République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de
La Recherche Scientifique
Université Djilali Bounaama-Khemis Miliana
Faculté des sciences et de la Technologie

Département de Mathématiques et Informatique

Mémoire de fin d'étude
En vue de l'obtention d'un diplôme de
Master *en Informatique*
Spécialité Génie Logiciel et systèmes distribués

Thème :

Une approche de transformation et de vérification des diagrammes d'états transitions UML avec Isabelle/ HOL

Présenté par :

- Khial Hiba
- Nemas Racheda

Devant le jury composé de :

- Examinateur 1 : Mme.BOUDALI.F
- Examinateur 2 : Mr.HARBOUCHE.O
- Encadreur :Dr. HACHICHI .Hiba

Année Universitaire : 2019/2020

Remerciements

Nous remercions Allah le tout puissant de nous avoir donné la volonté et le courage pour mener à bien ce travail.

Nous tenons à remercier vivement notre promoteur Dr HACHICHI HIBA pour avoir accepté de diriger ce travail et pour ses précieux conseils et ses encouragements. qui s'est toujours montré disponible tout au long de la réalisation de ce mémoire, ainsi pour l'inspiration, l'aide et le temps qu'il est bien voulu nos consacrer, et sans qui ce mémoire n'aurait jamais vu le jour.

Nos vifs remerciements s'adressent à tous les membres de jury qui nous ont fait l'honneur d'examiner ce travail.

A toutes les personnes qui ont contribué de près ou de loin, d'une manière directe ou indirecte à l'élaboration de ce travail de fin d'études.

Dédicace

Je dédie ce travail avec grand amour à :
Mes chers parents mon père Mohamed et ma mère Hassina pour leur patience, leur amour, leur soutien et leur encouragements au cours des années dès mes études.
Mon frère Imad et mes sœurs Abir et Alaa et tous les membres de ma famille.
Sans oublier mes amis Kari Ines, Imamame Zahera et Nemas Racheda. Et tous les professeurs.
Avec mes souhaites de bonheurs de santé et de succès.

Hiba

Dédicace

A mes très chers parents

Pour toutes les peines en durées toues les privations et sacrifice qu'ils ont déployé pour mon éducations Je leur doit mon plus grand respect, car je ne pourrais jamais oublier la tendresse et l'amour dévoués par lesquels ils m'ont toujours entouré depuis mon enfance

J'espère

Avoir répondu aux espoirs qui vous avez fondis en moi

Je veux rends hommage de ma reconnaissance éternelle et de mon amour infini

Que dieu, le tout puissant vous préserver et vous procure santé, longue vie, et bonheur.

A mes chères sœurs et mes très chers frères

Zineb, Nawel, Youcef et Abd El Azzize

Pour tant de confiance d'amour de patience et d'abnégation.

Que dieu vous gardé au pris de nous, et vous réalise tous vos rêves et vos veux.

A tous les membres de la famille

Pour leur assistance durant notre vie estudiantine

Je vous offre ce travaille avec tous mes vœux de bonheur, de santé et de réussite.

A ma chère amie et ma binome Khial Hiba

A mes amies Feryal ,ahlame et radhia

A tous mes professeures

Qui m'ont enseigne depuis ma première scolarité

Plus particulièrement à celui qui nous a encadrer et diriger par ses précieux conseils.

Celui qui nous a toujours encouragés

Mme ,Hachichi Hiba

Merci pour votre aide multiforme.

Puisse dieu, vous protège et vous procure santé, bonheur, longue vie et réussite.

A tous mes amies et collègues de promotions et d'étude

Racheda

ملخص :

وقد أدى التقدم المحرز في هندسة البرمجيات إلى تطوير نظم برمجيات متزايدة التعقيد. ويطرح هذا التعقيد مشاكل تتعلق بالتكلفة ووقت التطوير ونقص الدلالات اللفظية.

ومن بين الحلول التي يمكن أن تعالج هذه المشاكل هندسة IDM (هندسة النماذج) التي هي تقنية جديدة تستند إلى النماذج والتحولات التي تتم في جميع مراحل عملية تطوير البرمجيات. ويجب التحقق من صحة هذا التحول، كما يجب التحقق من الحاجة إلى دمج الطرق الرسمية لإضافة مواصفات مفصلة.

والعمل المقدم في هذا المشروع نهج جديد تتألف أولاً من جعل تحويل النماذج أكثر دقة ويتحول تحويل مخططات الحالة إلى شبكات بتري متأخرة زمنًا وثانياً للتحقق من هذا التحول.

الكلمات المفتاحية: هندسة البرامج التي تعتمد على النموذج، تحويل النموذج، التحقق، الطرق الرسمية، مساعد اثبات Isabelle / HOL

Résumé :

Les progrès dans le domaine du génie logiciel ont permis le développement de systèmes de logiciels de plus en plus complexes. Cette complexité pose des problèmes de coût, de temps de développement et un manque de sémantique.

L'une des solutions permettant de prendre en charge ces problèmes est l'IDM (l'ingénierie Dirigée par le Modèle) qui est une nouvelle technique se basant sur les modèles et leurs transformations tout au long du processus de développement des logiciels. Cette transformation doit être validée et on doit également vérifier la nécessité d'intégration des méthodes formelles pour ajouter des spécifications détaillées.

Le travail présenté dans ce projet est une nouvelle approche qui consiste premièrement à faire la transformation de modèles plus précisément la transformation des digrammes d'états transitions vers les réseaux pétri temporellement temporisés et deuxièment la vérification de cette transformation.

Mots clés : Ingénierie dirigée par les modèles, transformation de modèle, vérification, méthodes formelles, assistant de preuve Isabelle/HOL.

Abstract :

Advances in software engineering have led to the development of increasingly complex software systems. This complexity poses problems of cost, development time and a lack of semantics.

One of the solutions to address these problems is the IDM (Model-Led Engineering) which is a new technique based on models and their transformations throughout the software development process. This transformation must be validated and the need to integrate formal methods to add detailed specifications must be verified.

The work presented in this project is a new approach consisting first of making the transformation of models more precisely the transformation of state diagrams transitions to time-delayed petri networks and second of verifying this transformation.

Keywords : Model-drivenengineering, modeltransformation,verification, formalmetho s, Isabelle/HOLproofassistant.

Table des matières

Liste des figures

Liste des tableaux

Introduction générale

Depuis l'apparence du génie logiciel le développement des systèmes devient plus important dans différents domaines d'application et évalue d'une manière rapide et avec le temps les systèmes logiciels sont devenus complexes d'un côté par le grand nombre des composants et d'un autre coté par la diversité de ses composants ceci implique des problèmes tel que l'optimisation des couts et le temps de développement de ses systèmes ainsi que leur analyse et leur vérification.

L'IDM (Ingénierie Dirigée par des Modèles) se base sur deux aspects principaux : la métamodélisation et la transformation de modèles. Le premier aspect concerne la définition et la manipulation des métamodèles et des modèles pour spécifier les systèmes informatiques à un niveau très abstrait, et le deuxième aspect porte sur la transformation automatique ou semi- automatique des modèles pendant le développement afin de produire les applications informatiques, autrement dit l'IDM essaye de prendre en charge les problèmes cité auparavant et qui sont liées au contexte de développement des systèmes.

L'UML comme un langage à caractère plutôt visuel, il souffre d'un manque de sémantique formelle et aussi un manque des outils d'analyse et de vérification, en effet les notations semi-formelles et visuelles d'UML peuvent provoquer des inconsistances au niveau des systèmes développés, d'autre part les méthodes formelles comme les réseaux pétri et précisément les réseaux pétri temporellement temporisés RPTT considérés comme des outils graphiques formelles servent à mieux modéliser les systèmes complexes.

Pour englober toutes les informations précédentes nous avons proposés une nouvelle approche qui consiste non seulement à transformer les diagrammes d'états transitions UML vers les réseaux pétri temporellement temporisés RPTT en se basant sur la transformation de modèle mais aussi la vérification de la transformation elle-même en utilisant l'assistant de preuve Isabelle/HOL.

Notre mémoire est organisé selon le plan suivant :

Le premier chapitre présente le domaine de l'IDM, ces concepts de base avec ses différentes variantes en mettant l'accent sur l'approche MDA (Architecture Dirigée par les Modèles.).

Le deuxième chapitre présente l'UML, un bref historique, une simple définition ainsi que ses

différentes diagrammes en décrivant en détail le diagramme d'états transitions et ses principaux éléments vue qu'il représente le modèle de base dans notre étude.

Le troisième chapitre est une vue globale sur les méthodes formelles et leur classification, leur intégration a l'IDM et la fin de ce chapitre est consacré aux réseaux pétri temporellement temporisés RPTT comme une extension des réseaux de pétri.

Le quatrième chapitre décrit en détail notre approche de transformation des diagrammes d'états transitions UML vers les réseaux pétri temporellement temporisés RPTT et la vérification de l'exactitude de cette transformation avec le Théorème de preuve Isabelle/HOL et nous terminons ce mémoire par une conclusion générale.

Chapitre I

Ingénierie dirigée par les Modèles

I.1 Introduction :

L'ingénierie dirigée par les modèles (IDM, ou MDE : Model Driven Engineering) est un sujet de recherche important dans le domaine du génie logiciel. L'IDM est une approche de développement mettant à disposition un ensemble d'outils et de langages pour la manipulation des modèles. Ces derniers sont une manière intuitive et naturelle permettant de représenter l'état et le comportement d'un système, quel que soit son degré de complexité et à chaque étape de son cycle de vie. Les techniques d'ingénierie dirigée par les modèles appliquent une transition logique entre des modèles sémantiquement riches mais simplifiés et d'autres modèles incluant plus de détails techniques et de conception. Cette approche aurait été appliquée dans les différents domaines de génie logiciel. Elle a apporté plusieurs améliorations significatives dans le développement des systèmes logiciels complexes en fournissant des moyens permettant de passer d'un niveau d'abstraction à un autre ou d'un espace de modélisation à un autre.

Cependant, la gestion des modèles peut s'avérer lourde et coûteuse. Pour pouvoir mieux répondre aux attentes des utilisateurs, il est nécessaire de fournir des outils flexibles et fiables pour la gestion automatique des modèles ainsi que des langages dédiés pour leurs transformations.

Dans ce chapitre, nous proposons un tour d'horizon sur les aspects fondamentaux de l'IDM en mettant l'accent sur la transformation de modèles qui constitue le thème central de cette discipline.

I.2 L'ingénierie système :

Un système est un ensemble d'éléments identifiables, interdépendants, c'est-à-dire liés entre eux par des relations telles que, si l'une d'elles est modifiée, les autres le sont aussi et par conséquent tout l'ensemble du système est modifié et transformé. C'est également un ensemble borné dont on définit les limites en fonction des objectifs (propriétés, buts, projets, finalités) que l'on souhaite privilégier [3].

L'ingénierie système est une approche méthodologique pluridisciplinaire qui intègre l'ensemble des activités centrées autour du cycle de vie d'un système complexes, depuis sa définition jusqu'à son retrait de service en passant par sa conception, sa validation ou sa maintenance. L'objectif de ce processus est de contrôler la conception de systèmes dont la complexité ne permet pas le pilotage simple [1].

I.2.1 Notions de qualité pour un système :

En ingénierie système, divers travaux ont défini la qualité du logiciel en termes de facteurs ou de métriques qualitatives et quantitatives, qui dépendent du domaine de son application et des outils utilisés. Parmi ces derniers nous pouvons citer :

- ***La validité :*** *un système doit assurer son fonctionnement exactement comme il a été défini par le cahier des charges et les spécifications.*
- ***La fiabilité ou la robustesse :*** *un système doit pouvoir fonctionner aussi dans des conditions anormales.*
- ***L'extensibilité (maintenance) :*** *un système doit se laisser maintenir, c'est-à-dire supporter des éventuelles modifications ou une extension vers des fonctions qui lui seront demandées.*
- ***La réutilisabilité :*** *un système doit être apte à être réutilisé, partiellement ou dans son intégralité, dans d'autres systèmes.*
- ***La compatibilité*** *un système doit être interopérable avec d'autres systèmes.*
- ***L'efficacité :*** *un système doit pouvoir utiliser les ressources matérielles de manière optimale.*
- ***La portabilité :*** *un système doit pouvoir être facilement transféré vers différents environnements matériels et logiciels.*
- ***La vérifiabilité :*** *un système doit faciliter la préparation des procédures de test en son sein.*
- ***L'intégrité :*** *un système doit protéger son code et ses données et doit pouvoir gérer les autorisations d'accès.*
- ***La facilité d'emploi :*** *un système doit offrir une facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.*

Ces facteurs sont parfois contradictoires. Le choix des compromis doit s'effectuer en fonction du contexte [6].

I.3 Ingénierie Dirigée par les Modèles IDM :

Suite à l'approche objet et de son principe du "tout est objet", l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles et le principe du "tout est modèle" [2].

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

L'ingénierie dirigée par les modèles (IDM) est donc une approche du génie logiciel sur laquelle le modèle est considéré comme une première présentation du système à modéliser, et qui vise à développer, maintenir et faire évoluer le logiciel en réalisant des transformations de ce modèle. Au sens large, le paradigme de l'IDM propose d'unifier tous les aspects du processus du cycle de vie en utilisant les notions de modèle et de transformation [3].

I.3.1 Une approche autour des modèles :

L'IDM est un domaine de recherche en pleine expansion aussi bien dans le monde académique que dans le monde industriel qui considère les modèles comme les éléments de base tout au long du processus de développement. L'IDM raisonne entièrement à un haut niveau d'abstraction et non plus à celui des langages de programmation classiques. Une application sera alors générée en tout ou en partie, automatiquement ou semi automatiquement à partir de modèles, en utilisant notamment des transformations successives de ces modèles. Il s'agit donc d'une forme d'ingénierie générative, le code source de l'application n'est plus considéré comme l'élément central d'un logiciel, mais comme un élément dérivé d'éléments de modélisation. Le processus de conception est vu comme un ensemble de transformation de modèles. Cette approche adopte le principe : "tout est modèle" en analogie avec "tout est objet" dans la vision de l'approche orienté objet [1].

I.3.2 Une méthode orientée modèles :

La séparation du modèle de l'architecture est l'un des premiers objectifs de l'IDM qui place le modèle au centre du développement et assure que le développement d'un système se réalise par un processus fait de plusieurs transformations successives d'un modèle de base, jusqu'à obtention du code final [11].

I.3.3 Modèles, formalismes de modélisation et métamodèles :

- ***Modèle :***

Un modèle est une abstraction d'un système étudié utilisé pour présenter l'architecture générale d'une application ou sa place dans une organisation, il doit pouvoir être utilisé pour répondre à des questions des systèmes [2].

Il faut noter qu'il reste toutefois difficile de répondre à la question "Qu'est-ce qu'un bon modèle ?". Néanmoins un modèle doit être suffisant et nécessaire pour permettre de répondre à cer-

taines questions du système qu'il représente, exactement de la même façon que le système aurait répondu lui-même. Le modèle doit se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [1].

En effet la définition d'un langage de modélisation a pris la forme d'un modèle, appelé Méta modèle.

- ***Métamodèles et méta-modélisation :***

L'ingénierie dirigée par les modèles préconise l'utilisation d'un mécanisme standard et abstrait pour définir des modèles. Ce mécanisme abstrait est dénoté par le terme méta-modèle.

Dans la littérature, nous trouvons plusieurs définitions de méta-modèle. De manière générale, un méta-modèle est le modèle qui sert à exprimer (modéliser) le langage d'expression d'un modèle, avec ses entités, ses relations et ses contraintes. À son tour, le méta-modèle est aussi spécifié dans un langage de méta-modélisation par le méta-méta-modèle. Arrivé à ce niveau d'abstraction méta-circulaire, le langage est assez puissant pour spécifier sa propre syntaxe abstraite.

Enfin, la méta-modélisation est l'activité de construire des métamodèles. Elle est très utilisée aussi dans le domaine de l'ingénierie des systèmes d'information, et particulièrement dans l'ingénierie des modèles et des méthodes [4].

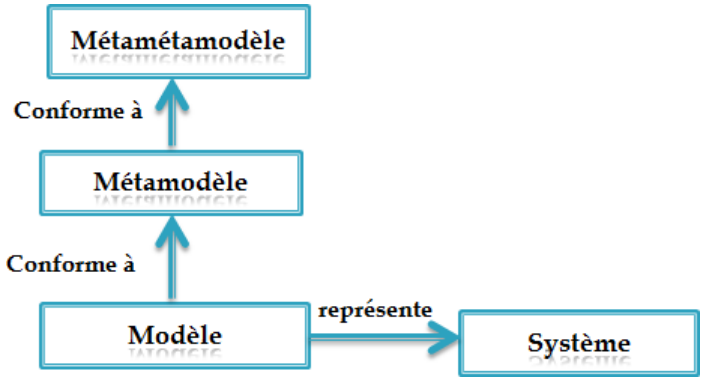


FIGURE I.1 – Notions de base en ingénierie des modèles [3].

La relation entre le système, le modèle et le méta-modèle est représentée dans la figure I.1.

I.3.4 Transformations de modèles :

L'IDM considère les opérations de transformations de modèles comme le moteur de développement, que ce soit pour l'analyse, l'optimisation ou la génération de code.

Cette procédure consiste à générer, à partir d'un ou de plusieurs modèles sources d'un système, un ou plusieurs modèles cibles du même système. Les modèles sont dans tous les cas conformes à

leurs métamodèles respectifs. Lorsque la transformation se fait dans un même formalisme (méta-modèle source = méta-modèle cible) elle est dite endogène. Dans l'autre cas, quand les deux méta-modèles source et cible sont différents, la transformation est dite exogène.

Ces transformations sont gouvernées par un ensemble de règles utilisées par le moteur de transformation. Ce moteur prend en entrée le modèle source, exécute les règles de transformation et génère le modèle cible.

La figure I.2 présente un scénario de transformation de modèle avec ses principaux concepts. Un modèle source en entrée et un modèle cible en sortie. Les deux modèles sont conformes à leurs métamodèles respectifs. Le méta-modèle exprime la syntaxe abstraite de la notation du modèle. Une transformation respecte les métamodèles et est réalisée par l'intermédiaire d'un moteur de transformation [5].

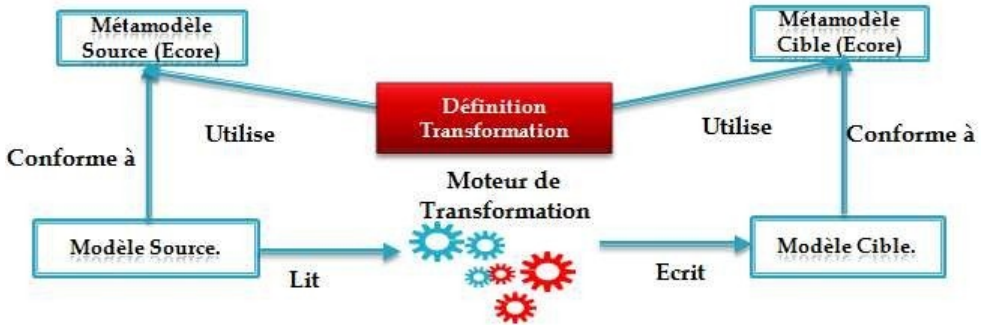


FIGURE I.2 – Concepts de base des transformations de modèles [3].

I.3.5 Types de transformations de modèles :

Un facteur important à prendre en considération dans les transformations de modèles est le niveau d'abstraction. Selon ce facteur, nous pouvons distinguer trois types de transformations [4] :

- ***Transformations verticales :***

Ces transformations se jouent sur les niveaux d'abstractions. Un raffinement se fait lorsque cette transformation descend d'un niveau, mais lorsqu'on élève le niveau d'abstraction, la transformation est dite : une abstraction [4].

- ***Transformations horizontales :***

Ces transformations gardent le même niveau d'abstraction en modifiant les représentations d'informations du modèle source (ajout, modification, suppression ou restructuration) [4].

- ***Transformations obliques :***

Ces transformations sont généralement utilisées par les compilateurs qui optimisent le code source avant la génération du code exécutable. Elles sont le résultat de la combinaison des deux premiers types de transformations. Selon la nature des métamodèles sources et cibles, nous distinguons deux types de Transformations [4].

I.3.6 Classification des approches de transformation de modèles :

Selon les classifications on peut distinguer deux types de transformation de modèles principales : les transformations de type modèle vers code et les transformations de type modèle vers modèles. Pour chaque de ces deux catégories, on distingue des sous catégories la figure 1.3 résume la classification des approches de transformations.

