contrôlés	Respecter les contraintes induites par la sémantique d'agrégation	D'ordre supérieur et méta transformations
Standardisation	GXL, GTXL	XMI	GXL	XMI	UML, Java, XMI, MOF	ECORE	XMI, MOF, UML
Type d'utilisation	Usage général	Transformation de modèle	Usage général	Haute performance	Haute performance	Transformation de modèle	Transformation de modèle
Typage	Requis	Requis	Optionnel	Requis	Requis	Requis	Requis
Contrôle	Priorité	Priorité	Impératif/Priorité	Impératif	Impératif	Priorité	Impératif
Exploration	Linéaire	Linéaire	Multiple	Linéaire	Linéaire	Linéaire	Linéaire

Tableau 3.2: Tableau de comparaison [3].

5. Grammaires de graphes triples (triple graph grammar TGG)

5.1. Introduction

L'approche des grammaires de graphes triples (Triple Graph Grammar : TGG), introduit par Andy Schürr [50] est une tentative de créer une méthode pour connecter différents systèmes/modèles par rapport à certains règles/critères prédéfinis, de sort que les changements dans un système/modèle conduirait inévitablement à des changements dans l'autre. TGG peut être utilisé dans différents transformation de modèles et les scénarios de synchronisation. TGG est spécifié pour les transformations bidirectionnelles (transformation à l'avant et en arrière) [3].

5.2. Fondements théoriques de TGG

Afin de mieux comprendre TGG, nous explicitions à partir de [51], les notions de base pour TGG.

5.2.1. Un Graphe

Est défini comme G= (V, E, s, t) on V ensemble de sommets (nœuds) et E ensemble des Arêtes et s,t : E$\longrightarrow$V sont des fonctions qui attribuent les sommets source et cible aux arêtes [3].

5.2.2. Un morphisme de graphe

De G à G' est une paire h= (h_V, h_E), ou h_V= V $\longrightarrow$V', h_E= E $\longrightarrow$ E' sont définis de telle sorte qu'ils ¨préservent¨ les nœuds sources et cibles [3].

5.2.3. Un Grammaire de graphe

Est une paire G= (G_0, P), ou $G_0 \in$ G est l'axiome et P= {p: (L$\overset{l}{\rightarrow}$ K$\overset{r}{\rightarrow}$R)} est une ensemble de règles de transformation [3].

5.2.4. Une production de graphe monotone

Est une paire de graphes p= (L,R), ou L $\subset$ R [3].

5.2.5. Une production de graphe

P est applicable à un graphe G s'il existe un morphisme h: $L \longrightarrow G$ [3].

5.2.6. Un triple de graphes

G= (SG$\xrightarrow{h_{SG}}$ CG $\xrightarrow{h_{TG}}$ TG), ou SG graphe source, CG, graphe de correspondance et TG graphe cible, h_{SG} est un morphisme de graphe entre les graphes SG et CG h_{TG} est un morphisme de graphe entre les graphes CG et TG [3].

5.2.7. Un triple de productions

Est un structure p= (sp$\xrightarrow{h_{sp}}$ cp $\xrightarrow{h_{tp}}$ tp), ou sp production source, cp production de correspondance, tp production cible, h_{sp} est un morphisme de graphe entre production source et de correspondance et h_{tp} est un morphisme de graphe entre production de correspondance et cible [3].

5.2.8. Un triple de production

P est applicable à un triple de graphe G si les composants de production sont applicables sur les composants de graphe. L'application de p donne un triple de graphe G' de telle sorte que les composants de productions sont appliqués sur les composants de graphe [3].

5.2.9. Un triple de production donné

P= ((SL,SR)$\xrightarrow{h_{sp}}$ (CL,CR) $\xrightarrow{h_{tp}}$ (TL,TR)) peut être devenir une production de source local P_{sl} = (SL,SR) et source-cible production p_{st} =((SR,SR)$\xrightarrow{h_{sp}}$ (CL,CR) $\xrightarrow{h_{tp}}$ (TL,TR)) ou p = $p_{sl}p_{st}$ [3].

5.3. Processus de transformation des modèles avec TGG

Le formalisme TGG a été conçu pour générer et transformation des paires de graphes connexes (généralement appelés graphe source et cible) en vertu d'un mécanisme de synchronisation bien formé (par exemple, un graphe de correspondance) qui permettrait de préserver les relations dans les graphes mis en correspondance après l'application d'une transformation. La motivation initiale du formalisme était de fournir une modélisation et une spécification d'outil graphique de haut niveau pour les problèmes impliquant des diagrammes connexes (arbres de syntaxe, diagrammes de contrôle) et les structures d'information (les exigences, les documents de conception et la traçabilité). Plus récemment les concepts clés du formalisme TGG ont été utilisés dans le langage de transformation standard de l'OMG, pour spécifier les « correspondances » d'une transformation. En outre, les applications de TGG dans le domaine des spécifications de transformations bidirectionnelles de modèles sont de plus en plus fréquentes [3].

6. Travaux connexes

-Dans l'article [52], Ximeng étudie la possibilité de modéliser les diagrammes de séquence « Sequence Diagram », de les transformer automatiquement vers les Statecharts (diagrammes d'états-transitions), et de générer automatiquement de texte de conditions des diagrammes de séquence .Il a réalisé un outil de modélisation qui convertit les diagrammes de séquence vers les diagrammes d'état transition et de générer du texte en s'appuyant sur une transformation constituée de quarante règles.

-Vangheluwe crée un méta-modèle pour modéliser la syntaxe de réseau du trafic de véhicule, en s'appuyant sur le formalisme E/R [53].

La sémantique a été attribuée par la transformation des modèles de trafic vers le modèle de Pétri Nets (mapping Traffic models onto Petri Net models).

-Dans El Mansouri, ils ont proposé une approche basée sur la combinaison de la méta-modélisation et la grammaire des graphes afin générer un outil visuel pour modéliser les ECATNets ; les ECATNets représentent une catégorie de réseau de pétri algébrique. Dans leur progression, ils sont créé un méta-modèle pour ECATNets en utilisant le formalisme des digrammes de classe d'UML [54].

-Christian Attiogbé, Pascal Poizat, Gwen Salaün, ils sont proposés une approche générique pour intégrer des données exprimées dans des langages de spécification formelle à l'intérieur de diagrammes d'états d'UML. Les motivations principales sont d'une part de pouvoir modéliser les aspects dynamiques des systèmes complexes avec un langage convivial et graphique tel que les diagrammes d'états d'UML ; d'autre part de pouvoir spécifier de façon formelle et potentiellement à un haut niveau d'abstraction les données mises en jeu dans ces systèmes à l'aide de spécifications algébriques ou de spécifications orientées état (telles que celles écrites en Z ou en B) [55].

7. Conclusion

Dans ce chapitre nous avons exposés les définitions du concept modèle, méta-modèle et méta-méta-modèle. Ainsi que la transformation des modèles, qui représente une discipline importante dans MDA.

Parmi ces types de transformations, la transformation de modèle vers modèle. En utilisant une grammaire de transformation TGG (triple graph grammar).Avec TGG on peut définir, déclarer une transformation bidirectionnelle. Un modèle peut être transformé en un autre modèle en se basant sur un modèle de correspondance.

4. CHAPITRE 4
L'APPROCHE PROPOSEE

CHAPITRE 4 : L'APPROCHE PROPOSEE

1. Introduction

Dans ce chapitre, nous présentons l'environnement d'implémentation des transformations de modèle ainsi que l'étude de cas.

Nous proposons notre approche intégrée qui sert à traduire des modèles Statechart vers des spécifications formelles en langage B, en s'appuyant sur des règles de réécriture de graphes.

2. Environnement d'implémentation

2.1. Eclipse

Eclipse est une plateforme universelle d'intégration d'outils de développement. Il s'agit en effet d'un IDE (Environnement de Développement Intégré) ouvert, facilement extensible et qui n'est pas lié spécifiquement à un langage de programmation. Cette plateforme fournit à la base, un ensemble de services pouvant être éventuellement enrichis par le biais de plugins [56].

2.2. Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) est un framework Java de modélisation et un outil de génération de code pour construire des applications basées sur des modèles. Depuis un modèle de spécification décrit en XMI, EMF fournit des outils et un support de moteur d'exécution pour produire des classes Java. De plus EMF permet de stocker les modèles sous forme de plusieurs fichiers reliées. Les modèles peuvent être spécifiés en utilisant des documents Java, UML, XSD, XML, puis sont importés dans EMF. Par contre EMF ne propose pas d'éditeur graphique pour la modélisation. Le plus important est qu'EMF fournit les fondements à l'interopérabilité avec d'autres outils ou applications basés sur EMF. EMF utilise le langage Ecore qui est un langage de métamodélisation graphique et textuelle défini par IBM [57].

2.3. Ecore

Le langage Ecore est un langage de métamodélisation graphique et textuelle défini par IBM et utilisé dans le framwork de modélisation d'Eclipse EMF. Il est utilisé pour définir les méta-modèles et le langage OCL est intégré pour d'écrire les contraintes dans le but de vérifier la conformité des modèles à leurs méta-modèles [58].

2.4. TGG Interpreter

Le TGG Interpreter a été développé pour la transformation de modèles TGG et est un outil de mise à jour incrémentale résultant de la comparaison de TGG et de la norme de l'OMG bidirectionnel QVT (Query/View/Transformation), pour la transformation de modèles. Le TGG est basé sur Eclipse et peut être installé via l'Update Manager Eclipse [59].

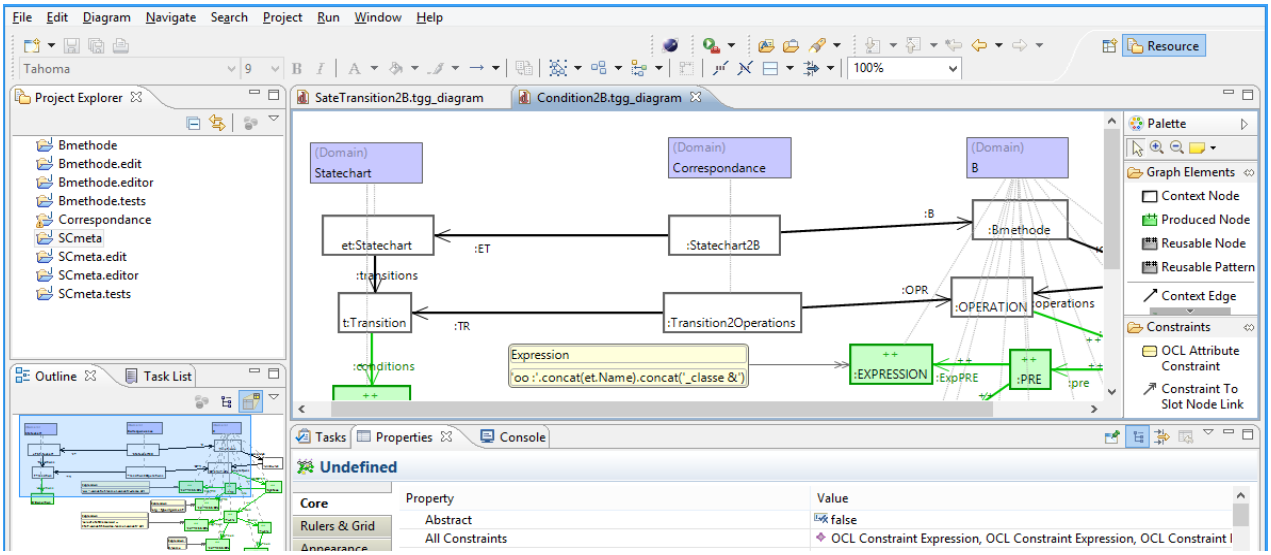


Figure 4.1: Editeur de règle de TGG Interpreter

Plusieurs vérifications sont réalisées statiquement pour éviter les erreurs de modélisation. Cependant, il n'y a pas de soutien pour l'analyse de pair critique afin d'identifier les règles qui peuvent être contradictoires. Un éditeur GMF est prévu pour l'édition de règle TGG ainsi que la fonctionnalité de commodité, par exemple la création des nœuds de correspondance sur le champ pour les nœuds de correspondance. Les transformations peuvent également être exécutées via un appel d'API.

Expressivité : des conditions d'application et de contraintes d'attributs peuvent être formulés en OCL, mais les relations bidirectionnelles sur les valeurs d'attributs doivent être exprimées en des affectations dans la direction vers l'avant et vers l'arrière. Pas de fortes restrictions sont imposées sur la structure de règles TGG, à savoir, les modèles ne doivent être faiblement connectés, des arcs peuvent être créés entre les nœuds de contexte, et des nœuds de correspondance peuvent être connectés aux éléments de source et aux éléments de cibles. TGG Interpreter soutient des concepts avancés tels que l'héritage de règle, les stéréotypes dans les domaines d'UML, et des nœuds et des modèles réutilisables pour conserver les informations dans le cas incrémentale [60].

Propriétés formelles : TGG Interpreter support des fonctionnalités avancées telles que les liaisons de bords explicites. En outre, TGG Interpreter est interprété pour effectuer la transformation, sans avoir recompilé ou calculer quoi que ce soit. Il prend en change les mises à jour incrémentielles. En ce qui concerne l'algorithme de contrôle. TGG Interpreter nécessite un nœud de départ désigné, il ne permet qu'un seul axiome dans le TGG, et nécessite un comportement fonctionnel pour éviter le retour en arrière. Le modèle d'entrée est traversé par un parcours de l'ensemble des nœuds traduits, en examinant tous les éléments qui peuvent être par l'intermédiaire d'un seul « edge ». Seuls ces éléments sont pris en compte dans la prochaine étape limitant la recherche de règles applicables à seulement ceux qui ont besoin de ces éléments. Le processus se termine lorsqu'il n'y aura plus de règles qui peuvent être exécuté [61].

2.5. Xpand

Xpand fait partie des outils *Model to Text* (M2T) d'Eclipse EMF. Il s'agit de transformer un modèle en fichier texte, ces fichiers constituants souvent les sources d'une application java (Eclipse oblige). Cependant le format de sortie importe peu, et tout est imaginable tant que c'est du texte. Par exemple, dans cet article, on génèrera des fichiers HTML (plus simple et plus visuel que du code source). Dans la même famille d'outils (M2T) on trouve Acceleo (édité par les français d'Obeo), ainsi que JET (méthode et concepts plus anciens, à déconseiller pour de nouveaux projets) [62] [63].

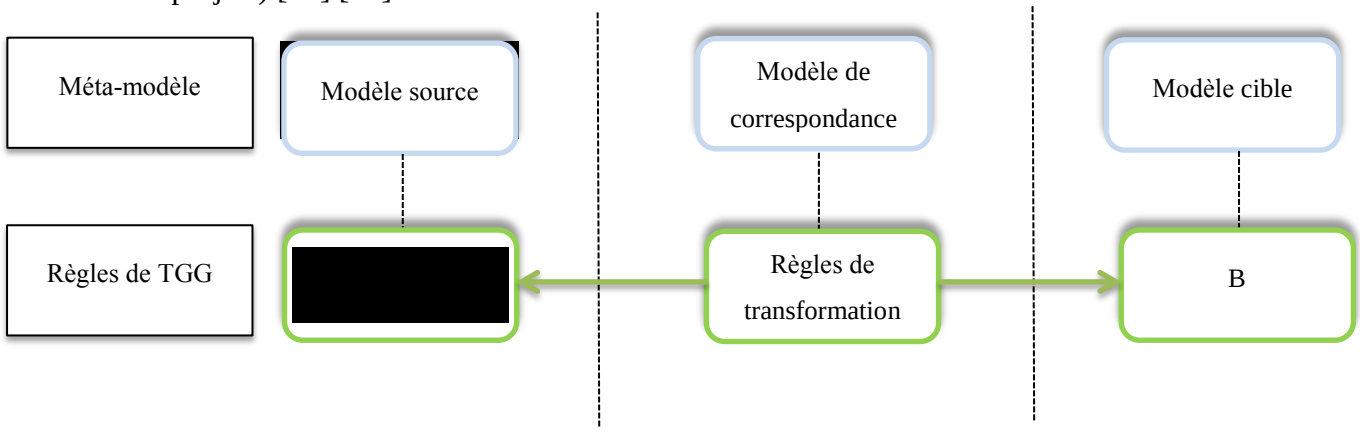


Figure 4.2: Principe de transformation avec TGG.

2.6. Spécification de règle de transformation

La transformation de modèle se base sur la définition des règles de transformation ou une bonne transformation de modèles dépend d'une bonne définition des règles de transformation.

TGG se compose de trois graphes ou chaque graphe appartient à un domaine.

Chaque domaine contient une règle de grammaire de graphe. Habituellement, le domaine sur la gauche est appelé source et le domaine sur la droite est appelé cible. Le domaine de correspondance relie le domaine source et cible.

Chaque domaine est attribué à un métamodèle. Les nœuds de ce domaine sont les classes de ce métamodèle.

Avant de spécifier les règles, on doit définir les différents nœuds qu'on va rencontrer dans les règles. Il existe différents nœuds de règles de transformation: nœud de contexte, nœud de production, nœud réutilisable et les contraintes.

2.6.1. Nœud de contexte

Parfois, seule une partie d'un modèle est pertinente et doit être transformée. Ensuite, les règles de TGG devaient être conçues pour les parties pertinentes de la transformation. Si les pièces d'un modèle moins pertinent influencent la transformation, ils peuvent apparaitre comme des nœuds de contexte dans les règles de TGG. La présentation graphique d'un nœud de contexte est présentée d'une case avec un contour noire.

Les nœuds contexte doivent être appariés avec les objets du modèle précédemment traités. Cela signifié que le TGG, tel qu'il est présenté à ce jour, besoin de spécifier une grammaire complète pour toutes les parties de modèle. [64]

2.6.2. Nœud de production

Les nœuds de production sont affichés sous forme de boîtes vertes avec une bordure verte et un label "++". Les arrêtes de production sont affichés sous forme de flèches de couleurs vert foncé avec une étiquette "++".

Les nœuds de production ne doivent pas correspondre à nœud qui existe déjà. Si on trouve ces éléments dans le domaine source, les éléments de modèle cible et de modèle de correspondance peuvent être créés selon la règle. Tous les objets créés sont liés aux nœuds de contexte de la règle.

En conséquence, un objet de modèle ne peut pas être présenté comme un nœud de production qu'une seule fois. Nous appelons cela sémantique bind-only-once pour les de nœuds de productions [65].

2.6.3. Nœud réutilisable

A noter les types de composants ne doit pas être générés, mais peuvent être réutilisés encore et encore depuis des nœuds déjà existants avec les propriétés requises. Par conséquent, nous appelons ces nœuds "les nœuds réutilisables". Graphiquement, ces nœuds représentés en gris avec []. La sémantique des nœuds réutilisables est qu'ils peuvent être nouvellement générées ou réutilisés de façon arbitraire. La couleur gris reflète le fait que, sémantiquement, chaque nœud réutilisable pourrait être soit en noir (un nœud du côté gauche de la règle TGG, à savoir qu'il réutilisé) ou en vert (un nœud du côté droit de la règle de TGG, à savoir qu'il est nouvellement généré), qui peut être choisi à chaque fois que la règle appliquée. Cette interprétation montre en fait que, les nœuds réutilisables ne sont pas strictement nécessaires. [65]

2.6.4. Les contraintes

La contrainte réelle dans un nœud de contrainte peut être toute expression OCL qui réfère à des objets qu'il est attaché. Les restrictions sont seulement nécessaires pour les mettre en œuvre des transformations ou des synchronisations plus efficace.

La présentation graphique d'une contrainte OCL représentée avec la couleur jaune. Une contrainte OCL doit être attachée avec un autre nœud.

Théoriquement, les grammaires de graphe peuvent être utilisées pour définir l'évolution dynamique d'un modèle unique. Les TGGs nous permettent de définir la relation entre différent domaines [67].