.3.6.1 Transformation Modèle vers Modèle [1] :

- ***Approche par manipulation directe :***

Cette approche est basée sur une représentation interne des modèles source et cible, en plus des API (Application Programming Interface) pour les manipuler.

- ***Approche relationnelle :***

Cette approche utilise les contraintes pour spécifier les relations entre les éléments du modèle source et ceux du modèle cible en utilisant une logique déclarative basée sur des relations mathématiques.

- ***Approche basée sur les transformations de graphes :***

Cette approche convient lorsque les modèles sont représentés par des graphes. Elle exprime les transformations sous une forme déclarative. Les règles de transformation sont définies sur des parties de modèle et non pas sur des éléments basiques. Une opération de filtrage de motifs (Pattern Matching) est ensuite lancée. Le moteur de transformation compare à chaque fois des fragments du modèle source pour trouver des règles applicables. Ce fragment est ensuite remplacé par son équivalent dans la règle appliquée. Quelques exemples d'approches basées sur les transformations de graphes : VIATRA,ATOM3, GreAT, UMLX, BOTL, etc.

- ***Approche basée sur la structure :***

Divisée en deux étapes, la première se charge de la création d'une structure hiérarchique du modèle cible, la seconde ajuste ses attributs et ses références. Le cadre de transformation fourni par Optimal J proposé par Compuware est un exemple d'une approche basée sur la structure.

- ***Approche hybride :***

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

.3.6.2 Transformation Modèle vers Code [7] :

- ***Les approches basées sur le principe du visiteur :***

Consistent à traverser le modèle en lui ajoutant des éléments (mécanismes visiteurs) qui réduisent la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte.

- ***Les approches basées sur le principe des Template :***

Sont actuellement les plus utilisées. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source.

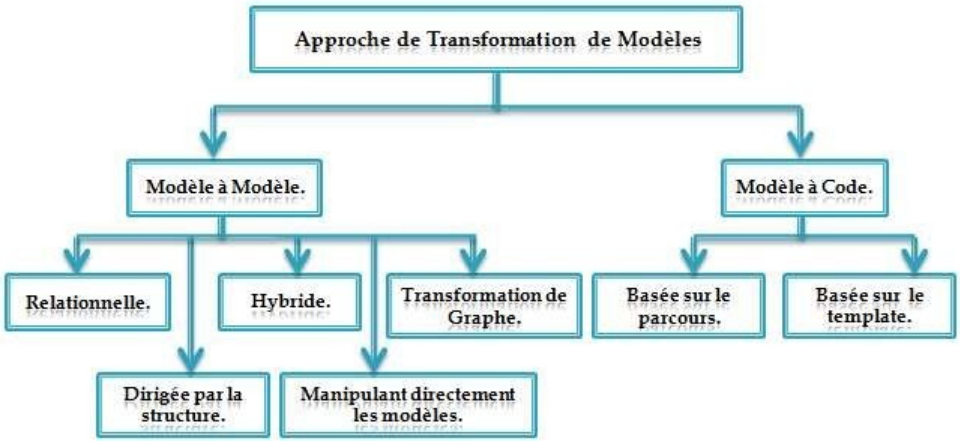


FIGURE I.3 – Les approches de transformation de modèles [3].

I.3.7 Propriétés d'une transformation :

- ***Les règles de transformation :***

Une règle de transformation est partagée en deux parties : une partie gauche (LHS Left Hand Side) accédant au modèle source, et une autre partie droite (RHS Right Hand Side) qui accède au modèle cible. La partie logique (déclarative ou impérative) de la règle comporte les calculs à effectuer sur les modèles ainsi que les contraintes appliquées.

Une logique déclarative spécifie les relations entre les éléments des modèles source et cible.

Une logique impérative utilise généralement des langages de programmation pour manipuler les éléments des modèles sur des interfaces dédiées [11].

- ***La relation entre le modèle source et le modèle cible :***

La relation entre le modèle source et le modèle cible dépend du type de la transformation. Nous parlons de transformation endogène lorsque le modèle source et le modèle cible sont exprimés

dans le même formalisme. La transformation est exogène lorsque les deux modèles sont exprimés dans deux formalismes différents [11].

- ***L'organisation des règles :***

C'est une organisation qui définit la stratégie selon laquelle les règles seront appliquées. Ces règles peuvent être organisées de façon modulaire avec importation. Les règles peuvent utiliser le principe de réutilisation en passant par le mécanisme d'héritage entre les règles, ou la composition en passant par l'ordonnancement explicite. Les règles peuvent être organisées aussi selon une structure dépendante du modèle source ou du modèle cible [11].

- ***Réversibilité :***

Une transformation est dite réversible si elle peut se faire dans les deux sens. Exemple : Model to text et t ext to Model [8].

- ***Traçabilité :***

La traçabilité permet de garder des informations sur le devenir des éléments des modèles au cours des différentes transformations qu'ils subissent. Dans un contexte d'ingénierie dirigée par les modèles, il est normal que l'information relative à la traçabilité soit représentable par un modèle. Une instance de ce modèle est donc associée à chaque exécution d'une transformation tracée. La définition d'un méta-modèle de traces nous permet de structurer les traces qui seront générées par la plate-forme de traçabilité et ainsi de mieux les manipuler [8].

- ***Réutilisabilité :***

La réutilisabilité permet de réutiliser des règles de transformation dans d'autres transformations de modèles. L'identification de patrons de transformation est un moyen pour mettre en œuvre cette réutilisabilité [8].

- ***Ordonnancement des règles :***

L'ordonnancement consiste à représenter les niveaux d'imbrication des règles de transformation. En effet, les règles de transformations peuvent déclencher d'autres règles [8].

- ***Modularité :***

Une transformation modulaire permet de modéliser les règles de transformation en faisant un découpage du problème. Un langage de transformation de modèles supportant la modularité facilite la réutilisation des règles de transformation [8].

I.3.8 Manipulation des modèles [5] :

Dans le domaine de l'ingénierie dirigée par les modèles, nous pouvons trouver plusieurs activités liées à la manipulation des modèles. Chacune appartient à un contexte spécifique. Parmi ces activités nous pouvons citer :

- ***La réalisation des modèles :***

Une bonne maîtrise du langage utilisé dans la modélisation et une expertise technique sont nécessaires pour la réalisation des modèles. Plus les systèmes sont complexes, plus leurs modèles le deviennent et plus leurs tailles deviennent importantes. Une bonne stratégie technique (outils et supports) s'avère une tâche délicate pour réaliser et manipuler les modèles.

- ***Le stockage des modèles :***

Cette activité concerne les formats de stockage, l'organisation du stockage et la gestion des métadonnées des modèles.

- ***L'échange des modèles :***

Pour une bonne communication entre eux, les acteurs d'un projet doivent procéder à l'échange de leurs modèles. Ces derniers doivent être compréhensibles par tous, au risque d'exposer le système implémenté à des dégâts. L'échange de modèles prend en considération les contraintes du format (sérialisation, transport, etc.), les contraintes de traduction ainsi que celles d'interprétation de la sémantique pour leur interopérabilité.

- ***L'interrogation des modèles :***

L'interrogation des modèles est l'activité qui permet de rechercher et de récupérer des informations dans les modèles.

- ***L'exécution des modèles :***

Cette étape va de la simulation du système à son exécution en temps-réel, en passant par l'exécution symbolique et la génération du code.

- ***La vérification des modèles :***

La vérification d'un modèle consiste à vérifier ses propriétés propres par rapport à ce que l'on attend de lui. Cette vérification est syntaxique et sémantique. La vérification sémantique est la plus complexe. Il existe différentes techniques pour procéder à la vérification d'un modèle, avec chacune ses avantages et ses inconvénients. Nous citons : la technique de preuves qui s'appuie sur les représentations formelles du système basées sur la logique, les automates, les réseaux de Petri, etc. Une autre technique est le ''model-cheking" qui vise à analyser le comportement du système tout en vérifiant des propriétés telles que la sûreté, l'atteignabilité ou la vivacité. Cette technique apporte avec elle un risque d'explosion combinatoire du nombre d'états dans le cas des systèmes complexes. Le test complète le model-cheking dans le cas où ce dernier n'a pas été particulièrement efficace (cas de système très complexe).

- ***La validation des modèles :***

La validation affirme ou non que le système implémenté répond aux besoins initiaux. Certaines techniques de tests sont utilisables pour la vérification mais également pour la validation. Les modèles génèrent des scénarios et des vecteurs de test de façon automatique.

- ***La gestion de l'évolution des modèles :***

Les modèles évoluent tout au long du cycle de développement du système. Ils peuvent être corrigés ou modifiés en recevant d'autres fonctionnalités. Ces modifications sont automatiquement répercutées sur les autres modèles impliqués .

I.4 L'Architecture Dirigée par les Modèles (ADM) :

I.4.1 Définition :

Partant du constat de l'évolution continue des technologies et du coût élevé l"adaptation des applications logicielles à ces technologies, l'OMG (Object Management Group) proposa le ADM fin 2000 Pour des raisons de productivité, l'approche ADM préconise l'utilisation de plusieurs modèles indépendants des détails techniques de l'implémentation. Cette méthode consiste en l'élaboration et la transformation de modèles tout au long du processus de développement d'un système. Ces modèles ont pour objectif de simplifier la gestion de la complexité des systèmes en spécifiant différents niveaux d'abstraction, aussi bien pour la vue globale du système que pour les protocoles et les algorithmes. Ces modèles sont reliés par des liens de traçabilité et peuvent être exprimés de façon textuelle ou graphique [5] [9].

En résumé, l'ADM est une démarche de développement basée sur les modèles et un ensemble de standards de l'OMG. Cette démarche permet de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plateforme donnée. Parmi les standards fondamentaux de l'OMG liés à l'ADM nous pouvons citer [4] :

- ***UML (UnifiedModelingLanguage) :***

UML est l'un des premiers méta-modèles basés sur le MOF adopté par l'OMG. La notation UML est décrite sous forme d'un ensemble de diagrammes [2].

- ***MOF (Meta-Object Facility) :***

Le MOF est le méta-méta-modèle standard unique. Permettant de définir la syntaxe et la sémantique d'un langage de modélisation [2].

- ***XMI (XML MetadataInterchange) :***

XMI est le format qui va permettre l'échange de modèles et de méta-modèles. Il est basé sur XML [10]. En effet, XMI définit des règles permettant de construire des DTD (Document Type Définition) et des schémas XML à partir de méta-modèles, et inversement [2].

- ***OCL (Object ConstraintLanguage) :***

En utilisant uniquement UML est un langage d'expression permettant de décrire des contraintes sur des modèles. Une contrainte est une restriction sur une ou plusieurs valeurs d'un modèle non

représentable en UML [2].

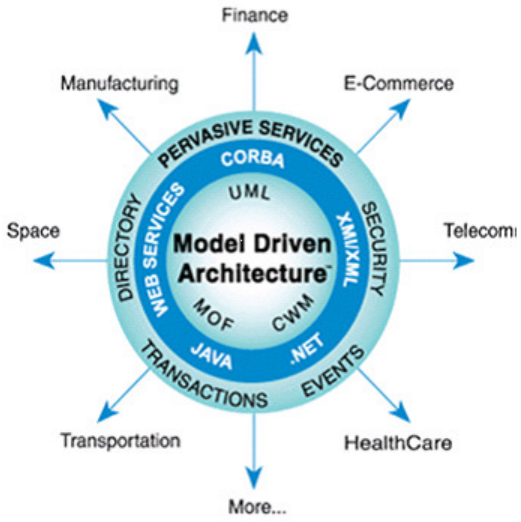


FIGURE I.4 – Les standards de l'Architecture Dirigée par les Modèles [9].

I.4.2 Modèles de l'ADM :

Dans l'approche ADM, l'OMG a défini plusieurs modèles qui vont servir dans un premier temps à modéliser l'application puis, par transformations successives, à générer le code de l'application. Les quatre principaux types de modèles définis dans l'approche ADM sont les suivants [1] :

- ***CIM (Computation Independant Model) :***

Appelé aussi modèle de domaine ou modèle métier, le CIM correspond au modèle des besoins au niveau métier. Il permet de recenser les différents besoins du client indépendamment de toute implémentation [11].

- ***PIM (Platform Independant Model) :***

Le PIM correspond au modèle de spécification de la partie métier d'une application. Cette spécification doit être conforme à une analyse informatique cherchant à répondre aux besoins métiers indépendamment de la technologie de mise en œuvre [11].

- ***PDM (Platform Description Model) :***

Le PDM est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit [11].

- ***PSM (Platform Specific Model) :***

Le PSM est le résultat de la combinaison du PIM et du PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation [11].

I.4.3 Transformation de Model en ADM :

L'ADM considère le processus de développement du système comme une suite successive et stratégique de transformations de modèles. Ces transformations établissent de manière automatique des liens de traçabilité entre les trois types de modèles discutés précédemment, La figure 1.5 montre les transformations possibles [1].

- ***Transformations PIM PIM et PSM PSM :***

Les transformations de type PIM vers PIM ou PSM vers PSM visent à enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle [1].

- ***Transformation PIM PSM :***

La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plateforme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM [1].

- ***Transformation PSM Code :***

La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte. Le code est parfois assimilé à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle et il est alors nécessaire de le compléter manuellement [1].

- ***Transformations inverses PSM PIM et code PSM :***

Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformations pose de nombreuses difficultés mais il est essentiel pour la réutilisation de l'existant dans le cadre de l'approche ADM [1].

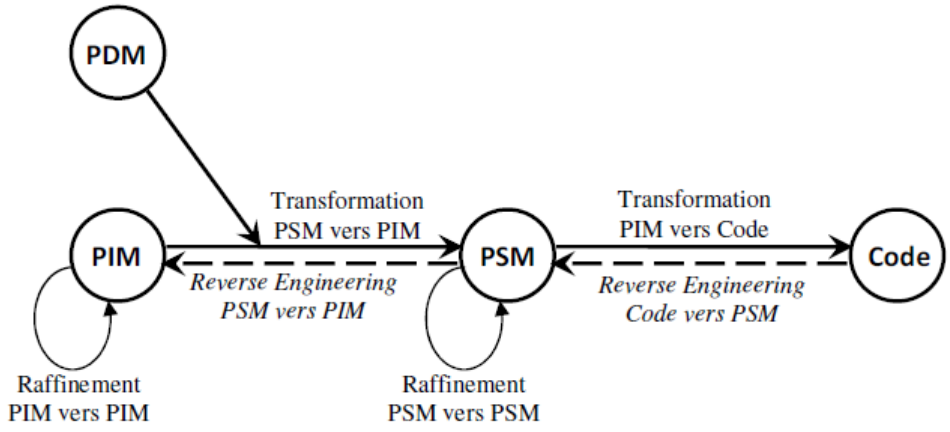


FIGURE I.5 – Les modèles et les transformations dans l'approche ADM [1].

I.4.4 L'architecture à quatre niveaux :

L'OMG, dans le cadre de ses travaux concernant la méta-modélisation, a défini la notion de méta-méta-modèle ainsi que la standardisation d'une architecture générale décrivant les liens entre modèles, méta-modèles et méta-méta-modèles. Cette architecture est hiérarchisée en quatre niveaux comme le montre la Figure I.6 [10].

Le niveau M0 : *se trouve le système à étudier, c'est le niveau des instances des modèles. Représente les différents sujets de modélisation Il définit des informations pour la modélisation des objets du monde réel [10].*

Le niveau M1 : *se trouve le modèle qui décrit certains aspects du premier niveau que nous voulons l'étudier : il peut s'agir d'un diagramme de classes UML, d'un modèle conceptuel de traitement MERISE, qui représente une vue abstraite des objets modélisés [10].*

Le niveau M2 : *se trouve le langage de modélisation ou méta-modèle : par exemple les définitions d'un MCD de MERISE d'un diagramme d'objet UML. C'est ce langage qui doit faire l'objet d'une spécification formelle [10].*

Le niveau M3 : *se trouve le méta-méta-modèle il s'agit d'un langage qui doit être assez générique pour définir les différents langages de modélisation existants, et assez précis pour exprimer les règles que chaque langage doit respecter pour pouvoir être traité automatiquement. Des exemples de méta-méta-modèles : DSL Tools de Microsoft, MOF (Meta Object Facility), KM3 (Kernel MetaMetaModel [10].*

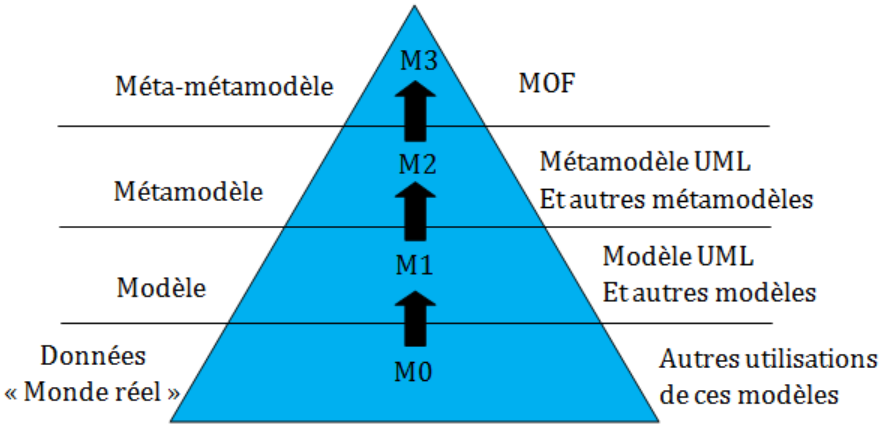


FIGURE I.6 – Les quatre niveaux d'abstraction pour ADM [6].

I.5 Les transformations de Graphes :

Comme les grammaires de Chomsky pour des textes, la transformation de graphes se fait grâce à des règles de transformation. Ces règles sont appliquées sur un graphe, et si les règles trouvent plusieurs concordances, elles effectuent les changements d bbéterminés en opérant des remplacements, des ajouts ou des suppressions. Donc le processus de transformation s'effectue, en partant d'un graphe de départ, produire un graphe d'arrivée qui pourra être modifié ensuite.

Une règle est constituée de deux parties, le Left Hand Side (LHS) et le Right Hand Side (RHS). Le LHS est une partie du graphe, destinée à être mise en concordance avec les parties du graphe (appelé host graph) où nous voulons appliquer la règle. Le RHS quant à lui décrit la modification qui sera effectuée, elle substitue la partie identifiée dans le host graph [10].

I.5.1 Notion de graphe :

Un graphe est constitué d'un ensemble de sommets reliés par des arêtes. Deux sommets reliés par une arête sont adjacents.

L'ordre d'un graphe est le nombre de ses sommets.
Le degré d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité.
La figure 1.6 illustre un exemple d'un graphe [5].

Graphe orienté :*un graphe orienté est un graphe dont les arêtes sont munies de directions : nous parlons alors de l'origine et de l'extrémité d'une arête. Dans un graphe orienté, une arête est appelée « arc » .*

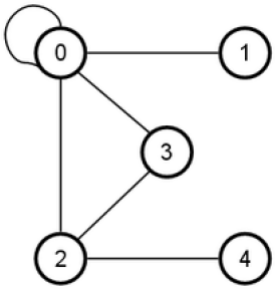


FIGURE I.7 – Exemple d'un graphe G [5].

La figure 1.8 illustre un exemple de graphe orienté [5].

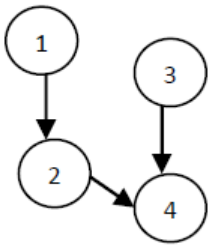


FIGURE I.8 – Graphe orienté [5].

Graphe non orienté : *un graphe est constitué de sommets (nœuds) qui sont reliés par des arêtes. Deux sommets reliés par une arête sont adjacents [5].*

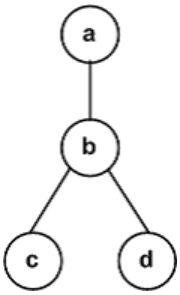


FIGURE I.9 – Graphe non orienté étiqueté avec des lettres [11].

Un sous-graphe d'un graphe G est un graphe G' composé de certains sommets de G, ainsi que de toutes les arêtes qui relient ces sommets. La figure 1.10 illustre un exemple où le graphe (b) est un sous-graphe du graphe (a) [5].

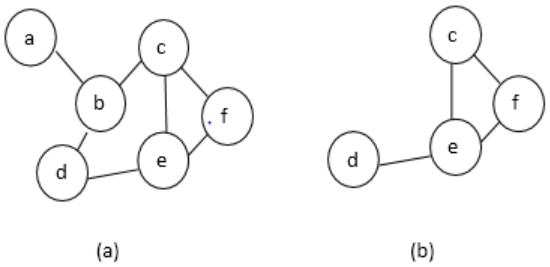


FIGURE I.10 – Graphe (a) et sous-graphe (b) [5].