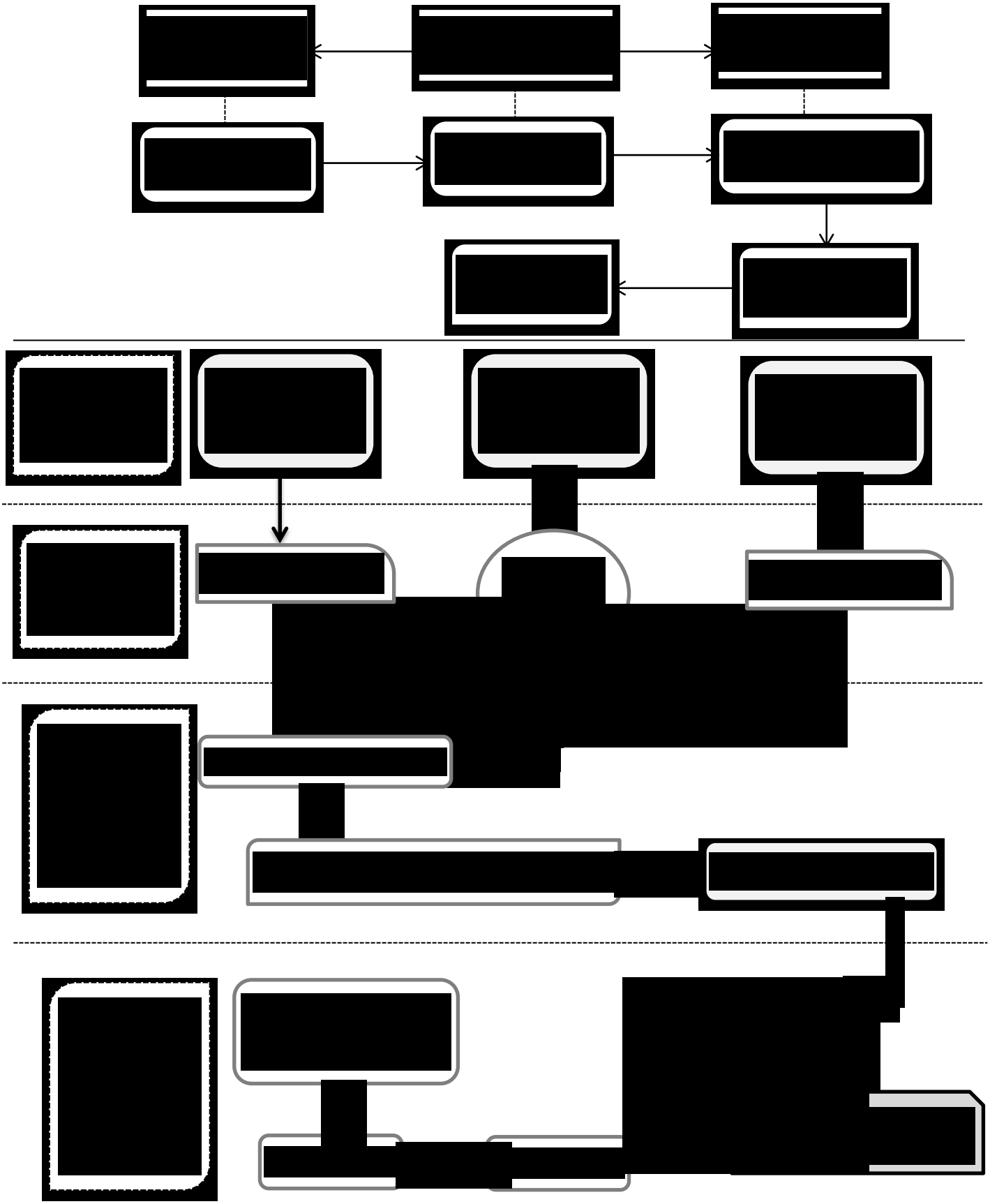
2.7. Transformation du modèle

L'application du scénario le plus évident de TGG est la transformation d'un modèle à un autre modèle. Dans ce scénario, l'un des deux modèles existe déjà, et l'objectif est générer un modèle correspondant.

TGG est neutre par rapport à la direction de la transformation ou il ne dépend que de ce qui appelé le modèle source ou le modèle cible. Parfois, il n'est même pas clair, ce qui devrait être la source et ce qui devrait être la cible, de sort que nous appelons simplement les domaines.

Ensuite, on doit indiquer quel domaine doit être utilisé en tant que source et cible pour une transformation [3].

3. Schéma descriptif de l'approche



3.1. Méta-modèle de statechart

Nous illustrons par la figure 4.3 le métamodèle du statechart diagrams (diagramme d'état-transitions). Ce méta-modèle dispose d'une classe pour l'élément racine *Statechart* décrivant le point de départ de ce méta-modèle pour le diagramme d'états-transitions. Elle est composée de deux autres classes fondamentales : *State* et *Transition* qui décrivent respectivement l'état et la transition dans un diagramme d'états-transitions.

La classe état peut être source ou bien cible pour une transition, selon le type d'associations dénommées respectivement source et *target*. Une action d'entrée *EntryAction* et une action de sortie *exitAction*.

La classe Transition est composée de deux classes décrivant la condition de déclanchement *Condition* et enfin l'action *Action*.

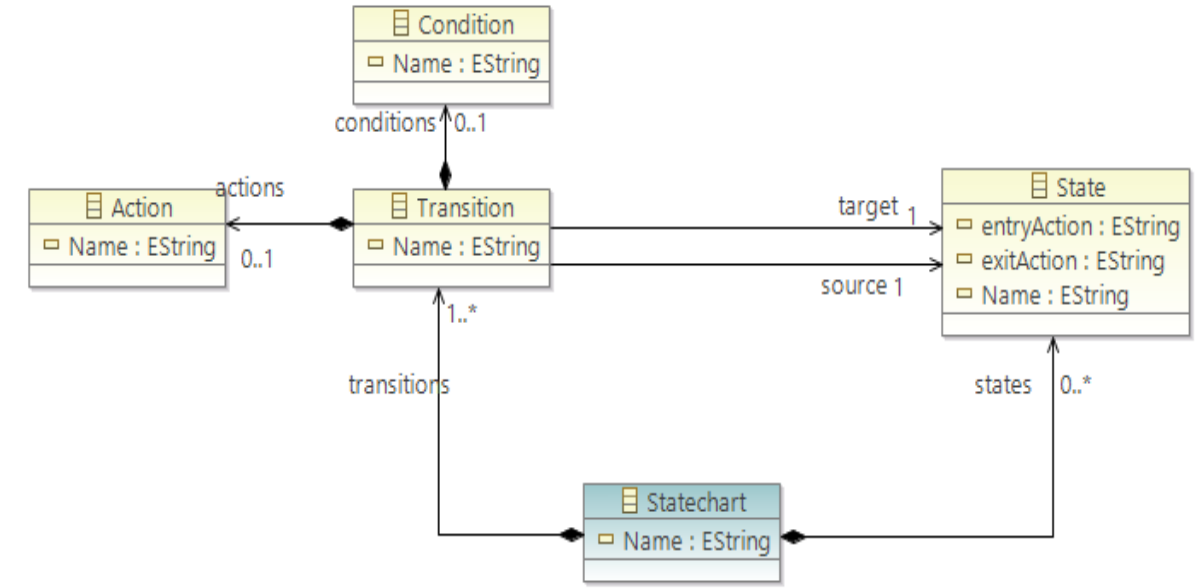


Figure 4.3: Méta-modèle de Statechart.

3.2. Méta-modèle du langage B

Le méta-modèle de la méthode B est bien illustré par la figure 4.4.

La classe racine est appelée Bmethod représentant un projet B. Ce récipient se compose de quatre classes baptisées MACHINE, REFINEMENT, IMPLEMENTATION et Types décrivant respectivement les machines abstraites, les raffinements, les implantations et les types non prédéfinis par le langage B dans un projet.

CHAPITRE 4 : L'APPROCHE PROPOSEE

La classe fondamentale *MACHINE*, est une clause composée de cinq classes qui représentent la structure générale d'une machine sous forme de deux parties statiques et dynamiques d'une celle-ci : *Data, USES, Predicate, INITIALISATION*, et *OPERATION*. Data englobe les données B sous forme de clauses *CONSTRAINTS*, *CONSTANTS*, *SETS* et *SEES*, *USES* représente une clause B pour exprimer la relation d'utilisation entre les machines abstraites, Predicate dérive les clauses B de variables (*VARIABLES*), des invariants (*INVARIANTS*) ainsi que des propriétés (*PROPERTIES*) ; *INITIALISATION* pour décrire la clause d'initialisation des variables de l'état B, et enfin, *OPERATIONS* afin d'exprimer les opérations des machines abstraites. Chaque classe comprend une autre classe EXPRESSION pour indiquer l'expression de chaque clause.

Concernant la classe *REFINEMENT* est composé de deux classes *INCLUDES* et *REFINES* représentant respectivement les clauses d'inclusion et de raffinement d'autres machines voire d'autres raffinements.

IMPLEMENTATION se compose ainsi de *REFINES* (c'est un raffinement particulier) et de la classe *IMPORTS* qui décrit la clause d'importation d'autres modules (machines abstraites, raffinements ou même implantations).

La classe Types comprend un composant appelé *SETS* représentant une clause B et exprimant les types non prédéfinis sous forme d'ensembles (en se basant sur la théorie des ensembles) [21].

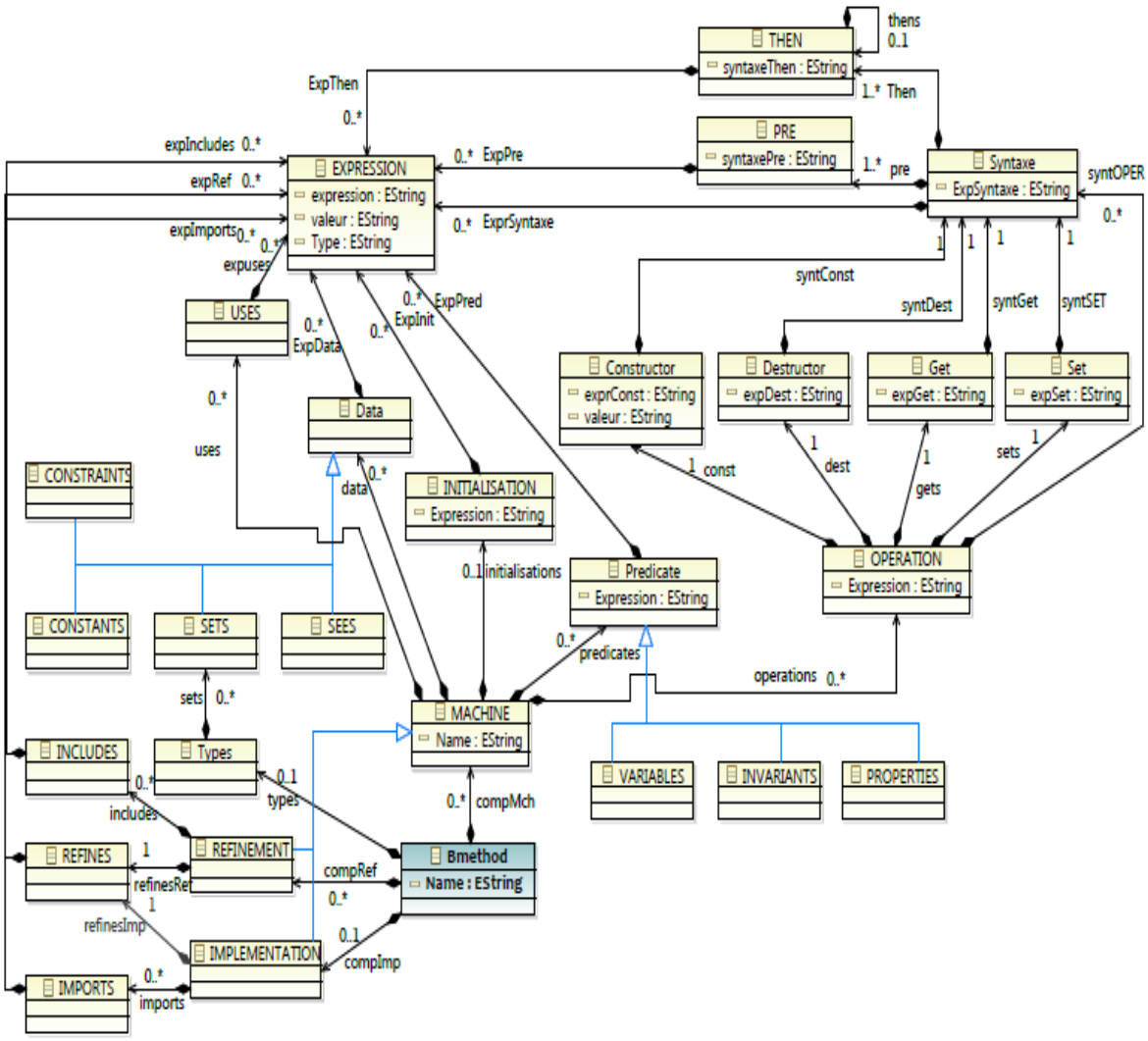


Figure 4.4: Méta-modèle de B.

3.3. Méta-modèle de correspondance

Dans la figure 4.5, nous illustrons le méta-modèle de correspondance entre les deux méta-modèles de diagramme d'états-transition et la méthode B. Nous relions les éléments du métamodèle de diagramme d'états-transitions et celui de la méthode B par des nœuds de correspondances. Ci-dessous, leur description :

– *Statechart2B* : décrit la correspondance entre les racines des deux méta-modèles de diagramme d'états-transitions et de la méthode B (respectivement *Statchart2B* et *Bmethod*).

– *State2Sets* : indique la correspondance entre l'état dans les diagrammes d'états-transitions et l'élément équivalent dans la machine abstraite B au sein de leur méta-modèle approprié (respectivement *State* et *SETS*).

– *Transition2Operations* : exprime la correspondance entre la relation de transition dans les diagrammes d'états-transitions et son équivalent dans la machine abstraite B au sein de leur méta-modèle approprié (respectivement *Transition* et *OPERATIONS*).

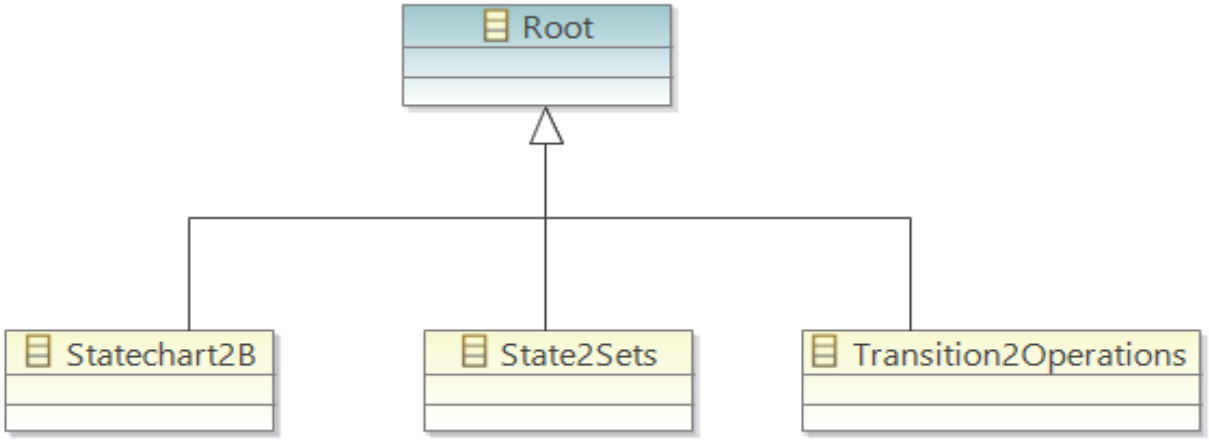


Figure 4.5: Méta-modèle de correspondance entre le Statechart et B.

3.4. Règles TGG pour la transformation

3.4.1. Règle 1 : Axiom (Statechart2B rule1)

C'est le point de départ de toutes les règles de transformation du statechart (diagramme d'état-transition) vers la méthode B en TGG-Interpreter. Cet axiome est montré dans la figure 4.6. Il a pour objet d'exprimer la première transformation en préparant par défaut, dans sa partie RHS et au sein d'une machine abstraite B déjà générée pour une classe, l'emplacement où on doit intégrer le code B à dériver (*SETS*, *VARIABLES*, *INVARIANTS*, *INITIALISATION* et *OPERATIONS*). Cela est réalisé à partir de l'élément racine *Statechart* du méta-modèle de la partie LHS.

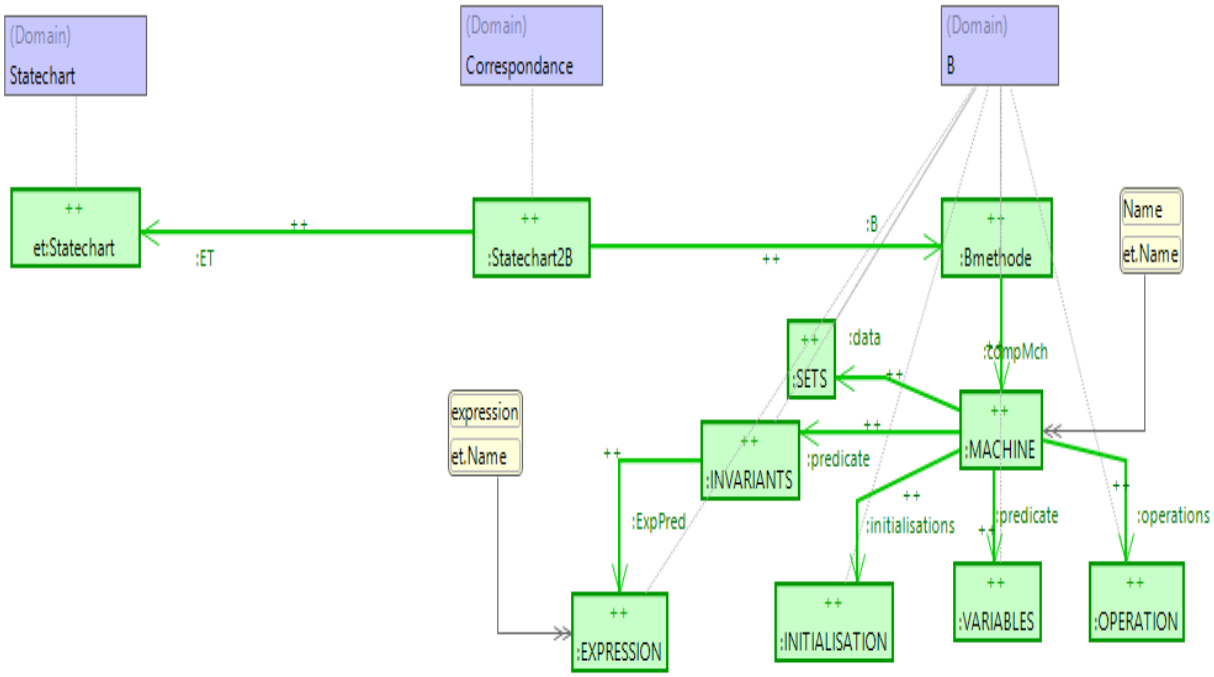


Figure 4.6: Règle de l'axiome.

3.4.2. Règle 2 : StateTransition2B

La transformation est illustrée par la figure 4.7. Elle permet de transformer des états d'un diagramme d'états-transitions vers leur équivalent en B. Les états d'un diagramme d'états transitions pour une classe Classe dans le LHS, sont transformés formellement en B par la définition en OCL dans la partie RHS (*SETS*, *VARIABLES* et *INVARIANTS*)

Elle décrit l'état courant de chaque objet de la classe.

Si les états sont directement liés aux valeurs des variables d'instance (attributs) et que cette relation peut être exprimée par un prédicat, celui-ci doit être rajouté au bloc des invariants *INVARIANTS* de la machine abstraite B.

La transformation des transitions dans le diagramme d'états-transitions vers leur équivalent en B. Une transition est transformée formellement en B par la définition en OCL dans le RHS, d'une opération dont le rôle est de d'effectuer le changement d'état. La figure 4.7 représente la règle de transformation.