I.5.2 Grammaire de graphe :

Une grammaire de graphe est généralement définie par un triplet :
GG = (P, S, T)
Où :
P : ensemble de règles.
S : un graphe initial.
T : ensemble de symboles.

Une grammaire de graphes distingue les graphes non terminaux, qui sont les résultats intermédiaires sur lesquels les règles sont appliquées, des graphes terminaux sur lesquels on ne peut plus appliquer aucune règle. Ces derniers sont dans le langage engendré par la grammaire de graphe, pour vérifier si un graphe G est dans le langage engendré par une grammaire de graphe, il doit être analysé. Le processus d'analyse va déterminer une séquence de règles dérivant G [5].

I.5.3 Règle de transformation :

Une règle de transformation de graphe « R » est définie par un 6-uplet Où :
R= (LHS, RHS, K, glue, emb, cond)
LHS : C'est le graphe de partie gauche.
RHS : C'est le graphe de partie droite.
K : C'est un sous graphe de LHS.
glue : C'est une occurrence de K dans RHS qui relie le sous graphe avec le graphe de partie droite.
emb : C'est une relation d'enfoncement qui relie les sommets du graphe de la partie gauche et ceux du graphe de la partie droite.
cond : C'est un ensemble qui indique les conditions d'application de la règle [4].

I.5.4 Système de transformation de graphe :

Un système de transformation de graphes est un système de réécriture de graphe qui applique les règles de la grammaire de graphe sur son graphe initial de façon itérative et dans un ordre bien

défini jusqu'à ce que plus aucune règle ne soit plus applicable [4].

L'application d'une règle R= (LHS, RHS, K, glue, emb, cond) à un graphe G produit comme résultat un graphe H en passant par les cinq étapes suivantes :

Première étape :

Choisir une occurrence du graphe de partie gauche LHS dans G [4].

Deuxième étape :

Vérifier les conditions d'application d'après cond [4].

Troisième étape :

Retirer l'occurrence LHS (jusqu'à K) de G ainsi que les arcs pendillés (tous les arcs ayant perdu leurs sources et/ou leurs destinations). Ce qui fournit le graphe de contexte D de LHS qui a laissé une occurrence de K [4].

Quatrième étape :

Coller le graphe de contexte D et le graphe de partie droite RHS suivant l'occurrence de K dans D et dans RHS, c'est la construction de l'union de disjonction de D et RHS, et pour chaque point dans K, identifier le point correspondant dans D avec le point correspondant dans RHS [4].

Cinquième étape :

Enfoncer le graphe de partie droite dans le graphe de contexte de LHS suivant la relation d'enfoncement emb : pour chaque arc incident retiré avec un sommet v dans D et avec un sommet v' dans l'occurrence de LHS dans G, et pour chaque sommet v" dans RHS, un nouvel arc incident est établi (même étiquette) avec l'image de v et le sommet v" à condition que (v', v") appartient à emb [4].

L'application de R à un graphe G pour fournir un graphe H est appelée une dérivation directe depuis G vers H à travers R, elle est dénotée par G = H.

Soit S un graphe initial, le langage engendré L (P, S, T) est l'ensemble des graphes dérivés à partir de S en appliquant les règles de P qui sont étiquetées par les symboles de T.

La figure (I.9) illustre les différentes étapes d'application d'une règle de transformation. Tandis que la figure I.11 représente le principe du système de réécriture de graphes [4].

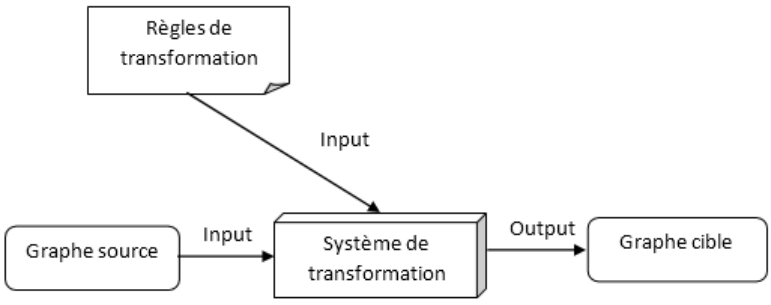


FIGURE I.11 – Système de réécriture de graphe [5].

I.5.5 Outils de transformation de graphes :

Il existe plusieurs outils de transformation de graphes. On peut citer quelques exemples d'outils de transformation de graphes [2] :

- ***AGG [The Attributed Graph Grammar System] :*** *c'est l'un des outils de transformation de graphes les plus cités dans la littérature. Il est implémenté en langage Java [2].*

- ***FUJABA [Fujaba] :(FromUML to Java and back again) :*** *un outil très puissant utilisé principalement pour la génération de code Java à partir de diagrammes UML il est très utilisé dans la modélisation et la simulation des systèmes mécaniques et électriques [2].*

- ***ATOM3 [Atom3] :(A Tool for Multi-formalism and Meta-Modelling) :*** *AToM3 est un outil visuel dédié à la transformation de graphes, implémenté en langage Python. Il possède une couche de Méta-Modélisation permettant une spécification syntaxique et graphique des formalismes utilisés [2].*

- ***VIATRA [Viatra] :(VisualAutomated model TRAnsformations) :*** *pour définir les règles de transformations. L'ordonnancement dans l'application des règles est basé sur une machine à états abstraite [2].*

- ***GreAT [Great] :(The Graph Rewrite And Transformation tool suite) :*** *un outil permettant la définition des transformations unidirectionnelles de plusieurs modèles sources vers plusieurs modèles cibles. Il est basé sur une notation graphique pour la spécification des règles de transformations [2].*

- ***TGG :*** *un environnement spécifique aux transformations bidirectionnelles. Il est très utile pour assurer la synchronisation et le maintien de correspondance entre les deux modèles sources et destination [2].*

- ***EMG [Eclipse Modeling Galileo] :*** *un environnement spécifique aux transformations de graphes intégré à la plateforme éclipse. Le framework principal est celui utilisé pour la modélisation. Il est appelé Eclipse Modeling Framework (EMF) [2].*

I.6 Conclusion :

Dans ce chapitre, nous avons défini le contexte dans lequel se place ce mémoire et détaillé les principaux concepts du domaine de l'ingénierie des modèles. Nous avons présenté le concept de transformation de modèles et plus précisément la transformation de graphes, une discipline importante dans la démarche ADM. Dans ce contexte, nous avons présenté les différentes approches, méthodes et outils existants dans la littérature. Les concepts présentés dans ce chapitre constituent un background nécessaire pour la compréhension de notre travail dans le cadre de ce mémoire.

Chapitre II

Modélisation semi-formelle avec UML2.0

II.1 Introduction :

Notre approche de développement, comme toutes les approches de type MDA, repose sur l'utilisation de modèles décrits dans un langage de modélisation clair et précis. Pour cet effet, l'OMG met en avant le langage UML qui est très populaire dans l'industrie.

UML est un langage graphique et textuel permettant de représenter les divers aspects d'un système. Il est destiné à comprendre et à définir des besoins, spécifier et documenter des systèmes, esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vue.il possède treize diagrammes devisés en deux catégories statique et dynamique. Statique pour la représentation statique du système et dynamique pour représenter le comportement et la communication du système. La flexibilité d'UML est un facteur très important, car UML peut être étendu et personnalisé par l'utilisation du mécanisme des profils.

Dans ce chapitre nous commençons par une brève présentation d'UML, en suite nous présentons en détail le diagramme d'états transitions, ces concepts de base, ces concepts avancés ainsi que ses intérêts.

II.2 La modélisation :

La modélisation d'un système est le processus de développement des modèles, un modèle est une représentation abstraite d'un système réel, construit pour un objectif donné et qui contient un ensemble restreint d'informations sur le système modélisé. Le modèle construit contient toujours des informations pertinentes vis-à-vis de son utilisation future.

D'autre part, la modélisation offre des avantages considérables aux concepteurs des systèmes tels que : la facilité de compréhension du fonctionnement des systèmes avant sa réalisation et un bon moyen de maîtriser sa complexité et d'assurer sa cohérence.

La modélisation en informatique peut être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelle (telles que : la sécurité, la fiabilité, l'efficacité, la performance, la ponctualité, la flexibilité, etc.) [7].

II.2.1 Pourquoi modéliser ?

Modéliser un système avant sa réalisation permet de mieux comprendre le fonctionnement du système. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence. Un modèle est un langage commun, précis, qui est connu par tous les membres de l'équipe et il est donc, à ce titre, un vecteur privilégié pour communiquer. Cette communication est essentielle pour aboutir à une compréhension commune aux différentes parties prenantes (notamment entre la maîtrise d'ouvrage et la maîtrise d'œuvre informatique) et précises d'un problème donné.

Dans le domaine de l'ingénierie du logiciel, le modèle permet de mieux répartir les tâches et d'automatiser certaines d'entre elles. C'est également un facteur de réduction des coûts et des délais. Par exemple, les plateformes de modélisation savent maintenant exploiter les modèles pour faire de la génération de code (au moins au niveau du squelette) voire des allers-retours entre le code et le modèle sans perte d'information.

Le modèle est enfin indispensable pour assurer un bon niveau de qualité et une maintenance efficace. En effet, une fois mise en production, l'application va devoir être maintenue, probablement par une autre équipe et, qui plus est, pas nécessairement de la même société que celle ayant créée l'application [26].

II.2.2 Les types de modélisation :

La classification de la modélisation peut se faire selon le degré du formalisme des langages ou des méthodes impliquées dans le processus de la modélisation. Ainsi, la modélisation peut être considérée comme étant formelle, semi-formelle ou informelle.

- ***Modélisation Informelle [7] :***

Le processus de modélisation informelle à base de langages informels, se justifie selon pour plusieurs raisons :
- La facilité de compréhension du langage permet des consensus entre les personnes qui spécifient et celles qui commandent un logiciel.
- elle représente une manière familière de communication entre personnes.

Par ailleurs, l'utilisation d'un langage informel rend la modélisation imprécise et parfois ambigüe . Le caractère informel de cette approche rend difficile toute tentative de standardisation.

- ***Modélisation Semi-Formelle [7] :***

Le processus de modélisation semi-formelle est basé sur un langage textuel ou graphique pour lequel une syntaxe précise est définie. La sémantique d'un tel langage est souvent assez faible. Néanmoins, ce type de modélisation permet d'effectuer des contrôles et de réaliser des automatisations pour certaines tâches.

La plupart des méthodes de modélisation semi-formelles s'appuient fortement sur des langages graphiques. Ceci se justifie par la puissance expressive du modèle graphique. Par ailleurs, l'appui de la modélisation semi-formelle (tels que : UML) sur des langages graphiques, permet la production de modèles assez faciles à interpréter.

Cependant, cette modélisation souffre de la déficience des aspects sémantiques impliqués dans l'approche. Afin de pallier aux insuffisances de cette approche, l'utilisation de contraintes a été introduite.

- ***Modélisation Formelle [1] :***

Les méthodes formelles sont des techniques basées sur les mathématiques pour décrire des propriétés de systèmes. Elles fournissent un cadre très rigoureux pour spécifier, développer et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc, afin de démontrer leur validité par rapport à une certaine spécification.

Ces méthodes permettent d'obtenir une très forte assurance de l'absence des incohérences, des contradictions et des failles dans la conception aussi bien qu'à déterminer la correction de l'implantation d'un système. Cependant, elles sont généralement coûteuses en ressources (en termes de temps et d'argent) et réservées aux systèmes critiques. Leur instrumentation et outillage sont la motivation de nombreuses recherches pour élargir leurs champs d'application.

II.3 Langage de modélisation UML :

UML est un langage de modélisation utilisant une représentation graphique basée sur les diagrammes. L'usage d'une représentation graphique est un atout car les diagrammes effacent les ambigüités dans les modèles. Un dessin exprime de manière plus naturelle et plus lisible ce qu'un texte peine à réaliser même lorsqu'il est bien commenté.

UML est un langage qui permet de modéliser non seulement des applications informatiques ou des structures de données, mais également les activités d'un domaine : mécanique, biologie, processus métier...

Plus précisément, UML permet d'offrir des outils d'analyse, de conception et d'implémentation des systèmes logiciels, ainsi que pour la modélisation d'entreprise et des systèmes non logi-

ciels. Ce langage de modélisation unifié repose sur deux concepts essentiels : 1- La modélisation du monde réel au moyen de l'approche orientée objet. 2- L'élaboration d'une série de diagrammes facilitant l'analyse et la conception des systèmes, et permettant de représenter les aspects statiques et dynamiques du domaine à modéliser ou à informatiser [13].

II.4 Historique des méthodes de conception :

Le langage UML est né de la mise en commun des trois plus importantes méthodes de modélisation orientées objet du début des années 90 (au sein d'un même langage) [OMG, 1999] :
- La méthode Booch dont l'auteur, Grady Booch, travaillait au développement de systèmes en Ada chez Rational Software Corporation. - La méthode Object Modeling Technique (OMT) développée par Jim Rumbaugh qui dirigeait une équipe de recherche chez General Electric. - La méthode Object-Oriented Software Engineering (OOSE) résultant des travaux d'Ivar Jacobson sur les commutateurs téléphoniques chez son employeur Ericsson. La méthode OOSE était la première à introduire le concept des cas d'utilisation (use case).

Le langage UML a bénéficié de bien d'autres travaux comme ceux de Sally Shlaer et Steve Mellor sur l'analyse et la conception de systèmes, ou ceux de Jim Odell et James Martin en génie logiciel, mais aussi des nombreux acquis de la communauté Smalltalk. La fusion de ces trois méthodes a été réalisée très rapidement dans les années 94-96 lorsque Jim Rumbaugh rejoignit Grady Booch chez Rational Software Corporation en 1994. En 1995, cette dernière acquit la société Objectory AB d'Ivar Jacobson.

Tous trois, souvent surnommés Les trois amigos, rédigèrent la version 0.9 d'UML qui fut publiée à OOPSALA 96 en juin 1996, texte qui sera soumis à l'OMG en janvier 1997 sous le titre de version 1.0. Cette version sera amendée par des propositions émanant des membres de l'OMG et donnera lieu à la version 1.1 soumise en septembre 1997 et adoptée en novembre de la même année. En juin 1998, la version 1.2 apportera des modifications mineures au langage UML alors que la version 1.3 fera l'objet de modifications plus conséquentes comme l'intégration du langage de contraintes Object Constraint Language(OCL), ou la génération d'interfaces via le langage Interface Definition Language(IDL). Cette dernière sera soumise en juin 1999 et adoptée en mars 2000. La version 1.4, adoptée en septembre 2001, introduira des concepts et des spécifications détaillées sur les notions de composants et de profils. Adoptée en mars 2003, la version 1.5 ajoutera la sémantique des actions.

La version 2.0 a été l'occasion d'une refonte majeure du langage UML dont les spécifications ont été mises en accord avec celles du Meta Object Facility(MOF), méta-méta-modèle du langage UML. Pour ce faire, les membres de l'OMG ont extrait les concepts identiques du langage UML et du MOF et les ont mutualisés au sein d'une nouvelle architecture dénommée UML 2.0 Infrastructure. Cette nouvelle organisation a permis d'alléger les spécifications d'UML 2.0 et celles du

MOF 2.0. Les nouvelles spécifications d'UML sont décrites dans le document intitulé UML 2.0 Superstructure. L'infrastructure d'UML 2.0 a été adoptée en septembre 2003 et la superstructure en octobre 2003. La version UML 2.0 est l'aboutissement d'une collaboration intensive de 3 ans entre les membres de l'OMG.

Entre les versions 1.0 et 2.0, sept ans se sont écoulés et sept versions ont été adoptées par les membres de l'OMG ce qui correspond en moyenne à une version par an. Ceci montre l'intérêt et le dynamisme de cette communauté. La figure II.1 expose histoire d'ULM [27].

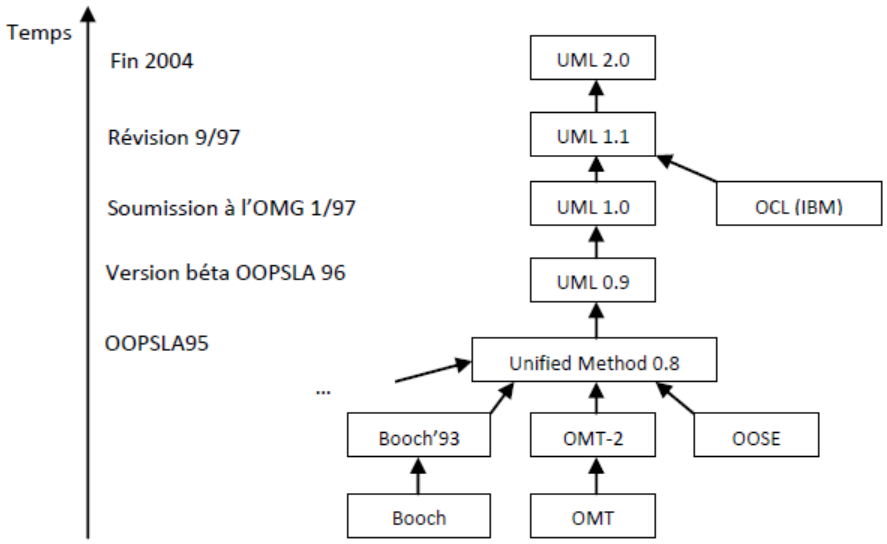


FIGURE II.1 – Historique de constitution du langage UML [5].

II.5 Les vues du langage UML 2.0 :

Il est impossible de donner une représentation graphique complète d'un logiciel, ou de tout autre système complexe, mais il est possible de donner sur un tel système des vues partielles, comme montré dans la figure II.2 , pour avoir une idée utilisable en pratique sans risque d'erreur grave [11].

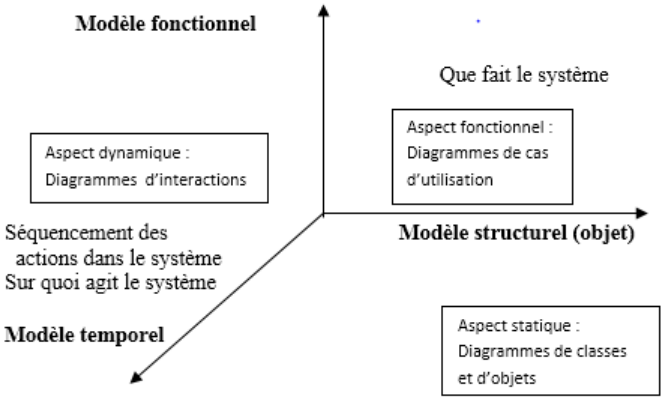


FIGURE II.2 – Les aspects d'un système [11].

Les vues UML permettent de mettre en évidence les différents aspects d'un système que nous souhaitons réaliser. UML 2.0 comporte ainsi treize types de diagrammes représentant autant de vues distinctes mais complémentaires pour représenter des concepts particuliers du système d'information. Ils se répartissent en trois grands groupes [11] :

La vue structurelle, ou statique [11] : *cette vue modélise la structure des différentes classes d'une application orientée objet, elle réunit :*

Diagramme de classes.
Diagramme d'objets.
Diagramme de composants.
Diagramme de déploiement.
Diagramme de paquetage.
Diagramme de structures composites.

La vue comportementale [11] : *cette vue est fonctionnelle, elle est plus algorithmique et orientée « traitement », et vise à décrire l'évolution (la dynamique) des objets complexes du programme tout au long de leur cycle de vie. De leur création à leur destruction, les changements d'états des objets sont guidés par les interactions avec les autres objets. Cette vue est présentée avec les diagrammes suivants :*

Diagramme de cas d'utilisation.
Diagramme d'activités.
Diagramme d'états-transitions.

En général, les diagrammes d'états à eux seuls ne permettent pas de faire apparaître les problèmes spécifiques posés par la synchronisation des processus en concurrence pour assurer la

cohérence du comportement et l'absence d'inter-blocage.

La vue dynamique ou d'interaction [11] : *pour montrer l'interactivité, des diagrammes traitent les interactions entre les différents acteurs/utilisateurs et le système sous forme d'objectifs à atteindre d'un côté, et sous forme chronologique de scénarios d'interaction typiques de l'autre. Ces diagrammes sont les suivants :*

Diagramme de séquence.
Diagramme de communication.
Diagramme global d'interactions.
Diagramme de temps.