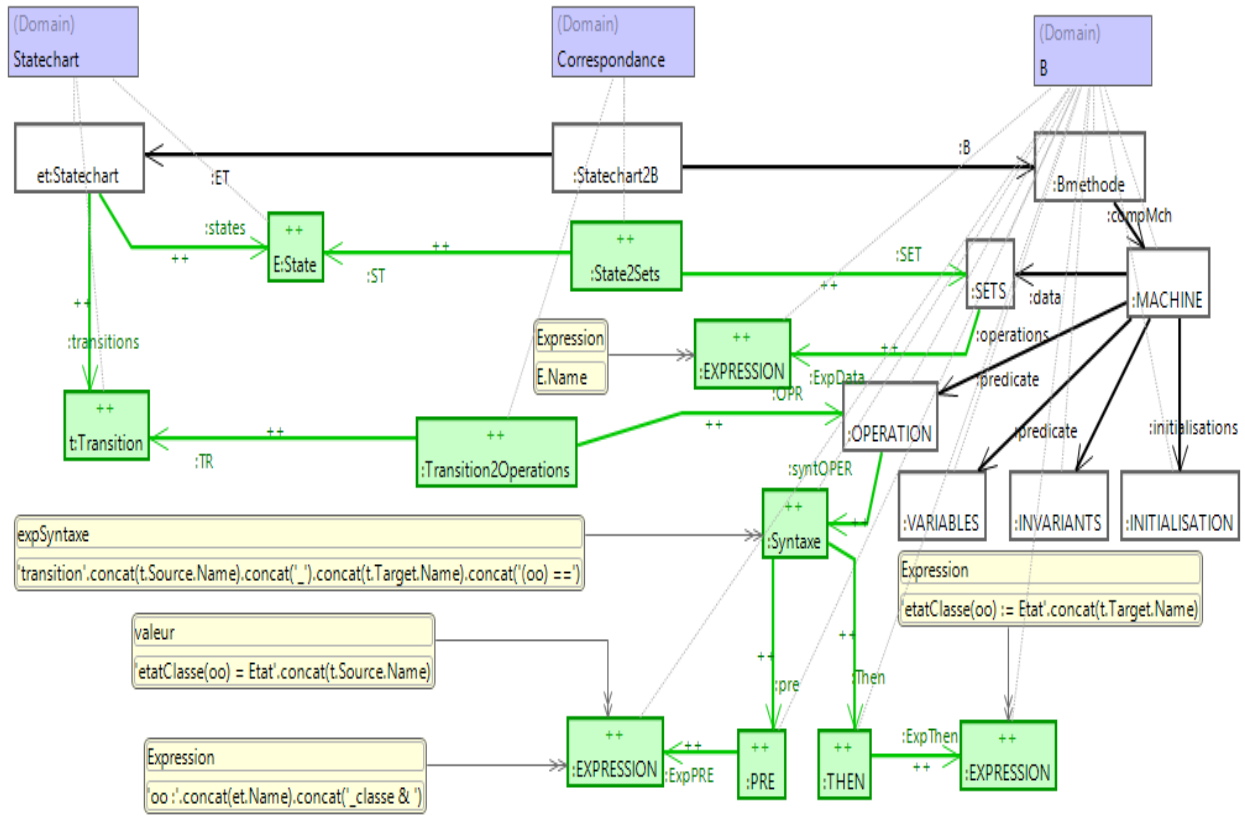


Figure 4.7: Règle pour transformer un état et une transition.

3.4.3. Règle 3 : Action2B

La transformation formelle en B d'une action (Figure 4.8) le corps de la sélection associée à la transition est affinée par l'appel de l'opération formelle décrivant l'action.

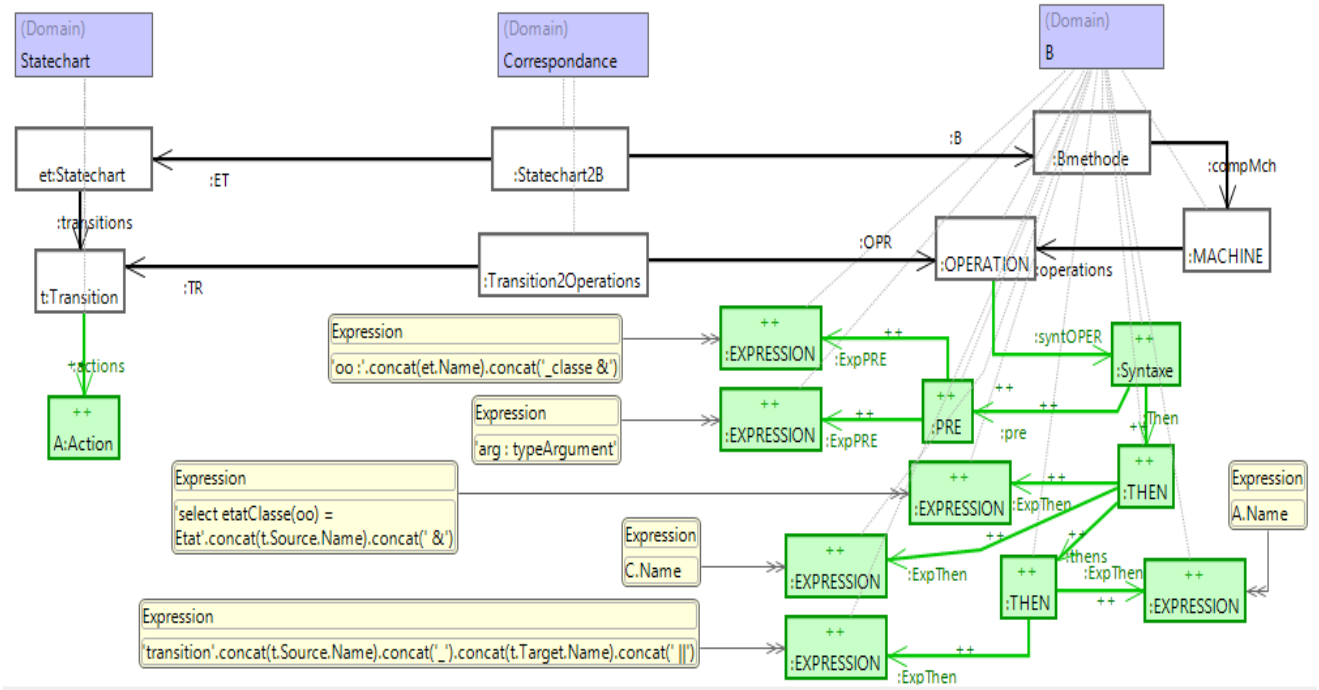


Figure 4.8: Règle pour transformer une action.

3.4.4. Règle 4 : Condition2B

Une condition est transformée formellement en B par la définition en OCL dans le RHS. (Figure 4.9).

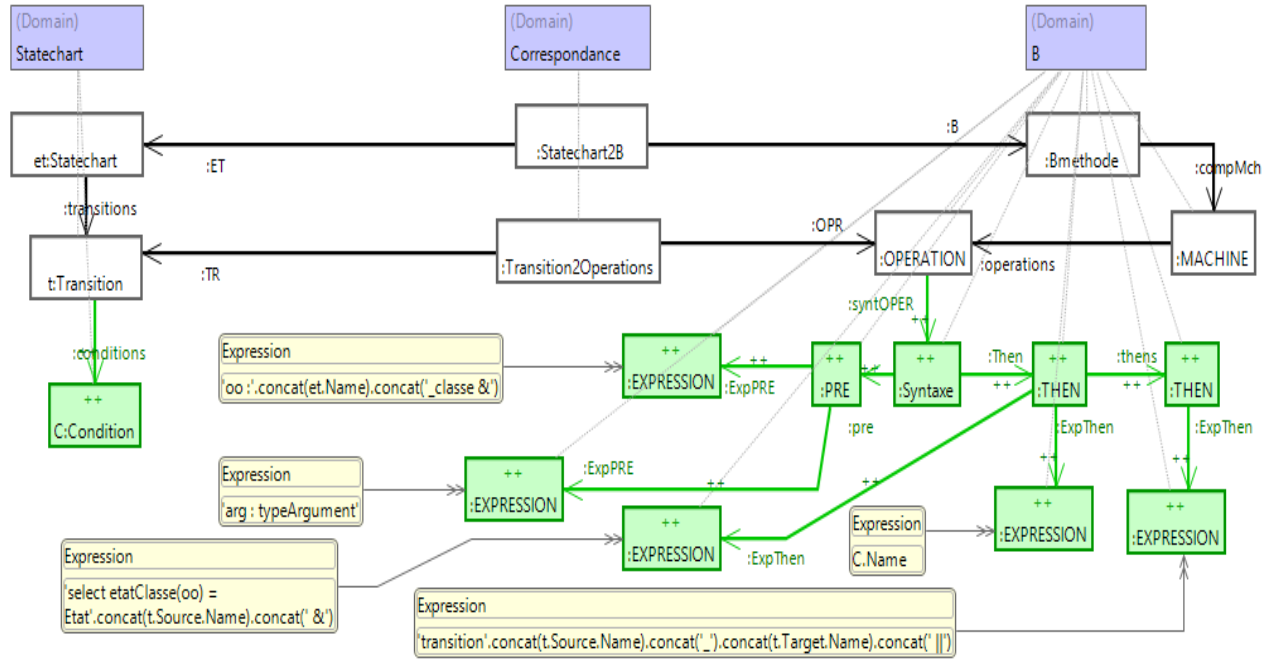


Figure 4.9: Règle pour transformer une condition.

3.5. L'application des règles de transformation

3.5.1. Etude de cas

Nous expliquons notre approche de transformation des statechart vers B à travers cette étude de cas de processus de commandes (process orders). Cet exemple montre l'application des règles de transformation réalisées en TGG-Interpreter.

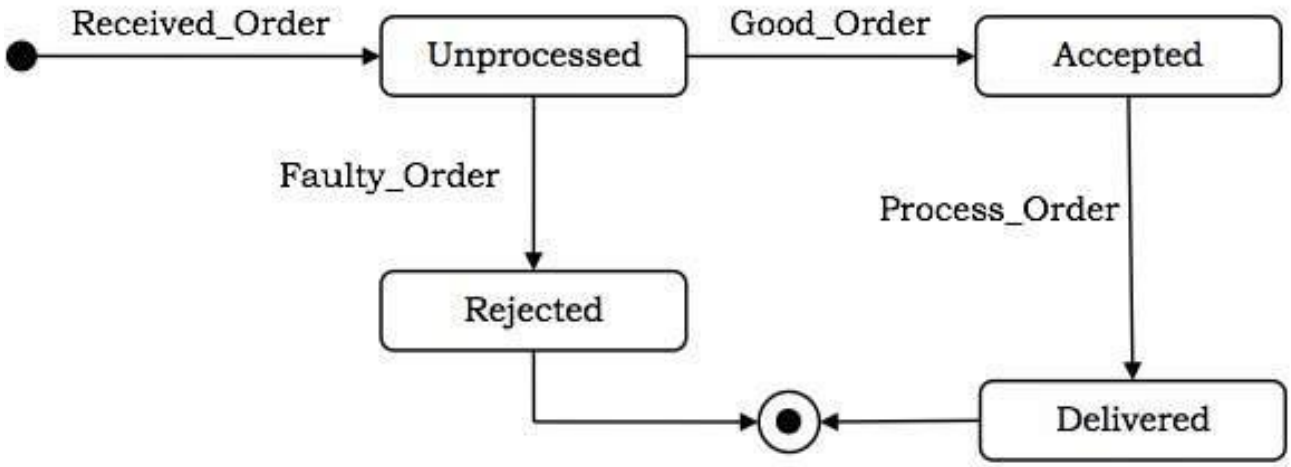


Figure 4.10: Statechart de processus de commandes (process orders) [68].