La figure II.3 montre un concept UML 2.0 avec le lien entre les différentes vues et les abstractions.

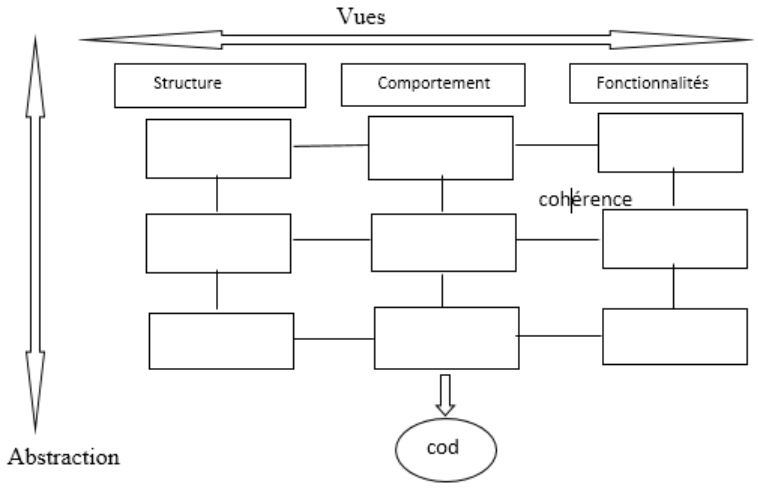


FIGURE II.3 – Différentes vues dans un concept UML 2.0 [5].

II.6 Diagrammes UML 2.0 :

Un diagramme est la représentation graphique d'un ensemble d'éléments qui constituent un système. La plupart du temps, il se présente sous forme d'un graphe connexe où les sommets correspondent aux éléments et les arcs aux relations. Les diagrammes servent à visualiser un système sous différentes perspectives et sont donc des projections dans un système. Pour les systèmes complexes, un diagramme ne représente qu'une vue partielle des éléments qui composent ces systèmes [4].

- ***Diagramme de cas d'utilisation :***

Les cas d'utilisation sont une technique de description du système étudié selon le point de vue de l'utilisateur. Ils décrivent sous la forme d'actions et de réactions le comportement d'un système. Donc, le diagramme des cas d'utilisation, permet d'identifier les possibilités d'interaction entre le système et les acteurs. Il permet de clarifier, filtrer et organiser les besoins [14].

- ***Diagramme de classes :***

Le but d'un diagramme de classes est d'exprimer de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes. Une classe a des attributs, des opérations et des relations avec d'autres classes [14].

- ***Diagramme d'objets :***

Le diagramme d'objet permet d'éclairer un diagramme de classe en l'illustrant par des exemples. Il montre des objets et des liens entre ces objets (les objets sont des instances de classes dans un état particulier)[14].

- ***Diagramme d'états-transitions :***

Permet de décrire sous forme de machine à états finis le comportement du système ou de ses composants. Il est composé d'un ensemble d'états, reliés par des arcs orientés qui décrivent les transitions [14].

- ***Diagramme d'activités :***

Un diagramme d'activité est une variante des diagrammes d'états-transitions. Il permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation, dans un diagramme d'activité les états correspondent à l'exécution d'actions ou d'activités et les transitions sont automatiques [14].

- ***Diagramme de séquence :***

Il représente séquentiellement le déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs. Le diagramme de séquence peut servir à illustrer un cas d'utilisation [14].

- ***Diagramme de communication :***

C'est une représentation simplifiée d'un diagramme de séquence, en se concentrant sur les échanges de messages entre les objets [14].

- ***Diagramme global d'interaction :***

Permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences (variante du diagramme d'activité)[14].

- ***Diagramme de temps :***

Le diagramme de temps permet de décrire les variations d'une donnée au cours du temps [14].

- ***Diagramme de composants :***

Il montre les composants du système d'un point de vue physique, tels qu'ils sont mis en œuvre (fichiers, bibliothèques, bases de données...). Il montre la mise en œuvre physique des modèles de la vue logique avec l'environnement de développement [14].

- ***Diagramme de déploiement :***

Ce type de diagramme UML montre la disposition physique des matériels qui composent le système (ordinateurs, périphériques, réseaux...) et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de nœuds, connectés par un support de communication [14].

- ***Diagramme des paquetages :***

Un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML, il sert à représenter les dépendances entre paquetages [14].

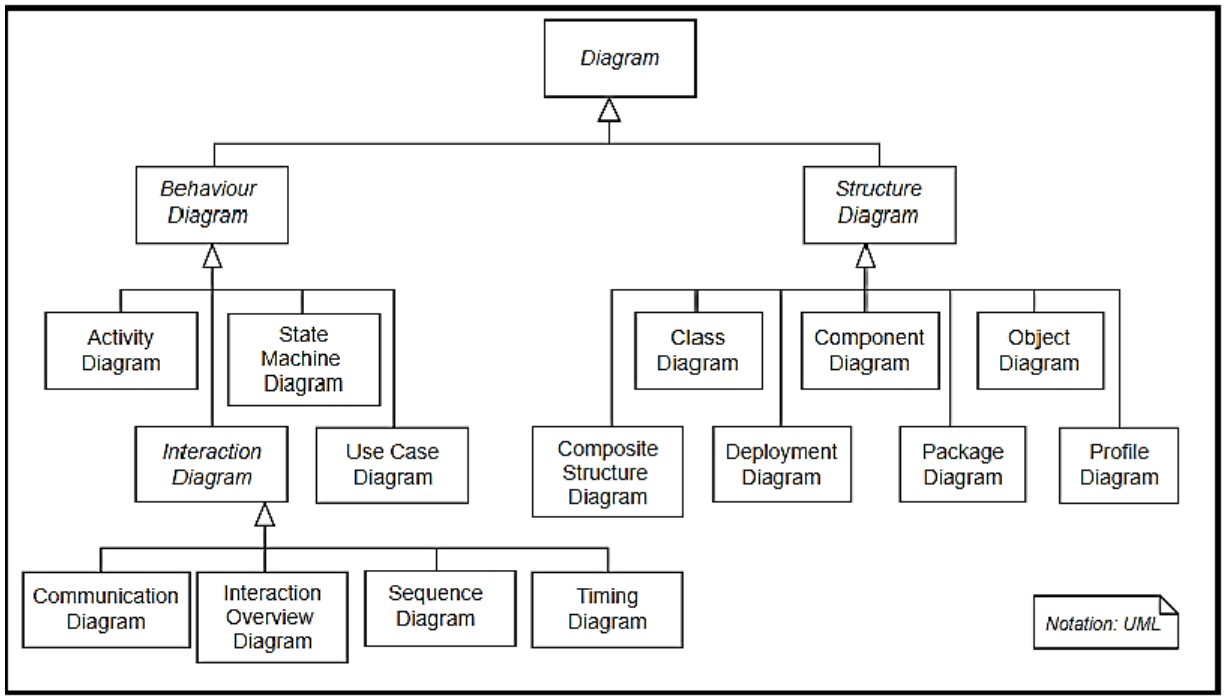


FIGURE II.4 – Les différents diagrammes d'UML 2.0 [4].

II.7 Les diagrammes d'états transitions :

II.7.1 Définition :

Les diagrammes d'états transition (statecharts diagram), concept utilisé par David Harel1 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. La figure II.5 exprime le protocole d'états- machine [12].

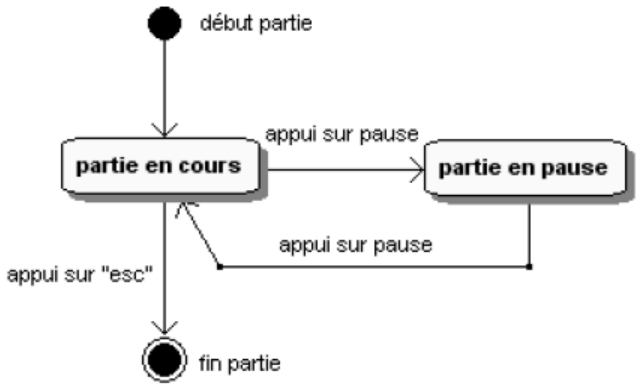


FIGURE II.5 – Diagramme d'états-transitions simplifié d'une partie d'un jeu vidéo [5].

II.7.2 Les concepts des diagrammes d'états-transitions :

II.7.2.1 Les états [12] :

Un état est la condition d'un objet à un moment donné, c'est une situation donnée durant la vie de l'élément qui satisfait à des conditions, réalise des actions ou est en attente d'évènement. 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.

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.

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.

Généralement un état est constitué de :

- ***Nom :*** *une chaine textuelle qui distingue l'état d'autres états (un état peut être anonyme).*
- ***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.*
- ***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.*
- ***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).*
- ***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.*
- ***Evénements rapportés :*** *c'est une liste d'évènements 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.*
- ***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.*
- ***Etat initial :*** *pseudo état qui indique l'état de départ par défaut de la région enveloppée.*
- ***Etat final :*** *pseudo état qui indique que l'exécution du diagramme ou de l'état composite est terminée. La figure 2.3 désigne le formalisme de représentation des états initial et final.*

II.7.2.2 Les transitions [12] :

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 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'évènement, une condition de garde, une action et un état cible, certains éléments peuvent ne pas figurer. Généralement, une transition est constituée de :

- ***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'évènement déclencheur de la transition de gard (soit satisfaite), l'état source devient inactif après le déclenchement de la transition.*
- ***Etat cible :*** *l'état cible est l'état qui devient actif après l'achèvement de la transition.*
- ***Déclencheur d'évènement :*** *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 vérifié. Un événement peut être un signal, un appel de méthode, un passage de temps ou un changement d'état.*
- ***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 ».*
- ***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'évènement déclencheur, ainsi que des attributs et des associations de l'objet détenteur.*

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'a une transition. Le tableau suivant illustre les types et les effets implicites.

II.7.2.3 Les évènements [12] :

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'évènement 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. La figure II.6 représente le protocole d'un diagramme états transitions .

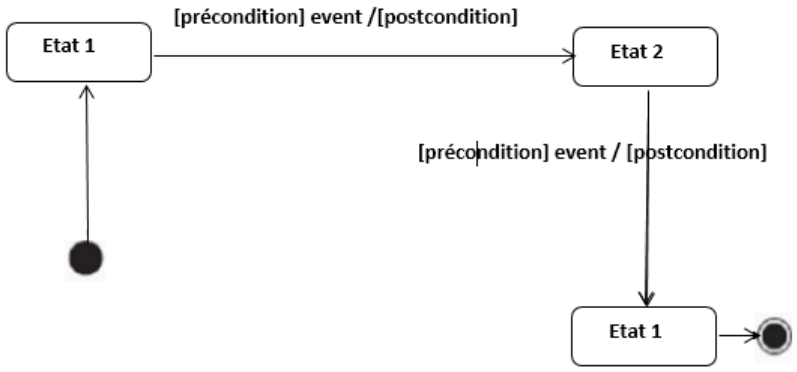


FIGURE II.6 – le protocole d'un diagramme d'états-transitions de Harel [12].

II.7.3 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 [12].

• ***Activités d'entrée /sortie :*** *un état peut avoir une activité entry et une activité exit. L'objectif est d'encapsuler l'état de façon qu'il soit utilisable en externe sans connaissance de sa structure interne. Une activité entry est exécutée lors de l'entrée dans l'état (après toute activité rattachée à la transition entrante et avant toute activité do interne). Une activité exit a lieu lors de la sortie de l'état (après l'achèvement de toutes les activités do internes et avant toute activité rattachée à la transition sortante).*

Les activités d'entrée/sortie généralement ne possèdent pas d'arguments ou de conditions de garde. Toutefois, l'activité d'entrée pour le premier état dans le diagramme d'états- transitions peut avoir des paramètres pour les arguments reçus lors de la création de l'objet [12].

• ***Transition interne :*** *un état peut avoir une liste de transitions internes qui ressemblent aux transitions normales, mais qui n'ont pas d'états cible et qui ne provoquent pas de changement d'état. Leur événement se produit pendant qu'un objet se trouve dans l'état détenant la transition*

ou dans l'un de ses sous-états imbriqués, alors l'action de la transition interne s'exécute, mais il n'ya pas de changement d'état et aucune activité entry ou exit ne s'exécute [12].

• ***Activité do :*** *un état peut contenir une activité do décrite par une expression donnée, lorsque un objet est en état donné, il reste généralement en attente de l'arrivée des événements, mais parfois, on voudra accomplir des activités à ce moment jusqu'à ce que l'évènement apparaisse. UML2 utilise le mot clé « do » pour spécifier les activités jusqu'à ce que l'évènement apparaisse. Chronologiquement, lors de l'entrée dans l'état, l'activité do commence après que l'activité entry est terminée, l'état est à terme. Une transition d'achèvement se déclenche alors et quitte l'état. Sinon, l'état attend une transition déclenchée pour provoquer le changement d'état. Si une transition se déclenche pendant que l'activité do est en cours, cette dernière est interrompue et l'activité exit de l'état s'exécute [12].*

• ***Evénements différés :*** *les événements différés sont des événements qui ne sont pas traités par l'objet dans cet état, mais plutôt ces événements seront gardés dans une file d'attente pour les traiter par l'objet dans d'autres états.*

UML utilise l'action « différer » pour définir des événements différés ; c'est à dire des événements qui, sils ne déclenchent pas de transition lors de leur occurrence, ne sont pas perdus mais mémorisés dans une file d'attente jusqu'à ce qu'ils soient acceptés ou que l'automate soit dans un état ou ils ne sont plus différés (alors dans ce cas, les événements sont perdus) [12].

• ***Sous machine :*** *le corps d'un état peut représenter une copie d'une machine d'état distincte référencée par son nom. Cette dernière est appelée « sous machine », en anglais sub-machine, car elle est imbriquée dans la machine d'états plus étendue. La sous machine est prévue pour être réutilisée dans plusieurs machines d'états afin d'éviter la répétition du même fragment de machine d'état. Une sous machine est une sorte de machine d'état [12].*

• ***Sous état :*** *les diagrammes d'états-transitions deviennent illisibles du fait de l'explosion combinatoire, le nombre de connexions entre états est élevé, la solution adoptée pour maîtriser cette situation est l'utilisation des états composites. Un état composite (ou état englobant) est décomposé en sous états.*

Un sous état est un état imbriqué dans un autre état. Chaque sous état peut également être un composite, c'est-à-dire décomposé en d'autres sous états imbriqués à un niveau hiérarchique inférieur. Les sous états sont des sous états concurrents (orthogonaux) et/ou des sous états séquentiels (non orthogonaux) [12].

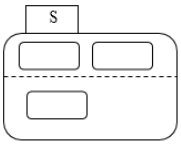
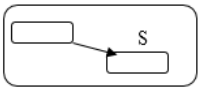
• ***Sous état séquentiel :*** *un état composite est composé d'une ou plusieurs régions, chacune peut contenir un ou plusieurs états. Un état composite possédant une seule région est dit état séquentiel (non concurrent, non orthogonal), généralement la région séquentielle contient un état initial et un état final [12].*

• ***Sous état orthogonal :*** *le concept état orthogonal permet de présenter efficacement le mécanisme de concurrence. Un état orthogonal est un état composite contenant plus d'une ré-*

gion, chaque région représente un flot d'exécution. Graphiquement, dans un diagramme d'états-transitions les différentes régions sont séparées par un trait en pointillé allant du bord gauche au bord droit de l'état composite. Un état peut être composé de plusieurs sous états concurrents (appelés régions). Cette composition est de type conjonctive (et), ce qui implique que l'objet doit être simultanément dans tous les sous états. La conjonction d'états représente une forme de parallélisme entre automates [12].

- ***Etat historique :*** *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 ou un historique profond 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*.*

Pour résumer un état simple ne possède pas de sous structure mais uniquement un jeu de transitions et éventuellement des activités entry et exit. Un état composite est un état décomposé en régions contenant un ou plusieurs états, le tableau II.1 décrit les différents types de états [12].

Type état	***Notation***	***Description***
État simple		Etat dépourvu de sous structure
Etat orthogonal		Etat divisé en deux régions ou plus,un sous état direct de chaque région est simultanément actif avec l'état composite lorsque ce dernier est actif.
Etat non orthogonal		Etat composite contenant un ou plusieurs sous état directs ; un seul d'entre eux est exactement actif à la fois lorsque l'état composite est actif.
Etat initial		Pseudo –état, qui indique l'état de départ lorsque l'état enveloppant est invoqué

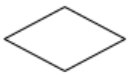
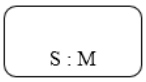
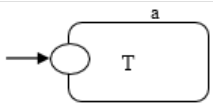
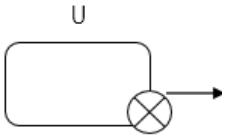
Etat final		Eta spécial dont l'activation indique que l'état enveloppant est terminé
Terminaison		Etat spécial dont l'activation achève l'exécution de l'objet de la machine d'états.
Jonction		Pseudo–état qui relie des segments de transition en une seule transition de type RTC (run–to- completion)
Choix		Pseudo état qui crée un embranchement dynamique dans une transition de type RTC
Etat historique		Pseudo état dont l'activation restaure l'état précédemment actif dans un état composite
Etat de sous -machine		Etat qui indique une définition de machine d'états qui remplace conceptuellement l'état de sous machine.
Point d'entrée		Pseudo–état visible de l'extérieur dans une machine d'états qui identifie un état interne comme cible
Point de sortie		Pseudo état visible de l'extérieur dans une machine d'états qui identifié un état interne comme une source.

TABLE II.1 – Types d'états [7].

• ***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 II.7[12].*

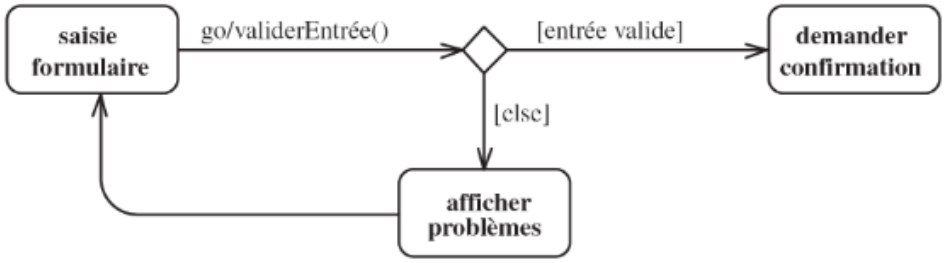


FIGURE II.7 – Point de choix [24].

• ***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 II.8 [25]*

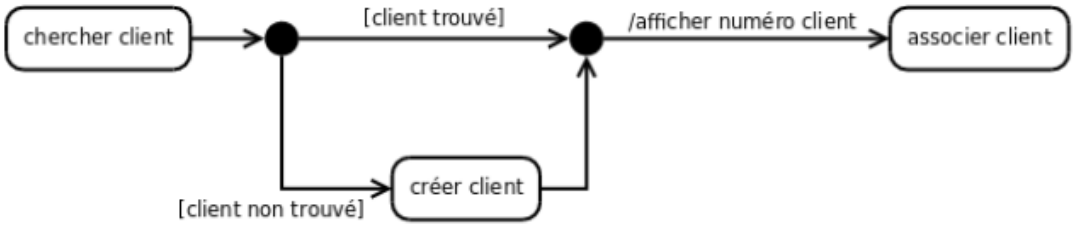


FIGURE II.8 – Exemple d'état-transition avec point de jonction [25].

• ***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] [26].*

• ***La transition complexe :*** *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.*

Si elle possède plusieurs états source, elle représente une jointure de contrôle.

Si elle possède plusieurs états cibles, elle représente un débranchement de contrôle.

Si elle dispose plusieurs états source et cible, elle représente une synchronisation de threads parallèles.

La transition complexe est représentée graphiquement par une barre épaisse et courte [12].

• ***La concurrence [11] :*** *les diagrammes d'états-transitions permettent de décrire la concurrence d'une manière très efficace grâce à l'utilisation des états orthogonaux (chaque région de l'état orthogonal représente un flot d'exécution). Graphiquement, dans un état orthogonal, les différentes régions sont séparées par un trait horizontal en pointillé dans l'état composite.*

Chaque région peut posséder un état initial et un état final. Une transition qui atteint la limite d'un état composite orthogonal est équivalente à une transition qui atteint les états initiaux de toutes ses régions concurrentes.

Pour que l'état composite soit considéré comme terminé, toutes les régions concurrentes doivent atteindre leurs états finaux.

La figure II.9 montre un exemple de concurrence dans un état composite orthogonal. La préparation de boissons dans un distributeur se fait en parallèle au rendu de monnaie.

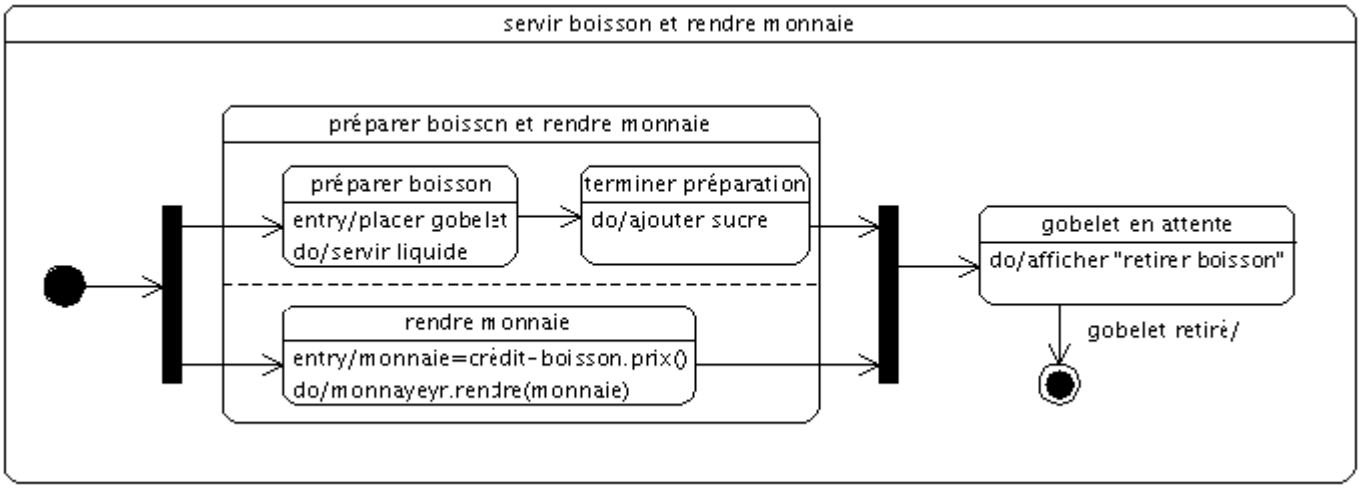


FIGURE II.9 – Exemple d'utilisation de transitions complexes et concurrence [25].

II.7.4 Intérêts des diagrammes d'états- transitions :

Les intérêts de la modélisation par les diagrammes d'états transitions sont [12] :

- *Donner vie aux objets (sils s'y prêtent), représentés jusqu'à présent de manière statique comme des occurrences de classes qu'on peut généraliser par les classes.*
- *Visualiser le système en diminuant sa complexité.*
- *Tenir compte des états lors de l'implémentation (en effet la traduction des états peut être faite simplement, la plupart de langages le permettent).*
- *Présenter un aspect du modèle dynamique, l'autre étant illustré par les diagrammes d'activités, les diagrammes de collaboration et les diagrammes de séquence.*

II.8 Conclusion :

Dans ce chapitre, nous avons essayé de fournir une introduction au langage de la modélisation UML, en commençant par la présentation du formulaire dans un cadre général, puis le journal, UML et ses diagrammes , puis nous avons donné une description détaillée des diagramme d'états transitions UML 2.0 qui constitue des composants spéciaux de ce graphique. Nous avons souligné les éléments du diagramme d'états transitions (état, transition, évènement), qui seront formalisés dans la source de transformation qui fait l'objet de notre travail et qui seront cités au chapitre 4.

Chapitre III

Méthodes formelles et modèle RPTT

III.1 Introduction :

Les méthodes formelles sont de plus en plus utilisées pour répondre aux exigences des systèmes critiques. Elles offrent plusieurs approches : l'approche comportementale, l'approche logique et l'approche test. L'un des intérêts majeurs des réseaux petri réside dans leur faculté d'offrir des méthodes formelles d'analyse, de validation et d'évaluation de performances. Ces méthodes formelles donnent souvent des résultats globaux sur les modèles étudiés.

Dans la suite de ce chapitre, nous essayons d'introduire une notion principale dans l'ingénierie dirigée par les modèles et plus particulièrement au niveau de la transformation : La vérification ensuite nous décrivons l'état relatif à l'introduction des méthodes formelles pour cette notion où nous précisons, les techniques proposées pour la vérification dans la transformation de modèles ensuite, nous positionnons notre approche de transformation de modèles par rapport aux travaux existants, ensuite nous mettons l'accent sur les réseaux de Petri, une technique formelle couramment utilisée.

III.2 pourquoi utiliser les méthodes formelles ?

Les méthodes formelles sont une sorte de techniques permettant de raisonner rigoureuse ment sur des programmes informatiques afin de démontrer leurs conformités (corrections) en se basant sur des raisonnements de logique mathématique. En pratique, une méthode formelle est basée sur une notation formelle permettant d'atteindre des exigences de qualité élevées du système à modéliser [15].

III.3 Méthodes Formelles :

Une méthode formelle permet à offrir un cadre mathématique permettant de décrire d'une manière précise et stricte les programmes que nous voulons construire. Ce cadre formel vise d'éliminer

les ambiguïtés existant au niveau du cahier des charges et du langage naturel [15].

Les méthodes formelles (MFs) sont des techniques basées sur les mathématiques pour décrire des propriétés de systèmes. Elles fournissent un cadre très rigoureux pour spécifier, développer et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc, afin de démontrer leur validité par rapport à une certaine spécification.

Ces méthodes permettent d'obtenir une très forte assurance de l'absence des incohérences, des contradictions et des failles dans la conception aussi bien qu'à déterminer la correction de l'implantation d'un système. Cependant, elles sont généralement coûteuses en ressources (en termes de temps et d'argent) et réservées aux systèmes critiques. Leur instrumentation et outillage sont la motivation de nombreuses recherches pour élargir leurs champs d'application [16].

III.4 Langage formelle :

Les méthodes formelles reposent sur l'utilisation de langages formels pour donner une spécification du système que l'on souhaite développer à un niveau de détail désiré.

Un langage formel est en effet un langage doté d'une sémantique mathématique adéquate basée sur des règles d'interprétation et des règles de déduction Les règles d'interprétation garantissent l'absence d'ambiguïté dans les descriptions produites, contrairement à des descriptions en langage informel ou semi-formel qui peuvent donner lieu à différentes interprétations. Alors que les règles de déduction permettent de raisonner sur les spécifications afin de découvrir de potentielles incomplétudes, inconsistances ou pour prouver des propriétés attendues [1].

III.5 Techniques d'analyse :

Il existe plusieurs méthodes d'analyse des systèmes complexes, elles permettent de garantir leur bon fonctionnement ou alors limiter les risques liés à leur performance, Parmi ces méthodes nous citons :

III.5.1 La vérification :

*La vérification comprend l'ensemble des activités d'inspection, de test, de simulation, de preuve automatique ainsi que les autres techniques qui permettent d'affirmer ou non la question «Le système a-t-il été conçu correctement ?». Elle est définie selon l'ISO comme "*la confirmation par examen et apport de preuves tangibles (informations dont la véracité peut être démontrée, fondée sur des faits obtenus par observation, mesures, essais ou autres moyens) que les exigences spécifiées ont été satisfaites *" [5].*

III.5.2 La validation :

*La validation compare le système réalisé aux besoins exprimés par ses utilisateurs. Cette tâche permet d'évaluer le système développé en répondant à la question : «Sommes-nous en train de développer le bon système ?». L'ISO définit la validation comme "*La confirmation, par examen et apport de preuves tangibles, que les exigences particulières pour un usage spécifique prévu sont satisfaites. Plusieurs validations peuvent être effectuées s'il y a différents usages prévus*" [5].*

III.5.3 La qualification :

La qualification assure que le modèle peut être utilisé dans la communication sans ambigüité d'interprétation entre les activités du projet et les acteurs. [5].

III.5.4 La certification :

La certification nécessite l'intervention d'un organisme indépendant qui assurera le respect des normes par le modèle, et qui reconnaîtra sa pertinence, sa rigueur et ses qualités pour une éventuelle réutilisation, voire pour l'établissement d'un référentiel générique à son domaine [5].

III.6 Classification des Méthodes Formelles :

Dans la littérature, il existe plusieurs classifications des méthodes formelles. Selon J.M. Wing, on peut distinguer les méthodes [1] :
- Orientées opérations pour décrire le fonctionnement du système et son comportement par des axiomes.
- Orientées données pour décrire les états du système .
- Hybrides en combinant les deux orientations.

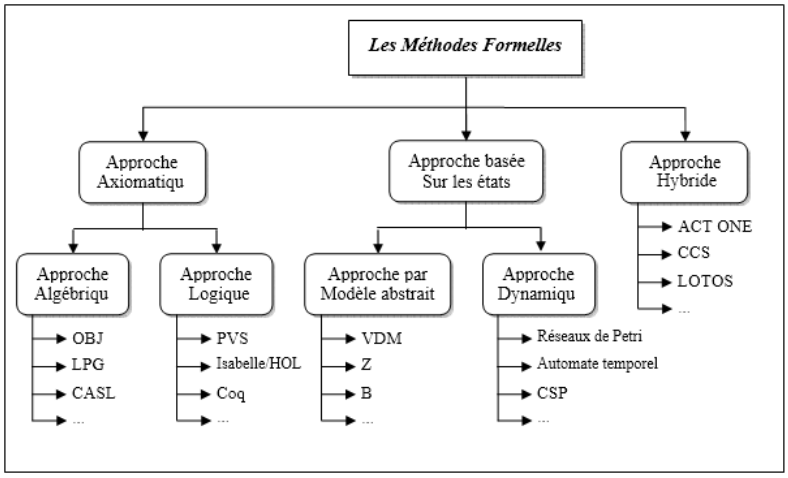


FIGURE III.1 – La classification des méthodes formelles [7].

III.6.1 L'approche axiomatique :

Dans cette approche, on s'intéresse au comportement du système de manière indirecte en construisant une axiomatisation qui fournit les propriétés comportementales du système traité en utilisant une approche algébrique ou une approche logique [31].

L'approche algébrique permet de définir des types abstraits de données en spécifiant pour chaque opération les types de valeurs de ses paramètres et du résultat, en précisant une expression construite à l'aide des variables appelées termes, et en décrivant les propriétés des opérations sous forme d'équivalences entre termes (les axiomes). L'application des axiomes à des termes permet d'obtenir d'autres expressions d'équivalence. Parmi les langages de spécification algébrique connus nous pouvons citer : OBJ, LPG et Larch.

L'approche logique, quant à elle, permet d'exprimer des systèmes transformationnels en utilisant la logique temporelle pour exprimer des propriétés dynamiques de sûreté et vivacité des systèmes réactifs. Dans cette approche on s'intéresse à la démonstration de programmes en appliquant les théories de la démonstration automatique ou semi-automatique de théorèmes. Parmi les langages de spécification logiques connus nous pouvons citer : PVS, Isabelle/HOL et Coq [32].

III.6.2 L'approche basée sur les états :

Dans l'approche basée sur les états, on s'intéresse aux données du système en construisant un modèle du système en termes de structures mathématiques. Ce modèle doit avoir les propriétés du système. On peut ensuite raisonner sur le fonctionnement du système en utilisant le modèle obtenue. Dans ce sens, les approches ensemblistes et les approches dynamiques sont utilisées.

Les approches ensemblistes, appelées aussi approches par modèle abstrait, permettent de fournir une syntaxe et une sémantique du modèle abstrait en se basant sur la théorie des ensembles, logique du premier ordre, ou la théorie des types. Les approches ensemblistes différent des approches algébriques par l'utilisation des types abstraits prédéfinis pour modéliser l'état du système à construire. Chaque opération est spécifiée indépendamment en décrivant son effet sur l'état du système. Parmi les langages de spécification ensemblistes connus nous pouvons citer : VDM, Z et B.

Les approches dynamiques se basent sur la notion de processus pour spécifier les systèmes de transitions en utilisant les automates, les réseaux de Petri et les algèbres de processus comme CSP [4].

III.6.3 L'approche hybride :

L'approche hybride complète une axiomatisation par un modèle de données ou bien l'inverse. Par exemple, LOTOS est considéré comme un langage algébrique car ses expressions de contrôle sont caractérisées par une algèbre dont les termes sont des processus. Alors que les données sont

caractérisées par une algèbre de types abstraits dont les termes sont des expressions fonctionnelles. Le langage CCS et le langage ACT ONE sont les prédécesseurs de LOTOS [5].

Généralement, l'approche basée sur les états est l'approche la plus utilisée pour vérifier les aspects dynamiques d'un système car cette approche permet un raisonnement sur le fonctionnement d'un système. Spécialement, les réseaux de Petri constituent le langage le plus utilisé parce qu'ils combinent les avantages de la représentation graphique avec la sémantique formelle attribuée au comportement des systèmes. Pour cette raison, nous allons détailler les réseaux de Petri dans la section 8.

III.7 L'intégration des méthodes formelles dans l'IDM :

L'utilisation des méthodes formelles et de plus en plus courantes dans le développement des systèmes depuis que la complexité de ces derniers a évolué, surtout dans le cas des systèmes critiques, là où aucune erreur n'est tolérée.

Depuis que l'IDM est devenue une démarche de conception qui couvre le cycle de développement dans son intégralité, des études ont démontré que cette approche peut être combinée avec les méthodes formelles, en minimisant les inconvénients de chacune et tirant profit des avantages des deux [2].

III.7.1 Les avantages des méthodes formelles :

Il est aujourd'hui trivial pour un développeur de passer par les méthodes formelles pour la validation des systèmes critiques. Un modèle formel sert de support pour l'analyse formelle qui assurera ou non sa validité par rapport aux exigences du client, par simulation ou par tests. Cette analyse formelle garantira certaines propriétés comportementales [2].

III.7.2 Les inconvénients de l'IDM :

Pour le moment, les environnements de méta-modélisation manquent de supports rigoureux qui permettraient de fournir la sémantique des méta-modèles. La sémantique est exprimée en langage naturel, ce qui empêche l'analyse formelle des modèles du langage défini par la méta-modélisation. L'IDM permet, grâce à la méta-modélisation et aux transformations de modèles, de mettre en place des plateformes pour les méthodes formelles et de conserver leur interopérabilité [2].

III.8 La vérification dans l'IDM :

Les notions de vérification a été définie à la fin des années 70 dans le contexte des sciences expérimentales en général et plus précisément dans le génie logiciel. De temps en temps, cette

définition a été progressivement précisée, depuis l'émergence de l'IDM, on assiste à adapter cette notion, pour détecter et corriger les erreurs en plus pour garantir une bonne production du logiciel, ainsi, les chercheurs se sont intéressés à la prise en charge de cette notion au niveau de la transformation de modèles et qui consiste d'abord à déterminer s'il existe des techniques de vérification pour faciliter à offrir des transformations fiables, correctes et prêtes à utiliser dans le contexte de l'IDM.

La première technique proposée pour résoudre cette problématique est le "Test". Cette dernière est une méthode classique de vérification par l'exécution d'une partie ou de la totalité du code implémenté. Le but du test de logiciel est de détecter les erreurs des programmes le plus tôt possible. L'inconvénient majeur de cette technique est l'absence de raisonnement mathématique sur les propriétés de la spécification, car ces techniques reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise et Puisque les techniques formelles se basent sur les aspects mathématiques, les chercheurs proposent de les intégrer dans l'IDM.

III.8.1 La vérification dans la transformation de modèle :

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

Vérification : *c'est une méthodologie assurant que le modèle est conforme à sa spécification. Elle cherche à répondre à la question suivante : "Construisons nous correctement le modèle ?". Elle traite soit la conformité des modèles à leurs métamodèles, ou le contrôle des règles de transformation. Dans la section suivante, nous décrivons les techniques d'application de la vérification dans la transformation de modèles [3].*

III.8.2 Techniques de vérification :

Plusieurs techniques de vérification ont été utilisées dans la transformation de modèles. Elles peuvent être classifiées en deux catégories Figure III.2. La première concerne les techniques semi-formelles, c'est-à-dire que, malgré l'utilisation des définitions rigoureuses, elles ne se basent pas sur un raisonnement mathématique, car ces méthodes reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise. La seconde regroupe les méthodes qui font appel à la logique et aux mathématiques pour effectuer cette vérification [3].

.

Dans ce qui suit, nous nous présentons les méthodes semi-formelles et les méthodes formelles,

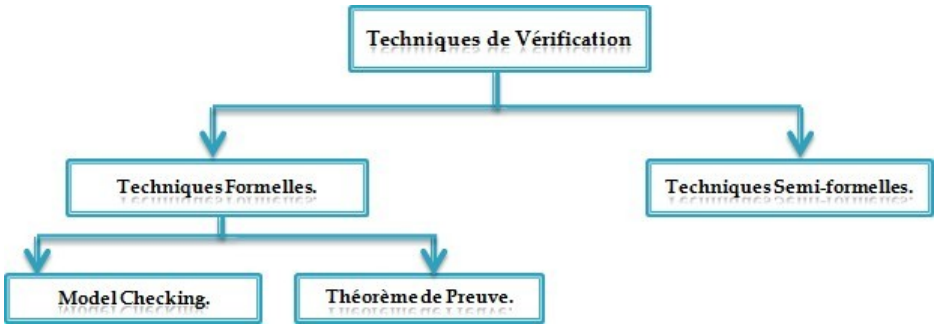


FIGURE III.2 – Les techniques de Vérification [3].

et particulièrement celles qui sont proposées pour définir la vérification de la transformation de modèles dans le cadre de l'IDM.

III.8.2.1 Techniques semi formelle :

Généralement, on trouve plusieurs techniques de vérification qui sont utilisées au sein de l'ingénierie dirigée par les modèles, notamment celles qui reposent sur les tests. Les méthodes de test permettent de détecter un grand nombre d'erreurs avec un faible coût de mise en œuvre par rapport aux techniques formelles comme la démonstration. Un test est une méthode de vérification par exécution d'une partie ou de la totalité du code implémenté. En pratique, un test est effectué en deux phases : la construction des tests et l'exécution de ces tests [3].

La vérification de la transformation de modèles par les techniques semi-formelles peut être appliquée avec différentes méthodes : le test manuel, le test automatique et la vérification par relecture. Nous les présentons dans cette sous-section. Fleurey et al. [17] proposent d'utiliser le test fonctionnel pour la transformation de modèles en se basant sur un ensemble de critères. Dans [18], Brottier et al proposent un processus et un outil pour la génération automatique de modèles de test satisfaisant les critères de test de [17].

III.8.2.2 Techniques formelles :

Les techniques de vérification formelle désignent un ensemble de méthodes basées sur les mathématiques et la logique pour assurer qu'un produit final est conforme à ses spécifications, ces dernières doivent être implémentées dans un langage défini mathématiquement (syntaxe et sémantique), Les méthodes formelles englobent deux types de méthodes pour la vérification formelle : le théorème de preuve et le modèle checking.

- ***Théorème de preuve :***

Le théorème de preuve possède des racines qui remontent à la logique mathématique, en plus c'est un moyen de construire la preuve mathématique.

La transformation de modèles dans l'IDM nécessite l'implémentation des techniques formelles pour vérifier le processus de transformation. Ces techniques se basent sur la preuve d'un ensemble de propriétés contenues dans la transformation de modèles. Comme un exemple de ces techniques, on peut citer : Coq, PVS, Isabelle dont La majorité des assistants de preuve sont basés sur un unique calcul formel, en général la logique d'ordre supérieur. Isabelle à la capacité d'accepter une variété de calculs formels. [3]. Dans [19] les auteurs utilise l'assistant de preuve Isabelle pour faire la spécification, la transformation des diagrammes UML vers réseaux pétri colorés et vérifier la transformation elle-même. Avec l'assistant Coq il est possible d'énoncer des théorèmes, des spécifications de programme, et de développer interactivement leurs preuves à l'aide de tactiques. L'assistant PVS (Prototype Verification System) est un outil de vérification intégrant le langage de spécification PVS qui se base sur la logique classique d'ordre supérieur.

L'avantage de cette technique par rapport aux autres méthodes réside dans sa prise en compte des données complexes, ce qui lui permet d'être appliquée sur des espaces d'états non-finis. L'inconvénient majeur est que la preuve de théorème reste une technique plus ou moins lente [5].

- ***Modèle cheking :***

Le modèle checking est une autre méthode de vérification formelle, elle se base sur la construction de toutes les situations possibles d'un modèle, ensuite de les vérifier automatiquement sans interaction avec l'utilisateur, elle permet de vérifier si un modèle du système satisfait une propriété souhaitée d'une façon automatique, l'avantage principal de cette méthode est que la vérification de propriétés est complètement automatisée. Parmi ses différents outils, on cite : SPIN, et UPPAAL.