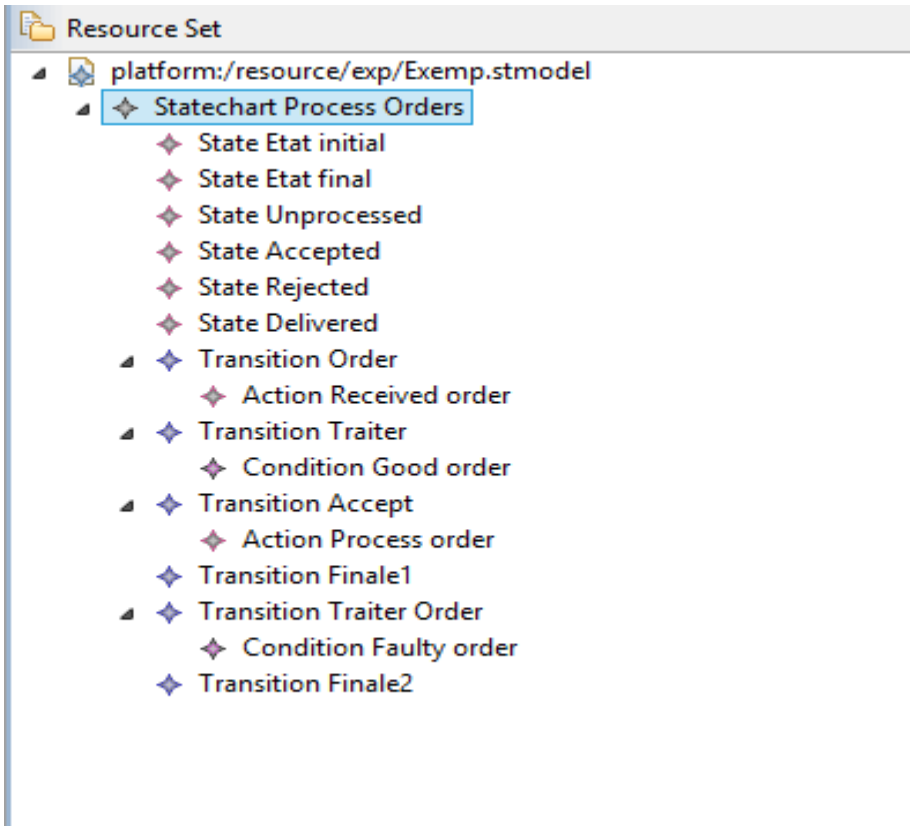


Figure 4.11: L'implémentation de l'exemple.

3.5.2. Vérification des règles

Après l'implémentation de l'exemple, nous avons appliqué les règles de TGG pour les vérifier, Si les règles sont appliquées correctement. Le fichier généré automatiquement avec l'extension .bmethod. La figure 4.12 présenté l'exécution des règles TGG.

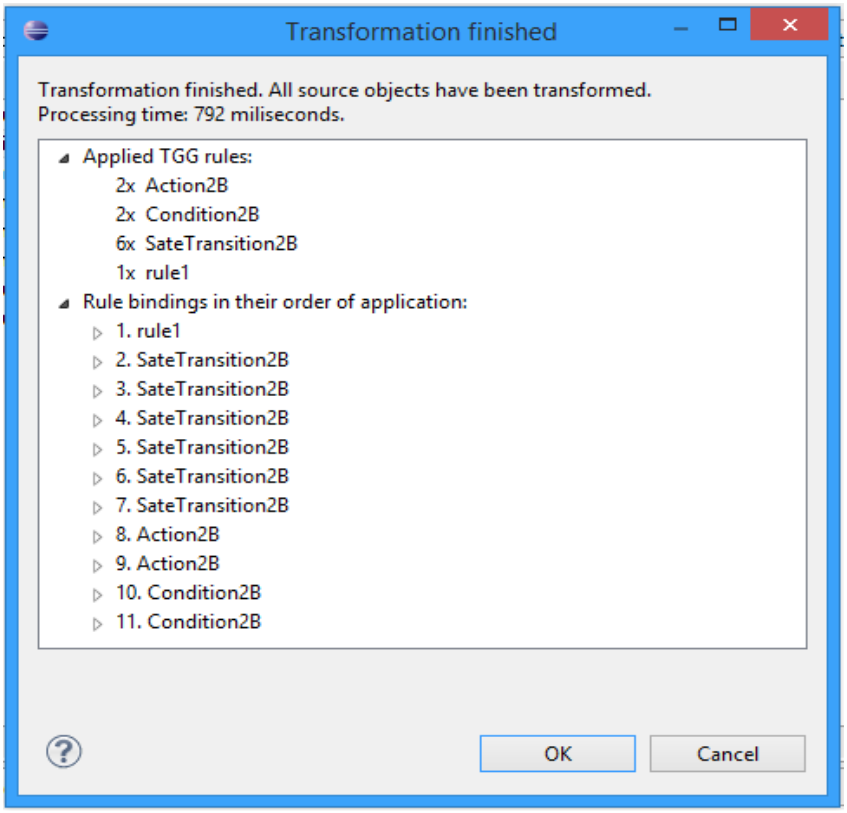


Figure 4.12: Application des règles.

Après avoir exécuté notre moteur de transformation on a un fichier B méthode. La figure 4.13 présenter le modèle cible de la transformation.

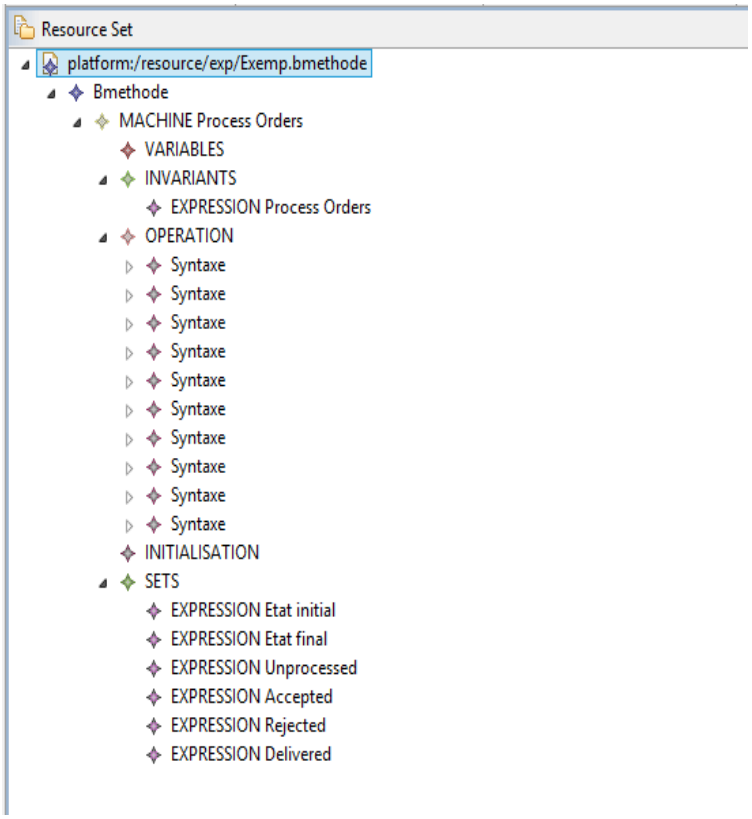


Figure 4.13: Après la transformation.

4. Génération de code B

Dans cette partie notre objectif est généré le code B à partir le diagramme état-transition (Statechart), cette objectif est réalisé par deux outils qui sont : TGG-Interpreter qui nous permet d'implémenter les règles de transformation M2M en se basant sur une grammaire de graphes triples TGGs, et l'outil Xpand qui nous donne la faveur de programmer des générateurs de code efficaces en s'appuyant sur des templates. Cet outil nous offre la possibilité d'implémenter des transformations M2T.

4.1. Générateur Xpand

La figure 4.14 représenté le fichier générateur Xpand. Ce fichier est composé de trois packages.

Le premier package comprend le méta-modèle du langage B sous l'extension « .ecore » afin de générer le code B conformément à ce méta-modèle.

Le deuxième package comprend trois sortes de fichiers : un fichier java, un fichier Xtend, un fichier de code template Xpand.

Le dernier package contient le moteur de workflow constituant une sorte de programme principal.

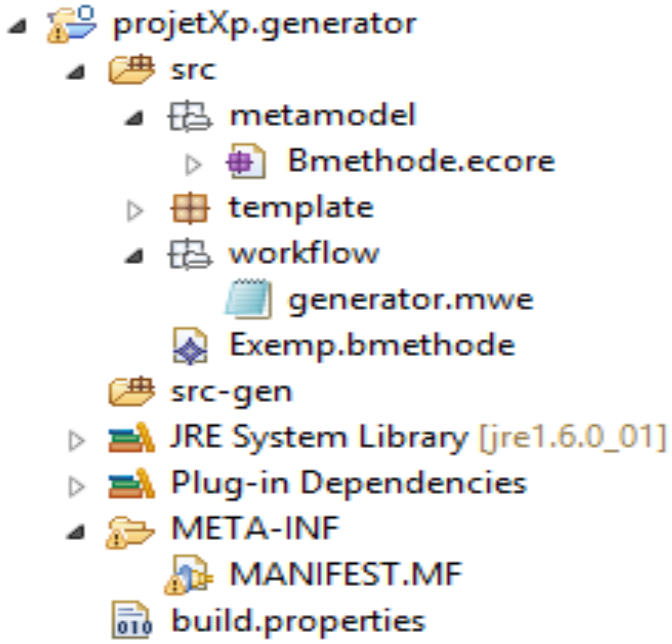


Figure 4.14: Le fichier de générateur Xpand.