La preuve de théorème demande une grande interaction avec l'utilisateur ce qui rendrait la technique moins automatique. La technique du model checking comme méthode formelle est une méthode très performante et très utilisée. Cette technique correspond au mieux à notre domaine, car elle pourra révéler des erreurs non détectées par les autres méthodes formelles comme le test et la simulation. Au contraire du model-checking, le theorem proving peut s'utiliser avec des espaces d'états infinis à l'aide de techniques comme l'induction structurelle [3].

III.9 Les réseaux de Pétri :

Les systèmes dynamiques sont caractérisés par leur comportement permanent. Ce comportement est décrit par une séquence d'états qui peut être infinie. Pour cette raison, ces systèmes ne peuvent être décrits que sur la base de leurs états initiaux et états finaux.

Différents paradigmes pour la description des systèmes dynamiques ont été développés mais ils sont peu nombreux à intégrer des méthodes d'analyse dans leur propre description, comme c'est le cas des réseaux de Petri.

Les réseaux de Petri constituent un outil graphique et mathématique qui permet de simuler et modéliser des systèmes dans lesquels la notion d'événements et d'évolution sont importants, ils jouent un rôle important car ils sont capables de modéliser des propriétés telles que synchronisation, parallélisme, conflits, mutuelle exclusion et partage de ressources.

Grâce à leur généralité et à leur souplesse, les réseaux de Petri sont utilisés dans une large variété de domaines tels que les protocoles de communication, les systèmes distribués, l'architecture des ordinateurs, etc [28].

III.9.1 Historique :

Les réseaux de Petri ont été introduits par Carl Adam Petri en 1962 dans sa thèse de doctorat, intitulée « Communication avec des Automates », en Allemagne. Ce travail a ouvert un champ d'étude et a été exploité par Anatol W. Holt, F. Commoner, M. Hack et leurs collègues dans le groupe de recherche de Massachussetts Institute OF Technology (MIT) dans les années 70. Le premier livre traitant des réseaux de Petri a été publié par J. Peterson [11].

III.9.2 Bases et définitions des réseaux de Petri :

Définition Graphique [20] :

Un réseau de Petri (RDP) est un graphe biparti orienté valué. Il a deux types de nœuds :

1. les places : *notées graphiquement par des cercles. Chaque place contient un nombre entier (positif ou nul) de marques (ou jetons). Ces derniers sont représentés par des points noirs.*

2. les transitions : *notées graphiquement par un rectangle ou une barre. Une transition qui n'a pas de place en entrée est appelée transition source et une transition qui n'a pas de place en sortie est appelée transition puits.*
Les places et les transitions sont reliées par des arcs orientés où :

- Un arc relie, soit une place à une transition, soit une transition à une place mais jamais une place à une place ou une transition à une transition.

- Chaque arc est étiqueté par une valeur (ou un poids), qui est un nombre entier positif. L(arc ayant k poids peut être interprété comme un ensemble de k arcs parallèles. Un arc qui n'a pas d'étiquette est un arc dont le poids est égal à 1.

La figure III.3 illustre la notation graphique d'un Réseau de Petri.

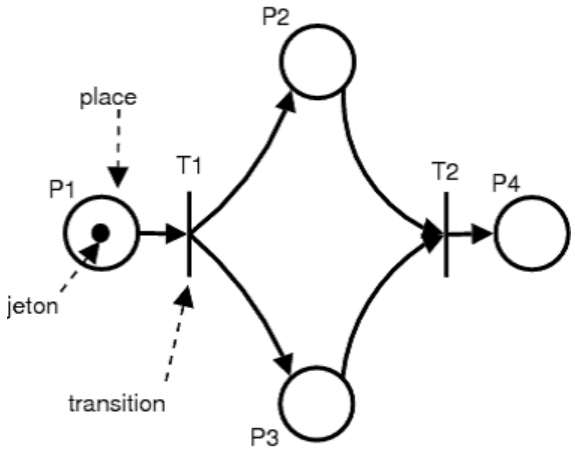


FIGURE III.3 – Exemple de réseau de Petri marqué [5].

Définition formelle [21] :

Un RdP est un quadruplet R = (P, T, Pre , Post) où

- *P est un ensemble fini de places .*
- *T est un ensemble fini de transitions .*
- *Pre est une application de $P \times T \to N$.*
- *Post est une application de $T \times P \to N$.*
- *A partir de P et T, on introduit la matrice $C = Post^t - Pre$.*
- *Un marquage est une application $M : P \to N$.*
- *Une transition t est franchissable si*

$$\forall p \in P, M(p) \geq Pre(p,t)$$

On note parfois $M(t >$ ou encore $M \to^t)$

• *Si t est franchissable, pour le marquage M, le franchissement (tir) de la transition t donne le nouveau marquage M^t*

$$\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(t,p) = M(p) + C(t)$$

On note $M \to tM' ou M(t > M')$

• *Soit s une séquence sur l'alphabet des transitions, on appelle vecteur caractéristique le vecteur s formé des nombres d'occurrences de chaque transition*

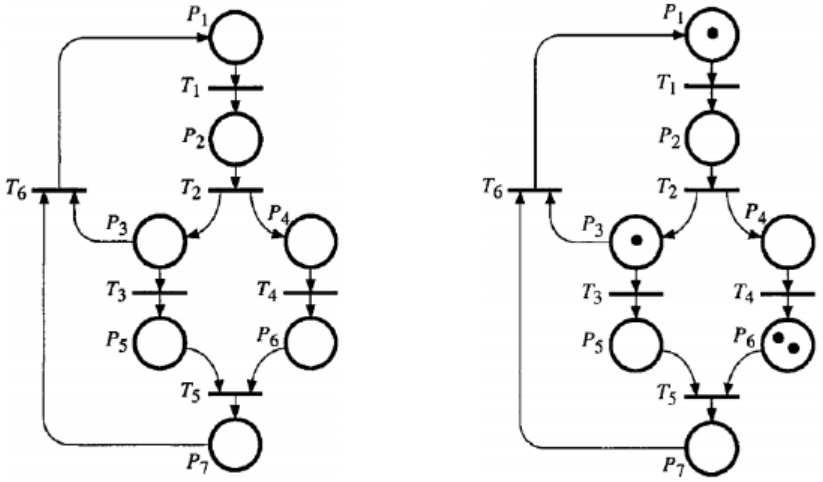
Marquage d'un Réseau de Petri [20] :

Un marquage est dénoté par un vecteur du nombre de jetons dans chaque place : la i'eme composante correspond au nombre de jetons dans la i'eme place .Le marquage de réseau de Petri représenté par la figure est donc

$$M = (1,\ 0,\ 1,\ 0,\ 0,\ 2,\ 0)$$

Un réseau de Petri N = (P, T, F,W) est un réseau de Petri sans marquage initial et un réseau de Petri Rdp avec marquage initial M0 est Rdp = (N,M0).

Le réseau de Petri représenté par la figure III.4a est un réseau de Petri non marqué alors que le réseau de Petri représenté par la figure III.4b est un réseau de Petri marqué.



(a) Réseau de Petri non marqué (b) Réseau de Petri marqué

FIGURE III.4 – Exemple de Réseau de Petri [20].

III.9.3 Évolution des réseaux de Petri :

L'évolution d'un Réseau de Petri correspond à l'évolution de son marquage au fil du temps (évolution de l'état du système) : il se traduit par un déplacement des jetons pour une transition t de l'ensemble des places d'entrée vers l'ensemble des places de sortie de cette transition. Ce déplacement s'effectue par le franchissement de la transition t selon des règles de franchissement qu'on va voir par la suite .

Transition validée :

On dit qu'une transition est validée si toutes les places en entrée de celle-ci possèdent au moins une marque. Une transition source est par définition toujours validée [20].

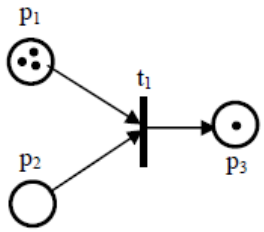


FIGURE III.5 – Transitions ne sont pas validées. [2].

Cette transition n'est pas validée, car M (p2)=0.

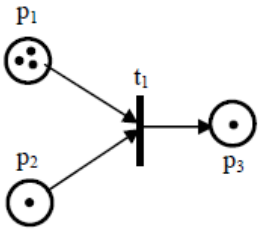


FIGURE III.6 – Transitions validées [2].

Cette transition est validée, car M (p2)=1.

Règle de Franchissement :

Si la transition est validée, on peut effectuer le franchissement de cette transition : on dit alors que la transition est franchissable. Le franchissement consiste à :

q retirer W(p, t) jetons dans chacune des places en entrée p de la transition t. q ajouter W(t, p) jetons à chacune des places en sortie p de la transition t [20].

Séquences de franchissement [5] :

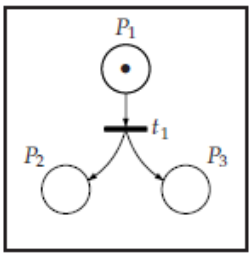


FIGURE III.7 – Exemple de transition franchissable [2].

Une séquence de franchissement S est une suite de transitions $T_i, T_j ....T_k$ qui peuvent être franchies successivement à partir d'un marquage donné. Une seule transition peut être franchie à la fois.

La notation :M_i [S -> M_j ou M_i [S > M_j exprime que le franchissement de la séquence S à partir du marquage M_i conduit au marquage M_j.

Arbre de marquage [2] :

Afin de suivre l'évolution du marquage au cours d'une séquence de tir de transitions, l'arborescence des marquages du réseau de Petri peut être représentée à partir d'un marquage initial donné.

Marquage accessible [2] :

*L'ensemble des marquages accessibles est l'ensemble des marquages Mi qui peuvent être atteints par le franchissement d'une séquence S à partir du marquage initial M0, Noté *M0.*

Pour un RdP et à partir d'un marquage initial M0, un marquage Mq est dit accessible si et seulement si : Il existe S / : $M0 \xrightarrow{S} Mq$

III.9.4 Propriétés des réseaux de Petri :

III.9.4.1 Propriétés génériques :

La vivacité, le caractère borné et la réinitialisabilité sont les principales propriétés génériques qui peuvent être automatiquement vérifiées dans un réseau de Petri.

- ***Vivacité :***

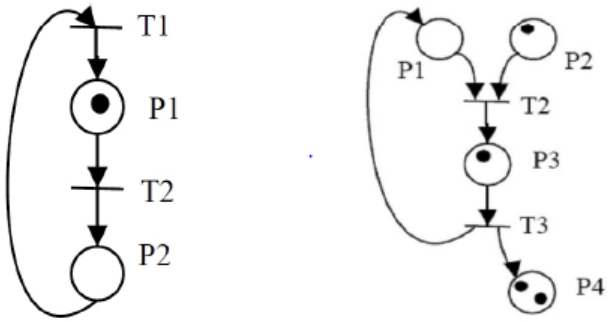
Il s'agit de vérifier si à tout instant, le système conserve la possibilité de reproduire toutes les transitions.

Transition vivante : *une transition $t \in T$ est vivante si pour tout marquage accessible M,*

il existe une séquence de transitions s qui contienne la transition t à partir de M. Autrement dit, quelle que soit l'évolution du système, il existera toujours une possibilité de franchir t.

La vérification de la vivacité d'une transition revient à vérifier qu'elle apparaît dans toutes les composantes fortement connexes du graphe des marquages.

Réseau vivant : un réseau de Petri est vivant si et seulement si pour toute transition $t \in T$, t est vivante [22].



(a) Rdp vivant (b) Rdp non vivant

FIGURE III.8 – Exemple de vivacité des Réseau de Petri [20].

- ***Blocage [20] :***

Un marquage M d'un réseau (N, M_0) est appelé marquage "puits" si aucune transition n'est franchissable depuis M.

Un réseau est dit sans blocage si tout marquage accessible depuis M_0 n'est pas un marquage "puits".

Le réseau de Petri marqué représenté par la figure III.8b a pour blocage le marquage :

$$M_3 = [1,\ 0,\ 0,\ 4]$$

- ***Réseau borné, Réseau Sauf :***

Un réseau de PetriRdp $= (N, M_0)$ est K-borné ou simplement borné si le nombre de jetons dans chaque place ne dépasse pas un nombre fini K pour tout marquage accessible depuis M_0 C'est-à-dire :

$$\forall M \in R(M_0), \forall p \in P : M(p) \leq k$$

Un réseau de Petri marqué Rdp $= (N, M_0)$ est sauf si et seulement s'il est 1-borné [20].

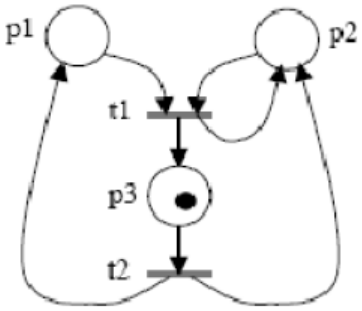


FIGURE III.9 – Exemple de blocage des Réseau de Petri [2].

- ***Conflit [23] :***

Conflit structurel :

Deux transitions t_1 et t_2 sont en conflit structurel si et seulement si elles ont au moins une place d'entrée en commun :

$$\exists p Pr(p, t_1) Pr(p, t_2) f = 0$$

Conflit effectif :

Deux transitions sont en conflit effectif pour un marquage M si et seulement si t_1 et t_2 sont en conflit structurel et que :

$$M \geq Pr(p, t_1)$$

$$M \geq Pr(p, t_2)$$

- ***Réinitialisabilité :***

Un réseau de Petri est réinitialisable si et seulement si pour tout marquage M, il existe une séquence de transitions qui permet de revenir au marquage initial M_0.

Cette propriété renseigne sur le fonctionnement répétitif, ce qui est pertinent pour la majorité des systèmes interactifs [5].

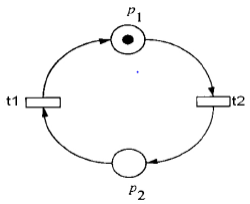


FIGURE III.10 – Exemple d'un réseau de Petri réinitialisable [20].

- ***Parallélisme :***

Dans le réseau de Petri représenté par la figure III.11 le franchissement de la transition T1 met un jeton dans la place P2 (ce qui marque le déclenchement du processus 1) et un jeton dans la place P2 (ce qui marque le déclenchement du processus 2) [20].

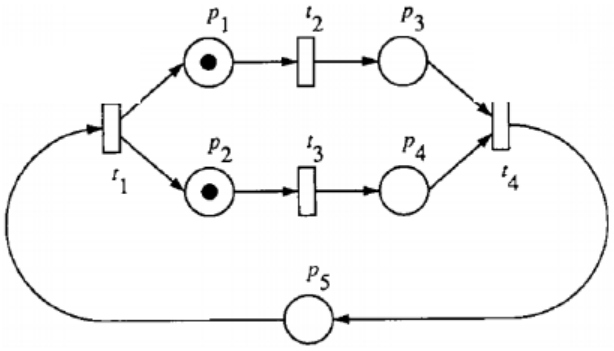


FIGURE III.11 – Parallélisme dans les Réseaux de Petri [20].

III.9.4.2 Propriétés spécifiques :

- ***Accessibilité :***

L'accessibilité est une propriété fondamentale pour étudier les propriétés dynamiques du système. Le franchissement d'une transition validée va changer la distribution des jetons sur le réseau selon les règles définies dans la section 2.1.4.2. Une séquence de franchissements σ entraîne une séquence de marquages.

Un marquage M_n est dit accessible à partir de M_0, s'il existe une séquence de franchissements σ permettant de transformer M_0 à Mn. Une séquence de franchissements est notée par

$$\sigma = M_0 t_1 M_1 t_2 M_2 .....t_n M_n$$

Ou simplement :

$$\sigma = t_1 t_2 .....t_n$$

Dans ce casM_n est accessible à partir de M_0 par σ et on écrit :

$$M_0[\sigma] > M_n$$

L'ensemble de séquences de franchissements à partir de M_0 dans un réseau (N, M_0) est noté par $L(N, M_0)$ ou simplement $L(M_0)$.

L'ensemble de marquages possibles accessibles à partir de M_0 dans un réseau (N, M_0) est noté par $R(N, M_0)$ ou simplement $R(M_0)$ [4].

- ***Couverture :***

Un marquage M dans un réseau de Petri (N, M_0) est dit ouvrable s'il existe un marquage M' dans $R(M_0)$ tel que $M'(p) \geq M(p)$ pour chaque place du réseau [20].

- ***Persistance :***

Un réseau de Petri (N, M_0) est dit persistant si pour n'importe quelles deux transitions, le franchissement d'une transition ne doit pas inhiber l'autre transition. Si une transition dans un réseau de Petri persistant est une fois validée, elle reste validée jusqu'à son franchissement [20].
La figure III.12 représente un réseau de Petri persistant.

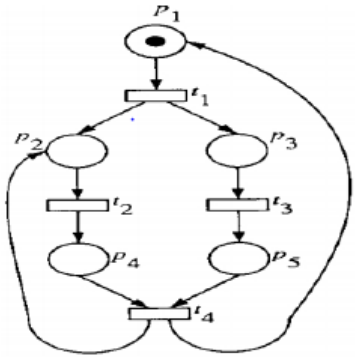


FIGURE III.12 – Exemple d'un réseau de Petri persistant [20] .

III.9.4.3 Graphe de marquage :

Le graphe des marquages accessibles de N depuis M_0, noté $G(N, M_0)$ est le graphe dont les états sont les marquages accessibles tel qu'il existe un arc entre deux sommets M_1 et M_2 si et seulement si $M_1[t > M_2$.
La figure III.13 montre un exemple du graphe des marquages du réseau de Petri (a) [11].

III.9.5 Modélisation des systèmes concurrents :

L'avantage des réseaux de Petri réside dans leur capacité à modéliser un grand nombre de comportements dans les systèmes complexes. Parmi ces comportements, nous trouvons la synchronisation, le partage de ressources, la mémorisation, la lecture d'informations, la limitation de capacité de stockage, etc [5].

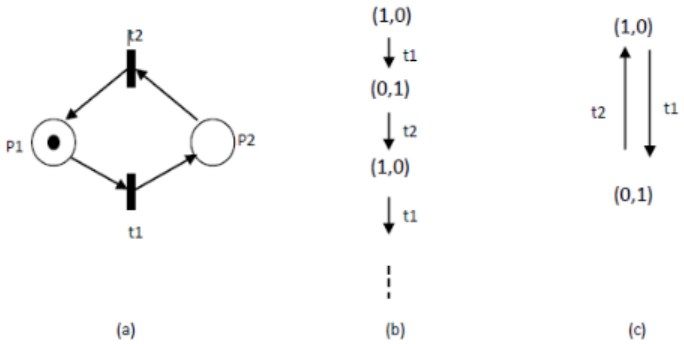


FIGURE III.13 – Graphe des marquages du réseau de Petri (a) [11] .

- ***La synchronisation [5] :***

Il existe deux types de synchronisation : la synchronisation mutuelle (rendez-vous) et la synchronisation par signal (sémaphores).

- *Synchronisation mutuelle : cette synchronisation permet de synchroniser les opérations de deux processus comme le montre la figure III.14.*

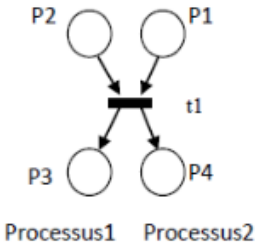


FIGURE III.14 – Réseau de Petri avec synchronisation mutuelle [5].

Pour que la transition t_1 franchisse, il faut que la place P1 qui correspond au processus1 et la place P2 qui correspond au processus 2 contiennent chacune au moins un jeton. Autrement, si par exemple la place P1 ne contient pas de jetons, le processus2 reste bloqué sur la place P2 ; il attend que le processus 1 réussisse à obtenir un jeton dans la place p1 au cours de son évolution.

- *Synchronisation par signal : les opérations du processus 2 se poursuivent à condition que le processus 1 ait atteint un certain niveau dans son évolution. Ceci n'est pas le cas du processus 1 qui ne dépend pas de l'avancement des opérations du processus 2. Un exemple est illustré dans la figure III.15.*

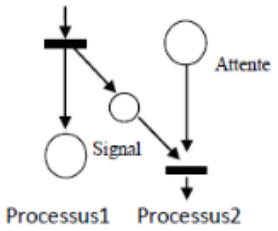


FIGURE III.15 – Réseau de Petri avec synchronisation par signal [5].

Lorsque la place "Signal" est marquée et la place "Attente" ne l'est pas, ceci se traduit par un processus 1 qui a envoyé le signal que le processus 2 n'a pas encore reçu. Si, à l'inverse, la place "Signal" n'est pas marquée et la place "Attente" est marquée, cela signifie que le processus 2 est en attente du signal.

- ***Le partage de ressources [5] :***

Il s'agit de modéliser le cas de plusieurs processus se partageant une même ressource dans un même système, en utilisant l'exclusion mutuelle.

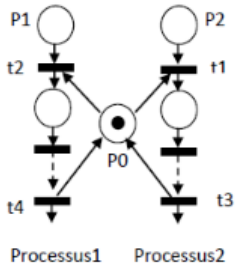


FIGURE III.16 – Réseau de Petri avec partage de ressource [5].

Dans la figure III.16, le jeton dans la place P0 représente une ressource commune entre le processus 1 et le processus 2. Après le franchissement de la transition t_1 lors de l'évolution du processus 1, le jeton de la place P0 est alors consommé. La ressource que constitue ce jeton n'est alors plus disponible pour l'évolution du processus 2. Lors du franchissement de la transition t_4, un jeton est placé dans la place P0, et la ressource est de nouveau disponible.

- ***La mémorisation [5] :***

Une place sans entrée ou sans sortie peut être utilisée dans tous les modèles pour compter le nombre de tirs d'une transition. Dans la figure III.17, les places "Attente" et "Compteur" peuvent indiquer combien d'instances d'un processus sont en attente.

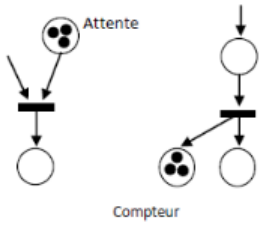


FIGURE III.17 – Réseau de Petri illustrant le cas de la mémorisation [5].

III.9.6 Les Réseaux de Petri de Haut Niveau :

Pour l'utilisation des réseaux de Petri dans la modélisation des systèmes réels, plusieurs auteurs ont trouvé qu'il est convenable d'étendre le formalisme de réseau de Petri pour compacter la représentation de modèle ou pour étendre le pouvoir de modélisation du formalisme de réseau de Petri. Ce qui a donné naissance aux réseaux de Petri de haut niveau [7].

a) ***Réseaux de Petri colorés :***

Dans un Rdp coloré, les marques peuvent être différenciées par des couleurs. Si le nombre de couleurs est fini, alors en peut se ramener à un Rdp classique.

Le Rdp coloré est quintuplé R = (P, T, Pré, Post, C) Ou C= C_1,C_2, ..,.C_i,....C_k = Ensemble des couleurs avec $C_i = Ci, Ci, Ci, \ldots\ldots\ldots C^*$.

La coloration des jetons permet de différencier les produits circulant dans Rdp. La couleur peut être s'agit un attribue unique, soit un n-uplet d'attributs formant ainsi un agrégat [28].

b) ***Réseau de Petri Objet [7] :***

Les réseaux de Petri Objet (OPN) étendent le formalisme des réseaux de Petri colorés avec une intégration complète des propriétés orientées objet y compris l'héritage, le polymorphisme et la liaison dynamique. L'orientation objet fournit des primitives de structuration puissante permettant la modélisation des systèmes complexes.

III.10 des réseaux de Petri temporellement temporisés PRTT :

III.10.1 Le principe :

L'idée fondatrice des réseaux de Petri temporellement temporisés (pour Duration Action Time Petri Nets, en anglais) est d'associer deux dates min et max à chaque transition, c'est l'intervalle de tir de cette transition. Cet intervalle représente une latence durant laquelle la transition peut être tirée. Quoique le tir de la transition soit instantané, la durée d'exécution de l'action associée à cette transition peut avoir une durée non nulle. A titre d'exemple, soit une transition « t » dont l'action associée est de durée d ; cette transition a été sensibilisée à la date « θ », alors « t » ne peut être tirée avant la date « θ + min » et doit être tirée au plus tard à la date « θ + max », à moins que « t » ne soit désensibilisée par le tir d'une autre transition. Dans le cas où cette transition est tirée, l'action associée commence son exécution et elle dure d'unités du temps.

Soit le tir de « t » à la date ϑ avecθ + min ≤ ϑ ≤ θ + max, l'action associée se termine à la date θ + d Le tir de la transition marque le début d'exécution de l'action associée.

Dans un réseau de Petri temporellement temporisé le passage des jetons de l'état indisponible à l'état disponible est conditionné par l'écoulement de la durée de l'exécution de l'action associée (respectivement par le tir de la transition). Un jeton déposé dans une place p (parties droites) à la date ϑ passe de l'état indisponible à l'état disponible à la date ϑ + d, le jeton est lié au tir de la transition durant l'intervalle [ϑ, ϑ + d] et il devient libre à l'instant ϑ + d (devient dans la partie droite de cette place).

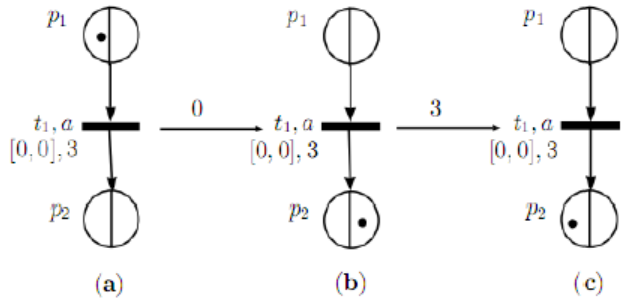


FIGURE III.18 – Le passage d'un état à un autre pour le jeton [29].

Le jeton qui se trouve dans la place amont de la transition n'est lié à aucune transition, ce jeton est libre dans cet état-là. Dans le cas où la transition se tire, l'action associée au tir de cette transition a commencé son exécution, ce qui est marqué par la présence du jeton dans la place aval figure (III.18) (C1), De ce fait, le jeton dans cette place est lié au tir de la transition, mais après l'achèvement de a, après 3 unités du temps, le jeton deviendra libre figure(III.18)[29].

III.10.2 La dépendance causale des transitions :

Si une transition « t' » dépend causalement de la transition « t », « t' » ne sera sensibilisée qu'après l'achèvement de l'action associée à « t », c'est-à-dire après les d unités du temps qui suivent le moment de tir de la transition « t ». Pour capturer cette notion de dépendance causale entre les tirs des transitions, les jetons produits par le tir d'une transition sont dits liés à cette transition et ce tout au long de la durée de l'exécution de l'action associée [4].

III.10.3 Définition formel :

Soit T un domaine temporel un RPTT sur T et de support, est un tuple tel que : $N = (P, T, F, B, \lambda, SIM, \Gamma)$ *tel que :*

- $N = (P, T, F, B)$*est un RdP marqué.*
- *Un alphabet Act est un ensemble fini; nous supposons que* $\tau \notin Act$*(*τ *désignera l'action invisible, dite aussi action silencieuse ou interne).*
- *L'étiquetage d'un RPTT est une fonction* $\lambda : T \to Act \cup (\tau)$ *Si* $\lambda(\tau) \in Act$ *alors* τ *dite observable ou externe ; dans le cas contraire,t dite silencieuse ou invisible.*
- $SIM = [\mathbb{T}] \times [\mathbb{T}]$ *est la fonction qui associe a chaque transition un intervalle statique de tir.*
- $\Gamma : Act \to D$ *est la fonction de durée statique, qui à chaque action associe sa durée statique.*
- • *Soit I l'ensemble de touts les intervalles d'un RPTT tel que :*
  I(t) = [min,max] est l'intervalle associé à la transitiontet on note par :

  $\downarrow I(t) = min, \uparrow I(t) = max$ *les fonctions qui donnent respectivement la borne inférieure et la borne supérieure d'un intervalle [30].*

III.10.4 Des sous-classes du RPTT :

Il est clair que le fait d'associer deux dates min et max à chaque transition (dates qui représentent une latence) avec une durée fixe de l'action associée, donne une intuition que ces réseaux sont une extension native des réseaux de pétri t-temporisés, dans un contexte d'une sémantique qui oblige le tir des transitions (par notion de sémantique forte). Donc on peut décrire n'importe quel réseau t-temporisé par un RPTT. Le fait d'associer l'intervalle $[0, +\infty[$*à toute transition du réseau de pétri t-temporisé permet de le voir comme un RPTT. Etant donné que les deux extensions temporisées des réseaux de pétri sont équivalentes (Réseaux de pétri t-temporisé et p-temporisés), les RPTTs sont encore une généralisation des réseaux de Petri p-temporisés.*

Le fait que toutes les actions d'un RPTT sont instantanées, ce dernier est vu comme un réseau de Petri temporel (T-TPN). Donc les réseaux de Petri temporels (T-TPNs) sont simulés par des RPTTs, où toutes les actions sont de durées nulles. Donc on peut déduire un résultat important, qui est la capacité des RPTTs de modéliser les chiens de gardes. En notant que les RPTTs, par définition, ne prennent pas en considération les âges des jetons .La remarque a tiré est que les RPTTs se présentent comme une généralisation de plusieurs modèles : t-temporisé, p-temporisé et T-TPNs. Une généralisation qui ne stipule pas des modifications au niveau de la structure d'un réseau, le nombre de places reste le même (respectivement le nombre de transitions) contrairement au T-TPN qui sont une généralisation des t-temporisés et p-temporisés, mais cette dernière nécessite une modification de la structure générale [30].

III.11 Conclusion :

Dans ce chapitre, nous avons rappelé l'importance de l'utilisation des techniques formelles, leurs différentes classifications en précisant dans quelle mesure elles sont pertinentes pour la spécification, la vérification et avec quel type de systèmes, nous avons donné aussi un aperçu sur la vérification de la transformation de modèles dans le cadre de l'approche IDM. Nous avons défini cette notion, plus les techniques d'application proposées, et plus particulièrement les méthodes formelles Nous avons présenté aussi les réseaux de Petri de façon générale.

À la fin de ce chapitre, nous avons parlé sur Les RPTT qui est une manière d'introduire la notion du temps dans les réseaux de Petri et nous avons également présenté le principe de ce réseau qui fait partie de notre étude.

Chapitre IV

Contribution

IV.1 Introduction :

Dans ce chapitre, nous proposons une approche pour la transformation des diagrammes d'états transitions vers les RPTT, notre approche est basée sur la transformation modèles à modèles aussi que la vérification de la transformation elle-même.

Nous commençons, par la présentation de l'outil Isabelle/HOL et notre approche. Ensuite, nous proposons notre algorithme qui permet la transformation de modèles et la vérification de la transformation elle-même et nous terminons par un cas d'étude pour illustrer notre approche.

Cette transformation est réalisée sous l'environnement l'assistant de preuve Isabelle/HOL.

IV.2 Isabelle/HOL :

Isabelle est une assistante de preuve générique. Il permet d'exprimer des formules mathématiques dans un langage formel et fournit des outils pour prouver ces formules dans un calcul logique. Isabelle a été initialement développée à l'Université de Cambridge et à la Technische Universität München, mais comprend maintenant de nombreuses contributions d'institutions et de particuliers du monde entier [34].

Isabelle / HOL est un théorème de preuve basé sur le paradigme de programmation fonctionnelle et la logique d'ordre supérieur (HOL) (Higher Order Logic), qui convient pour développer des formalisations rigoureuses. Ce fut un grand succès dans la certification des systèmes critiques et avec Isabelle / HOL, nous formalisons des systèmes, formulons des lemmes et des théorèmes sur eux, et prouvons leur justesse.

Le contexte de toute formalisation dans Isabelle / HOL est une théorie dont une théorie se

compose de : types de données, constantes, fonctions, définitions, lemmes et théorèmes.

Isabelle / HOL a deux modes : le mode de programmation et le mode de vérification. En mode de vérification, il existe deux types de méthodes de preuve : la preuve en avant et la preuve en arrière. Les preuves sont développées dans le langage ISAR, qui est un langage pour les preuves structurées [35].

L'expérience de HOL logique d'ordre supérieur est largement applicable dans de nombreux domaines des mathématiques et de l'informatique. Autrement dit la logique d'ordre supérieur est plus simple, plus fiable que la logique du premier ordre [36].

Des outils Isabelle / HOL tels qu'un simplificateur basé sur la réécriture de termes peuvent être appliqués pour prouver automatiquement un théorème simple. Et peut également prouver la terminaison de certaines fonctions récursives automatiquement, mais il ne peut pas pour d'autres fonctions et nécessite l'interaction de l'utilisateur pour compléter la preuve. Enfin, nous pouvons générer le code exécutable à partir de la spécification Isabelle / HOL dans le langage de programmation fonctionnel tel que : SCALA, SML [35].

Isabelle /HOL est un environnement interactif très complet pour la spécification, la transformation et la vérification de l'abstraction de la transformation elle-même pour garantir l'exactitude de l'algorithme de transformation, elle permet aussi une description textuelle (la description textuelle à faire manuellement) des différents diagrammes UML statiques et dynamique.

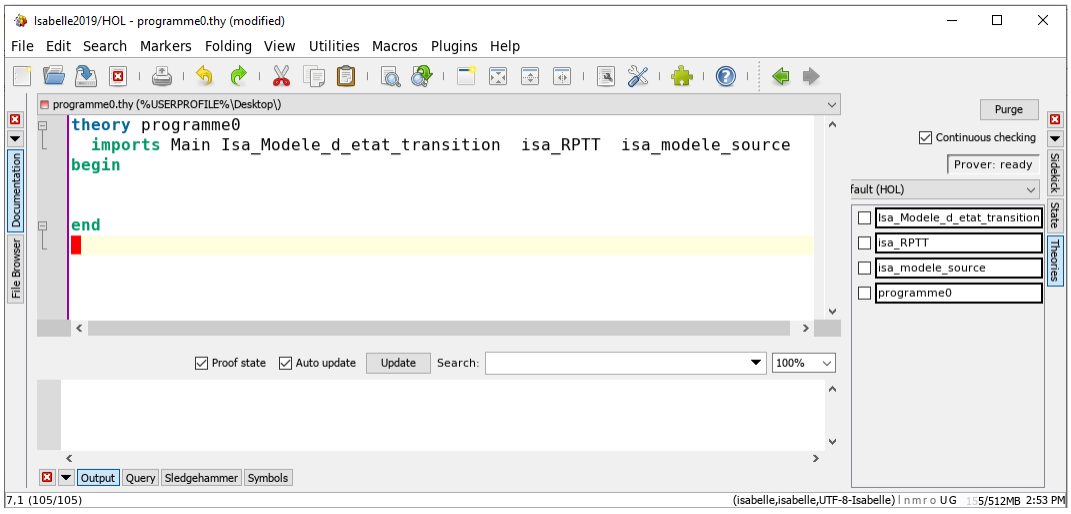


FIGURE IV.1 – L'interface d'Isabelle/HOL.

IV.3 Présentation de l'approche :

L'idée principale de notre approche est illustrée dans la figure IV. 2, qui consiste en un processus en trois étapes. Premièrement, la spécification de modèles état transition est transformée en une description équivalente au sein d'Isabelle nommée Isa _ Modele _ d _ etat _ transition . Dans la deuxième étape, la description Isa _ Modele _ d _ etat _ transition est transformée en description isa _ RPTT, c'est l'étape la plus importante et la plus difficile de notre approche. Par conséquent, cette étape nécessite une preuve d'exactitude en raison de la différence entre le domaine sémantique d'état transition et de RPTT. Dans la troisième étape, nous générons les modèles RPTT réels à partir de la description isa _ RPTT.

Nous n'avons réalisé que la deuxième étape de notre approche, qui se compose de deux parties. La première partie contient la transformation des diagrammes d'état transition en modèles RPTT. La deuxième partie se concentre sur la vérification de certaines propriétés d'exactitude de notre transformation. Dans ce qui suit, nous donnons en détail ces deux parties.

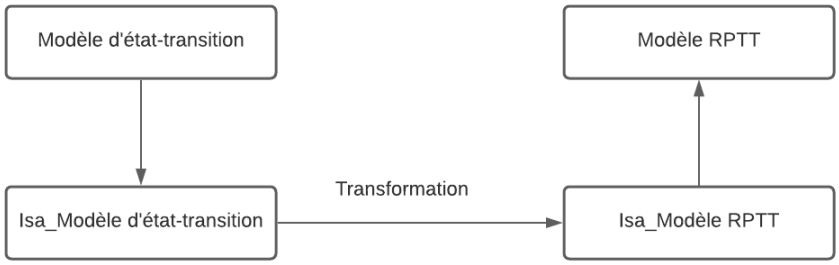


FIGURE IV.2 – Architecture de notre approche.

IV.3.1 Transformation modèle d'état-transition en modèle RPPT :

Afin de transformer les diagrammes d'état-transition en modèles RPTT, nous avons proposé au sein d'Isabelle / HOL une description des modèles d'état-transition nommés Isa_ Modele_ d_ etat_ transition , une description des modèles RPTT nommés isa_ RPTT et la définition Transformation qui effectue automatiquement la transformation. Dans ce qui suit, nous détaillons ces trois étapes.

IV.3.1.1 Spécifications du modèle d'état de transition :

Pour décrire les diagrammes d'état-transition, nous avons proposé la description d'Isabelle Isa_ Modele_ d_ etat_ transition représentée sur la figure IV. 3.

Le type allState représente tous les états des modèles diagrammes d'état transition, qui est un type de données récursif. Il peut être de trois types : la chaîne initiale représente les états ini-

tiales, la chaîne finale représente les états finaux, la chaîne simple représente les états simples. Les transitions sont définies par record transitionET. Une transition se compose de : nom, action, condition, input c'est un état source et outputt c'est un état cible.

Nous définissons des modèles d'état-transition à l'aide record etat _ transition. Un etat_ transition se compose de : un nom de modèle état transition, États une liste de type allState et les transitions sont une liste de type de transitionET.

```
theory Isa_Modele_d_etat_transition
  imports Main
begin
datatype allState =
Initial "string" | Final "string" | Simple "string"

record transitionET=
nom::"string"
action ::"string"
condition ::"string"
input::"allState"
outputt::"allState"

record etat_transition =
name_etat_transition::"string"
etat :: "allState list"
transition :: "transitionET list"

end
```

FIGURE IV.3 – Isa_ Modele_ d_ etat_ transition .

IV.3.1.2 Spécifications du modèle RPTT :

Pour décrire les modèles RPTT, nous avons proposé la description d'Isabelle isa _ RPTT représentée sur la figure IV. 4.

*Les types : transitionRPTT qui est représenté par tuple (string * string * strings * tring) et Place qui est représenté par tuple (string * nat).*

Nous définissons des modèles RPTT à l'aide record RPTT. Un RPTT se compose de : un nom de modèle RPTT, places une liste de type place et les transitions sont une liste de type de transitionRPTT, arc_ in une liste d'arcs Place vers transitionRPTT et arc_ out une liste d'arcs transitionRPTT vers Place.

```
theory isa_RPTT
  imports Main
begin

type_synonym transitionRPTT ="(string×string×string×string)"
type_synonym Place ="(string×nat)"

record RPTT=
name_RPTT::"string "
place::"Place list"
transition_RPTT :: "transitionRPTT list"
arc_in ::  "(Place×transitionRPTT)  list"
arc_out :: "(transitionRPTT  × Place ) list"
end
```

FIGURE IV.4 – Isa_ RPTT.

IV.3.2 Définition de la transformation :

Dans cette partie, nous présentons les idées principales de notre transformation et sa mise en œuvre à l'aide d'Isabelle/HOL, où les états se transforment en Places, tandis que les transitions sont transformés en transitions de RPTT. Cette définition reçoit comme paramètre les modèles d'état-transition et renvoie ses modèles RPTT équivalents. elle repose essentiellement, sur quatre fonctions (sont nos propre fonctions) :

- ***getPlaces :*** *Son objectif est de transformer les états en Places correspondants dans les modèles RPTT.*
- ***gettransition :*** *Son objectif est générer les arcs de place vers transition.*
- ***getoutarc :*** *Son objectif est générer les arcs de transition vers place.*

```
definition Transformation ::" etat_transition ⇒ RPTT"
  where "Transformation etat_transition
          ≡(name_RPTT = name_etat_transition  etat_transition   ,
            place = getPlaces(etat  etat_transition),
            transition_RPTT = gettransition (transition  etat_transition),
            arc_in = getinarc(transition  etat_transition) ,
            arc_out = getoutarc(transition  etat_transition))"
```

FIGURE IV.5 – Définition de la transformation.