Sc.interpreterconfig | Sc.bmethode | Template.xpt | GeneratorExtensions. | generator.mwe | projetXp.generator | MyClasse.java

```
package template;
import java.io.*;
public class MyClasse {
    private static  String filename = "c://test/code.txt";
    private static boolean existed = true;
    public static PrintWriter print1;
    public static int co=0;
    public static int co1=0;
    static String   chaine ="";


        public static void enregistrer(String aString) {
            try {
                 print1 = new PrintWriter( new BufferedWriter ( new FileWriter  (filename,existed)));
                 print1.println();
                 print1.println(aString);


                 print1.close();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();}
            System.out.println(aString);
          }

        public static void enregistrer1(String aString) {
            try {
                 print1 = new PrintWriter( new BufferedWriter ( new FileWriter  (filename,existed)));
                 print1.println("  "+aString);

                 print1.close();
```

Figure 4.15: Code Java pour créer les fichiers B généré.

Sc.stmodel | StatechartDiagram2B.tgg | Template.xpt | *generator.mwe | projetXp.generator

```
<?xml version="1.0"?>
<workflow>
<property name="model" value="projetXp.generator/src/Sc.bmethode"  />
<property name="src-gen" value="src-gen" />

<bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
<platformUri value=".."/>

    <RegisterEcoreFile value="platform:/resource/projetXp.generator/src/metamodel/Bmethode.ecore"/>

</bean>

<bean id="mm_emf" class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>


<component class="org.eclipse.emf.mwe.utils.Reader">
<uri value="platform:/resource/${model}" />
<modelSlot value="model" />
</component>

<component class="org.eclipse.xpand2.Generator">
<metaModel idRef="mm_emf"/>
<expand
value="template::Template::main FOR model" />
<outlet path="${src-gen}" >
<postprocessor class="org.eclipse.xpand2.output.JavaBeautifier" />
</outlet>
</component>
</workflow>
```

Figure 4.16: Configuration du workflow en XMI

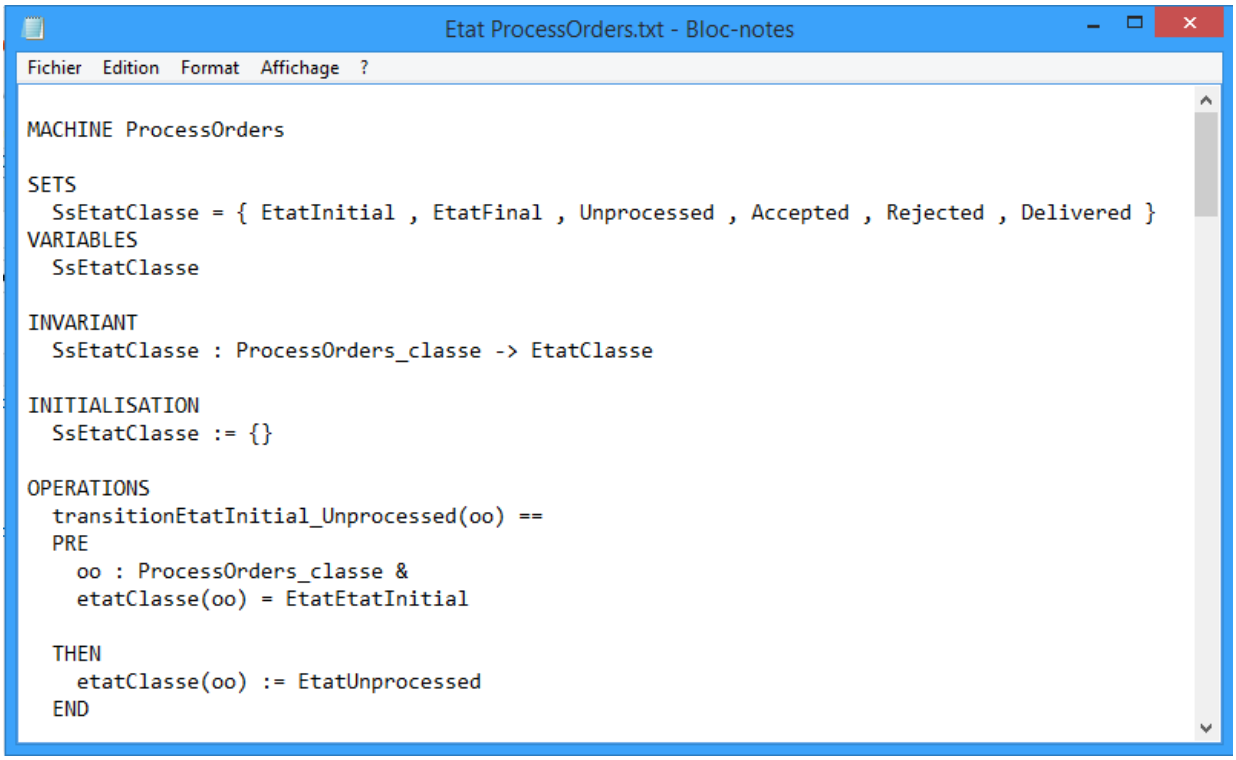


Figure 4.17: Fichier texte généré

5. Conclusion

Dans ce chapitre, nous avons présenté la transformation des modèles (statechart vers B), basée sur l'utilisation de grammaires de graphes triples TGG, après la transformation M2M on a utilisé la transformation M2T pour la génération de code B.

CONCLUSION GENERALE

CONCLUSION GENERALE

Dans ce mémoire, nous avons parlé et représenté le diagramme état-transition (statechart diagram) dans le chapitre 1.

Les diagrammes d'états transitions font partie des diagrammes du langage UML, qui est un langage semi-formel caractérisé par une syntaxe riche et une sémantique ambigüe et non précise, pour surmonter ce problème nous présentons les méthodes formelles et donner la définition de la méthode formelle B dans le chapitre 2, La méthode B couvre toutes les phases amont du cycle de vie pour le développement d'un logiciel à savoir la spécification, la conception et l'implémentation dont toute phase étant validée par des preuves. Le grand avantage, est que celle-ci est soutenue par des outils performants employés dans l'industrie.

Les définitions des concepts : modèle, méta-modèle, langage de modélisation, méta-méta-modèle, transformations de modèles proposées par MDA et leur taxonomie sont explorés dans le chapitre 3. L'outil de transformation TGG a fait l'objet d'une description détaillée, ce mémoire a pour l'objectif de transformer les diagrammes d'états-transitions vers B en utilisant TGG. Ce processus est décrit au chapitre 4.

Perspectives

Comme perspectives, nous envisageons à l'avenir la réalisation des travaux suivants :

-Notre approche actuelle permet le passage d'un modèle Statechart vers une spécification formelle B. pour le but de réaliser les modifications nécessaires dans le modèle statechart de départ de manière automatique en raison de la réversibilité des règles-TGG déjà implémentées. Donc, Le passage d'une spécification formelle B vers des modèles statechart doit être étudié.

-Nous avons pris en compte pour l'application de notre approche proposée des diagrammes d'états-transitions. De par leur complémentarité, les diagrammes de communications, de séquences et d'activités . . . etc. d'UML doivent être ajoutés.

-Nous avons utilisé la méthode formelle B, comme perspective on peut utiliser une autre méthode formelle comme Maude, Z, ……

RÉFÉRENCES BIBLIOGRAPHIQUES

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] P. Kruchten. Architectural Blueprints - The "4+1" View Model of Software Architecture, IEEE Software, vol. 12 (6), pp. 42-50, 1995.

[2] http://tadic.eb2a.com/chapitre%203.pdf, consulté le 01/04/2017.

[3] Mecheri Nacera, Une approche hybride pour transformer les modèles, Mémoire de Magister, Université d'Oran, 2015.

[4] M. Boudia, Transformation des diagrammes d'états-transitions vers Maude, Mémoire de Magistère, Université de M'sila, 2011.

[5] D. Harel, Statecharts : A Visual Fomalism for Complex Systems. Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[6] UML2. Disponible à : http://laurent-audibert.developpez.com/Cours-UML, consulté le : 12/02/2017.

[7] E. Mayer .Développements formels par objets : utilisation conjointe de B et UML .Thèse de doctorat, université Nancy 2, Loria, mars2001.

[8] C. Larman, UML 2 et les design patterns analyse et conception et développement itératif, Pearson Education, 3e édition 2006.

[9] P.A Muller, N. Geartner. Modélisation objet avec UML, Eyrolles, 2éme édition 2000, Deuxième tirage 2001.

[10] Rumbaugh, I. Jacobson, G. Booch, UML 2.0 GUIDE DE REFERENCE, Campus Press, 2005.

[11] H.E. Erikson, M. Penker, B. Lyons, D. Fado, UML 2 ToolKit, Wiley publishing, USA, 2004.

[12] J. Gabay, YD .Gabay. UML 2 analyse et conception mise en œuvre guidée avec études de cas, DUNOD, 2008.

[13] Patrick Grässle, Henriette Baumann, Philippe Baumann. UML2 in Action: A Project-Based Tutorial, Copyright 2005 Packet Publishing, 2005**.**

[14] M. SAMEK, Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems, Elsevier, 2009.

[15] P. Kimmel, UML Demystified, McGraw-Hill Osborne Media, October 2005.

[16] Ninh Thuan Truong. Utilisation de B pour la vérification de spécifications UML et le développement formel orienté objet. Génie logiciel [cs.SE]. Université Nancy II, 2006. Français.

[17] J. M. Wing, A Specifier's Introduction to Formal Methods, Computer, vol. 23(9):8- 23, 1990.

RÉFÉRENCES BIBLIOGRAPHIQUES

[18] J.R. Abrial. The B-Book : Assigning Programs to Meanings. Cambridge University Press, 1996.

[19] J. M. Spivey. The Z Notation : A Reference Manual. Prentice Hail International Series in Computer Science, 2nd edition, 1992. ISBN 013-978529-9.

[20] J.R. Abrial. B : 2000 et plus. Ecole jeunes chercheurs en programmation, Ecole Normale Supérieure de Lyon - 46, Allée d'Italie 69364 LYON Cedex 07, March 2000.

[21] Seïdali Rehab, Une approche de transformation des diagrammes UML vers les spécifications B, Thèse de Doctorat, Université Constantine 2(ABDELHAMID MEHRI), 2015.

[22] D. Gries. The Science of Programming. Springer Verlag. Berlin, 1981.

[23] E. W. D. Dijkstra and A Discipline of Programming. Prentice-Hall, Englewood Cliffs, 1976.

[24] J.R. Abrial. Extending B without Changing it (for Developing Distributed Systems). H. Habrias, editeur, Butting Into Practice Methods and Tools For Information System Design 1st Conference on the B Method, Nantes (F), November, 1996.

[25] N.Shankar, S.Owre, and J. M. Rushby. PVS Tutorial. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, Formal Methods Europe '93, Odense, Danmark, April 1993.

[26] J.R. Abrial and L.Mussat. Introducing Dynamic Constraints in B. D. Bert, éditeur, B'98 : Recent Advances in the Development and Use of the B Method - 2nd International B Conference, numéro 1393 in Lecture Notes in Computer Science, Montpellier (F), Springer-Verlag, April 1998.

[27] J.Julliand and F. Bellegarde. Extension des spécifications b pour décrire des propriétés dynamiques de systèmes réactifs. AFADL'98 : Actes des journées AFADL, Futuroscope - Poitiers (F), Octobre 1998.

[28] E. Seidwitz, What Models Mean, IEEE Software (September 2003).

[29] J. Bézivin, On the Unification Power of Models, on Software and SystemsModeling. 4 (2005), no. 2, 171_188.

[30] G. Vega. Développement d'Applications à Grande Echelle par Composition de Méta-Modèles, PhD. thesis, Université Joseph Fourier, Décembre 2005.

[31] Jean. Bézivin, S.Gérard, P.-A. Muller, and L. Rioux, MDA Components: Challenges and Opportunities, Proceedings of Metamodelling for MDA (York, England), 2003.

[32] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics", Computer, 37(10):64–72, 2004. Disponible sur: http://rumpe.net/~rumpe/publications20042008/HR_ModSemantics_IEEEComp_04.pdf, consulté le : 01/03/2017.

RÉFÉRENCES BIBLIOGRAPHIQUES

[33] J.-M. Favre, Foundations of Meta-Pyramids : Languages vs. Metamodels – Episode II : Story of Thotus the Baboon1, Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development (Dagsthul, Germany), 2004.

[34] Calvez (J. P.). Spécification et conception des systèmes - Une méthodologie : MCSE. Masson, 1990.

[35] Morel Gérard. Contribution à l'automatisation et à l'ingénierie des Systèmes Intégrés de Production. Thèse, Université de Nancy I, 1992.

[36] Sriplakichp Prawee., Techniques des transformations de modèles basées sur la métamodélisation, Mémoire de DEA Systèmes Informatiques Répartis, Université Pierre et Marie Curie, septembre 2003.

[37] Object Management OMG. Unified modelling language specification version 2.0 Infrastructure. Technical Report ptc/03-09-15, OMG, 2003. Disponible sur : https://vu.fernunihagen.de/lvuweb/lvu/file/FeU/Informatik/2005WS/01794/oeffentlich/UML/ML2-Infrastructure.pdf, consulté le 01/03/2017.

[38] Object Management Group OMG, Meta Object Facility (MOF), Technical Report OMG Document: ad/97-08-14, Sept 1997.

[39] k. Czarnecki, S. Helsen .feature –based survey of model transformation approaches, IBM SYSTEMS JOURNAL, VOL45, NO3, 2006.

[40] Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained: The Model-Driven Architecture: Practice and Promise. Addison Wesley (2003).

[41] Frédéric Jouault, Contribution à l'étude des langages de transformation de modèles, THÈSE DE DOCTORAT, Le 26 septembre 2006, l'UFR Sciences & Techniques, Université de Nantes.

[42] Schürr Andy. Specification of graph translators with triple graph grammars. In WG 94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science, pages 151–163, London, UK, 1995. Springer-Verlag, ISBN 3-540-59071-4.

[43] Stéphane Bonnet .Une démarche dirigée par les modèles pour la personnalisation des applications embarquées dans les cartes à puce. PhD. thesis, Université de LILLE 1, Mai 2005.

[44] Hubert Kadima. Conception orienté objet guidée par les modèles. Dunod, 2005.

[45] k. Czarnecki, S. Helsen. Classification of model transformation approaches. 2003. OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, University of Waterloo, Canada.

[46] J., E. Guerra. Formal Support for Model Driven Development with Graph Transformation Techniques. Disponible sur: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-

WS/Vol-157/paper04.pdf Ou http://users.dsic.upv.es/workshops/dsdm05/files/DSDM2005Proceedings.pdf#page=38, consulté le 05/03/2017.

[47] Selma Azaiez .Approche dirigée par les modèles pour le développement de systèmes multi-agents. Thèse de doctorat. L'université de SAVOIE, décembre 2007.

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

[49] Combemale benoît. Approche de méta-modélisation pour la simulation et la vérification de modèle. Thèse soutenu à l'ENSEEIHT, juillet 2008. Disponible sur: http://hal.archivesouvertes.fr/docs/00/32/18/63/PDF/phd-combemale-finalversion.pdf, consulté le 05/03/2017.

[50] Andy Schürr, Specification of graph translators with triple graph grammars, In Graph-Theoretic Concepts in computer science, page 151-163, Springer, 1995.

[51] Francisco de la Parra, Application of graph grammars to model transformations, Technical Report : 2013-604, 2013.

[52] Ximeng Sun. A Model-Driven Approach to Scenario-Based Requirements Engineering, School of Computer Science McGill University, Montréal, Canada, 2007.

[53] Hans Vangheluwe, Juan de Lara. Computer Automated Multi-paradigm Modelling for Analysis and of Traffic Networks, Proceedings of the 2004 Winter Simulation Conference, School of Computer Science, McGill University Montréal, Québec H3A.

[54] Raida El Mansouri, Elhillali Kerkouche, and Allaoua Chaoui, A Graphical Environment for Petri Nets INA Tool Based on Meta-Modelling and Graph Grammars. World Academy of Science, Engineering and Technology 44 2008.

[55] Christian Attiogbé, Pascal Poizat, Gwen Salaün, Intégration de données formelles dans les diagrammes d'états d'UML, Université de Nantes, Université d'Évry Val d'Essonne, Disponible sur: https://pages.lip6.fr/Pascal.Poizat/documents/publications/APS03a.pdf, consulté le 11/03/2017.

[56] https://eclipse.org/ , consulté le 15/06/2017.

[57] http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr2, consulté le 15/06/2017.

[58] http://www.eclipse.org/ecoretools/, consulté le 15/06/2017.

[59]http://www-old.cs.uni-paderborn.de/en/research-group/softwareengineering/research/projects/tgg-interpreter/installation.html , consulté le 15/06/2017.

RÉFÉRENCES BIBLIOGRAPHIQUES

[60] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter, Automatic conformance testing of optimized triple graph grammar implementations. In applications of Graph Transformations with Industrial Relevance, Pages 238-253. Springer, 2012.

[61] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A compariason of incremental triple graph grammar tools, Electronic Communication of the EASST, 67, 2014.

[62] https://eclipse.org/modeling/m2t/downloads/?project=xpand, consulté le 15/06/2017.

[63]http://jmini.developpez.com/eclipse_emf/articles/premier_exemple_xpand/, consulté le 03/05/2017.

[64] Joel Greenyer, A study of model transformation technologies: Reconciling tggs with qvt, master's thesis, university of Paderborn, 2006.

[65] Joel Greenyer and Jan Rieke, Applying advanced tgg concepts for complex transformation of sequence diagram specification to timed game automata. In application of graph transformation with industrial relevance, pages 222-237, spring, 2012.

[66] Ekkart Kindler and Robert Wagner, triple graph grammars: concepts, extensions, implementations, and application scenarios, university of Paderborn, 2007.

[67] Ekkart Kindler, Vladimir Rubin, and Robert Wagner, Component tools: integrating petri nets with other formal methods. In Petri Nets and Other Models of Concurrency-ICATPN 2006, pages 37-56, Spring, 2006.

[68]https://www.tutorialspoint.com/object_oriented_analysis_design/pdf/ooad_uml_behavioural_diagrams.pdf, consulté le 16/05/2017.

ملخص

منحنيات الحالات-التحولات يحدد حالات مختلفة للكائن خلال حياته وهذه الحالات تتغير بالأحداث. ومع ذلك، فإن الكائنات الموجهة نحو الأهداف تفتقر إلى إضفاء الصبغة الرسمية على قاعدة موثوق بها لتحليل الطلبات والتحقق منها. للتغلب على هذه المشكلة، نقترح حلا لتحويل الرسم البياني منحنيات الحالات-التحولات نحو أساليب رسمية، وخاصة طريقة B.وهدفنا هو تحويل منحنيات الحالات-التحولات إلى B. يتم تحقيق هذا التحويل من قبل قواعد TGG.

الكلمات المفتاحية:

منحنيات الحالات-التحولات، تحويل النماذج، الطرق الرسمية، طريقة B، الثلاثي الرسم البياني النحوي TGG.

ABSTRACT

The statechart diagram defines different states of an object during its lifetime and these states are changed by events .Nevertheless, the object-oriented notations lack of formalization and miss of reliable bases for the analysis and verification of the applications. To overcome this problem, we suggest a solution to transfer statechart towards formal methods, in particular the B method. Our goal is to transform statechart to B. This automation is achieved by the TGG rules.

Keywords:

Statechart diagrams, model transformation, formal methods, B method, triple graph grammar TGG.

RESUME

Le diagramme état-transition définit différents états d'un objet au cours de sa durée de vie et ces états sont modifiés par des événements. Néanmoins, les notations orientées objet ne sont pas formalisées et manquent de bases fiables pour l'analyse et la vérification des applications. Pour surmonter à ce problème, nous proposons une solution pour transformer le diagramme état-transition vers des méthodes formelles, en particulier la méthode B. Notre objectif est de transformer statechart en B. Cette automatisation est réalisée par les règles de TGG.

Mots-clés:

Diagramme état-transition, transformation des modèles, méthodes formelles, méthode B, grammaires de graphes triples TGG.