IV.3.3 Vérification de la transformation du modèle :

En réalité, les transformations de modèles ont une variété de propriétés, qui sont nécessaires pour garantir leur exactitude, telles que : terminaison, confluence, correction syntaxique, correction sémantique, etc. Les défis les plus importants sont l'exactitude sémantique ou la préservation de la sémantique du modèle source dans le modèle cible. Donc, nous utilisons ici un type spécial de

propriété sémantique appelée propriété de préservation structurelle (STP) puisque chaque transformation a des structures du modèle source qui influencent d'autres structures du modèle cible.

Le principal de cette propriété est que nous faisons une paire de propriétés (P, Q), P est une propriété conforme au modèle source et Q est son équivalent conforme au modèle cible, et nous prouvons que : si P tient sur le modèle source alors Q tient sur le modèle cible, ce qui signifie (P -> Q) pour tout modèle d'entrée.

Dans cette partie, nous implémentons l'approche de vérification [STP] dans la transformation des modèles d'état transition en modèles de RPTT. Tout d'abord, nous avons besoin de la formalisation de cette transformation (des modèles d'état transition vers RPTT), qui est introduite dans la première partie de notre approche.

Ensuite, nous construisons des propriétés d'exactitude dans Isabelle/HOL et prouvons également que ces propriétés sont valables pour cette transformation, pour tout modèle d'entrée.

Dans tout le lemme, X représente le modèle d'entrée (modèle d'état transition) et Transformation (X) représente le modèle de sortie résultant (RPTT). Dans ce qui suit, nous donnons ces preuves d'exactitude en détail :

Lemma a : *prouve que le nom de modèle d'état transition (modèle source représenté par X) et RPTT (modèle cible représenté par Transformation (X)) sont toujours les mêmes.*

```
lemma a: "((name_etat_transition X) = (name_RPTT(Transformation(X))))"
  by (simp add: Transformation_def)
```

FIGURE IV.6 – Lemma a.

Lemma b : *il permet de compléter la preuve du lemma b1 et prouve par récurrence que la fonction récursive getPlaces transforme chaque état initial en un place RPTT.*

```
lemma b:"((List.member(X)(Initial y)))⟶ (List.member(getPlaces(X))(y,1))"
   apply (simp  add: member_def   )
   apply (induct X rule:getPlaces.induct)
   apply auto
   done
```

FIGURE IV.7 – Lemma b.

Lemma b1 : *prouve qu'un état initial (Initial y : : allState) appartenant à modèle d'état transition (modèle d'entrée X) implique que sa place correspondante (y, 1) appartient au modèle cible RPTT (Transformation (X)).*

```
lemma b1:"((List.member(etat X)(Initial y)))⟶ (List.member
     (place(Transformation(X)))(y,1))"
  apply (induct X )
  apply (simp_all  add: member_def Transformation_def)
using b in_set_member  member_def by fastforce
```

FIGURE IV.8 – Lemma b1.

Lemma c : *il permet de compléter la preuve du lemma c1 et prouve par récurrence que la fonction récursive getPlaces transforme chaque état final en un place RPTT.*

```
lemma c:"(List.member(X)(Final y) )⟶ (List.member(getPlaces(X))(y,0))"
   apply (simp  add: member_def   )
   apply (induct X rule:getPlaces.induct)
   apply auto
   done
```

FIGURE IV.9 – Lemma c.

Lemma c1 : *prouve qu'un état final (Final y : : allState) appartenant à modèle d'état transition (modèle d'entrée X) implique que sa place correspondante (y, 0) appartient au modèle cible RPTT (Transformation (X)).*

```
lemma c1:"((List.member(etat X)(Final y)))⟶ (List.member
   (place(Transformation(X)))(y,0))"
  apply (induct X )
  apply (simp_all  add:member_def Transformation_def)
using c in_set_member  member_def by fastforce
```

FIGURE IV.10 – Lemma c1.

Lemma d : *il permet de compléter la preuve du lemma d1 et prouve par récurrence que la fonction récursive getPlaces transforme chaque état simple en un place RPTT.*

```
lemma d:"(List.member(X)(Simple y))⟶ (List.member(getPlaces(X))(y,0))"
  apply (simp  add: member_def)
  apply (induct X rule:getPlaces.induct )
  apply auto
  done
```

FIGURE IV.11 – Lemma d.

Lemma d1 : *prouve qu'un état simple (Simple y : : allState) appartenant à modèle d'état transition (modèle d'entrée X) implique que sa place correspondante (y, 0) appartient au modèle cible RPTT (Transformation (X)).*

```
lemma d1:"((List.member(etat X)(Simple y)))⟶(List.member
       (place(Transformation(X)))(y,0))"
  apply (induct X  )
  apply (simp_all   add:  member_def Transformation_def)
  using d in_set_member  member_def by fastforce
```

FIGURE IV.12 – Lemma d1.

Lemma e : *il permet de compléter la preuve du lemma e1 et prouve par récurrence que la fonction récursive gettransition transforme chaque transition en une transition RPTT.*

```
lemma e:"(List.member(X)(T))⟶ (List.member(gettransition(X))
      (nom T,action T ,condition T,'']-oo,+oo['') )"
  apply (simp  add: member_def)
  apply (induct X )
  apply auto
  done
```

FIGURE IV.13 – Lemma e.

Lemma e1 : *prouve qu'une transition (T : : Transition) appartenant à modèle d'état transition (modèle d'entrée X) implique que sa transition correspondante (nom T, action T, condition T, "]-oo,+oo[")appartient au modèle cible RPTT (Transformation (X)).*

```
lemma e1:"((List.member(transition X)(T)))⟶(List.member(transition_RPTT
      (Transformation(X)))(nom T,action T ,condition T,'']-oo,+oo['']))"
  apply (induct X )
  apply (simp_all add:member_def Transformation_def )
  using e in_set_member  member_def by fastforce
```

FIGURE IV.14 – Lemma e1.

IV.4 Comparaison :

Dans le travail [11] proposaient une approche et un outil pour la modélisation et la transformation des diagrammes d'états transitions en modèles des réseaux de Petri temporellement temporisés en utilisant AToM3 ç'est un outil idéal pour faire la transformation parce qu'il a deux tâches principales sont la méta-modélisation et la transformation de modèles et il nous aide à faire un outil de modélisation qui ne peut pas être faire avec isabelle/hol.

ET dans notre travail nous proposons une approche pour vérifier la transformation des diagrammes d'états-transitions en modèles des réseaux de Petri temporellement temporisés en utilisant Isabelle/HOL ç'est un outil idéal pour la vérification qu'elle ne peut pas être faire avec AToM3.

IV.5 Etude de cas :

Pour illustrer l'approche proposée nous utilisons un exemple, le principe de ce système est comme suit : après le recrutement, une personne est considérée en activité dès sa prise de fonction dans l'entreprise. Au cours de sa carrière, nous retiendrons seulement les évènements : congé de maladie et prise de congé annuel. Enfin de carrière, nous retiendrons deux situation : la démission et la retraite.

Diagramme d'etat-transition d'exemple :

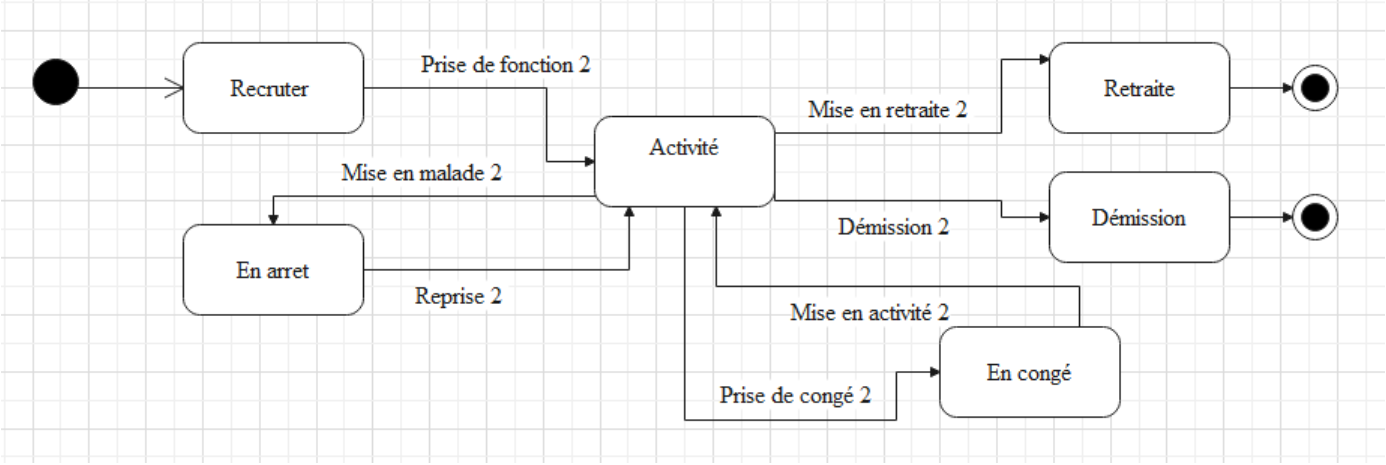


FIGURE IV.15 – Modèle source.

Sa spécification avec isabelle/hol :

```
"(name_etat_transition = '' exemple'',
    etat =
      [Initial ''Recruter'', Simple ''Activte'', Simple ''En arret'', Simple ''En conge'',
       Final ''Retraite'', Final ''Demission''],
    transition =
      [(nom = ''T0'', action = ''prise de fonction'', condition = ''2'', input = Initial ''Recruter'',
         outputt = Simple ''Activte'')),
       (nom = ''T1'', action = ''mise en malade'', condition = ''2'', input = Simple ''Activte'',
         outputt = Simple ''En arret'')),
       (nom = ''T2'', action = ''reprise'', condition = ''2'', input = Simple ''En arret'',
         outputt = Simple ''Activte'')),
       (nom = ''T3'', action = ''prise de conge'', condition = ''2'', input = Simple ''Activte'',
         outputt = Simple ''En conge'')),
       (nom = ''T4'', action = ''mise en activite'', condition = ''2'', input = Simple ''En conge'',
         outputt = Simple ''Activte'')),
       (nom = ''T5'', action = ''mise en retraite'', condition = ''2'', input = Simple ''Activte'',
         outputt = Final ''Retraite'')),
       (nom = ''T6'', action = ''demission'', condition = ''2'', input = Simple ''Activte'',
         outputt = Final ''Demission''))])"
  :: "etat_transition"
```

FIGURE IV.16 – Isa_ modele_ source.

Résultat dans Isabelle/hol :

```
"(name_RPTT = '' exemple'',
   place = [(''Recruter'', 1), (''Activte'', 0), (''En arret'', 0), (''En conge'', 0), (''Retraite'', 0), (''Demission'', 0)],
   transition_RPTT =
     [(''T0'', ''prise de fonction'', ''2'', '']-oo,+oo['']), (''T1'', ''mise en malade'', ''2'', '']-oo,+oo['']),
      (''T2'', ''reprise'', ''2'', '']-oo,+oo['']), (''T3'', ''prise de conge'', ''2'', '']-oo,+oo['']),
      (''T4'', ''mise en activite'', ''2'', '']-oo,+oo['']), (''T5'', ''mise en retraite'', ''2'', '']-oo,+oo['']),
      (''T6'', ''demission'', ''2'', '']-oo,+oo[''')],
   arc_in =
     [((''Recruter'', 1), ''T0'', ''prise de fonction'', ''2'', '']-oo,+oo['']),
      ((''Activte'', 0), ''T1'', ''mise en malade'', ''2'', '']-oo,+oo['']),
      ((''En arret'', 0), ''T2'', ''reprise'', ''2'', '']-oo,+oo['']),
      ((''Activte'', 0), ''T3'', ''prise de conge'', ''2'', '']-oo,+oo['']),
      ((''En conge'', 0), ''T4'', ''mise en activite'', ''2'', '']-oo,+oo['']),
      ((''Activte'', 0), ''T5'', ''mise en retraite'', ''2'', '']-oo,+oo['']),
      ((''Activte'', 0), ''T6'', ''demission'', ''2'', '']-oo,+oo[''')],
   arc_out =
     [((''T0'', ''prise de fonction'', ''2'', '']-oo,+oo['']), ''Activte'', 0),
      ((''T1'', ''mise en malade'', ''2'', '']-oo,+oo['']), ''En arret'', 0),
      ((''T2'', ''reprise'', ''2'', '']-oo,+oo['']), ''Activte'', 0),
      ((''T3'', ''prise de conge'', ''2'', '']-oo,+oo['']), ''En conge'', 0),
      ((''T4'', ''mise en activite'', ''2'', '']-oo,+oo['']), ''Activte'', 0),
      ((''T5'', ''mise en retraite'', ''2'', '']-oo,+oo['']), ''Retraite'', 0),
      ((''T6'', ''demission'', ''2'', '']-oo,+oo['']), ''Demission'', 0)])"
 :: "RPTT"
```

FIGURE IV.17 – Isa_ modele_ cible.

Modèle RPTT d'exemple :

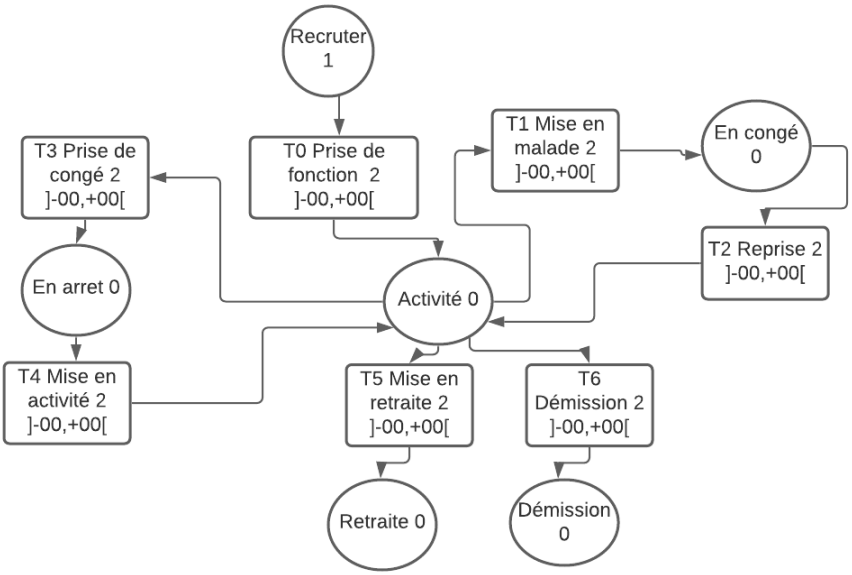


FIGURE IV.18 – Modèle cible.

IV.6 Conclusion :

Dans ce chapitre nous avons proposés une nouvelle approche qui permet la transformation des diagrammes d'états transitions vers les réseaux de Petri temporellement temporisés ainsi que la vérification de l'exactitude de la transformation elle-même.

Nous avons réalisés toutes les étapes de l'approche à l'aide de l'assistant de preuve Isabelle/HOL, à la fin nous avons présenté un cas d'étude a fin d'illustrer notre approche.

Finalement nous avons montré l'efficacité de notre approche à travers les résultats obtenus.

Conclusion générale

Dans ce mémoire nous sommes intéressés a deux concepts principaux le premier est la transformation de modèle qui est considéré comme un concept de base dans le domaine de l'ingénierie dirigée par les modèles IDM et le deuxième c'est l'intégration de la vérification dans ce processus.

La vérification de la justesse de transformation de modèles est une tâche très importante, elle nécessite premièrement la spécification des propriétés de transformation ensuite la preuve formelle de ces propriétés.

Le résultat de notre travail est une nouvelle approche qui sert à transformer les digrammes d'états transitions UML vers les réseaux de pétri temporellement temporisés RPTT ainsi que la vérification de cette transformation, cette approche est réalisée à l'aide de l'assistant de preuve Isabelle/HOL.

Notre approche est réalisée dans trois étapes :

La première étape est la spécification consiste à proposer une description textuelle pour le diagramme d'états transitions et une autre pour les RPTT.

La deuxième étape est la transformation consiste à proposer un algorithme de transformation compose d'un ensemble de règles de transformation.

La troisième et la dernière étape est la vérification consiste à vérifier la justesse ou l'exactitude de l'algorithme de transformation.

Enfin et comme perspectives vérifier d'autres propriétés de transformation de modèle par exemple terminaison, confluence, correction syntaxique, correction sémantique, etc.

Bibliographie

[1] *El-HILALI KERKOUCHE, Modélisation Multi-Paradigme : Une Approche Basée sur la Transformation de Graphes, Université de mantouri Constantine, 04 / 07 /2011*

[2] *Berabah Fatiha, Chemani Khadidja , Une Approche De Transformation Des Diagrammes de sequense UML Vers Les Réseaux De Pétri Temporellement Temporisés.Unviersité Djilali Bounaama, Khemis Miliana, 2019.*

[3] *BERRAMLA KARIMA, vérification et validation des transformations des modèles, Université d'Oran ,02/07/2014.*

[4] *Bouanae Razika, Une Approche De Transformation Des Diagrammes D'activités UML Vers Les Réseaux De Pétri Temporellement Temporisés.Unviersité Djilali Bounaama, Khemis Miliana, 2018.*

[5] *MounaBouarioua, Une approche basée transformation de graphes pour la génération de modèles de réseaux de Petri analysables à partir de diagrammes UML, UNIVERSITÉ CONSTANTINE 2, 24/ 04 /2013.*

[6] *Baouche Ikbal, Rahali Mohmed, Une approche dirigée par les modèles pour la modélisation des services web composés Université Djilali Bounaama, Khemis Miliana, 2017.*

[7] *BENDIAF Messaoud, "SPÉCIFICATION ET VÉRIFICATION DES SYSTÈMES EMBARQUÉS TEMPS RÉEL EN UTILISANT LA LOGIQUE DE RÉÉCRITURE " , UNIVERSITE MOHAMED KHIDER BISKRA , 15/05/2018.*

[8] *M. SELLAM MILOUD, "Vérification et validation de transformation de modèles", UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE d'ORAN Mohamed Boudiaf, 14/12/2014.*

[9] *HETTAB ABDELKAMEL, Mémoire de Magistère dans le cadre de l'école doctoral pôle Est. "De M-UML vers les réseaux de Petri « Nested Nets » : Une approche basée transformation de graphes". Université Constantine, Novembre 2009.*

[10] *Proposition d'une méthode de conception d'un superviseur de contrôle pour un processus de production distribué, chapitre 2.*

[11] *kefif Nadia, Kouadri boudjelthi arabéa, Une Approche De Transformation Des Diagrammes D'état transition UML Vers Les Réseaux De Pétri Temporellement Temporisés. Université Djilali Bounaama, Khemis Miliana, 2016.*

[12] *BOUDIA MALIKA "Transformation des diagrammes d'états-transitions vers Maude" UNIVERSITE DE M'SILA, 02 /06/2011.*

[13] Houda HAMROUCHE, Présenté en vue de l'obtention du diplôme de MAGISTER en INFORMATIQUE Ecole Doctorale en Informatique de l'Est – EDI Est "Une Approche de transformation des diagrammes D'activité d'UML vers CSP basée sur la transformation de graphes", UNIVERSITE 20 AOUT 1955 – SKIKDA.

[14] Kirli Asma, Bouhini Meryem, Une approche dirigée par les modèles basée sur UML pour la mise en œuvre de services web composés Université Djilali Bounaama, Khemis Miliana, 2017.

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

[16] Jeannette M Wing. Formal Methods. Encyclopedia of Software Engineering, Revision in Second Edition, 1994.

[17] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, Jean-Marc Jézéquel, Model-Driven Engineering for Software Migration in a Large Industrial Context, 29 /4/2010.

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

[19] Said Meghzili, Allaoua Chaoui, Martin Strecker, Elhillali Kerkouche, On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation using Isabelle/HOL, MISC Laboratory, Computer Science. Dept., University of Constantine 2-Abdelhamid Mehri, Constantine, Algeria, 2017.

[20] GUERROUF FAYÇAL, "Une Approche de Transformation des Diagrammes d'Activités d'UML Mobile 2.0 vers les Réseaux de Petri ", Université El Hadj Lakhdar – BATNA.

[21] Jean-Paul Comet, Les réseaux de Petri pour la simulation de systèmes biologiques, Université de Nice-Sophia-Antipolis Ecole Polytech 25/4/ 2014.

[22] Maurice Comlan, Contribution au dépliage des réseaux de Petri et à l'analyse des processus de branchement, 'Université Bretagne Loire, 3/11/2016.

[23] Yann Morére, Cours de réseau de Petri, Avril 2002.

[24] Cedric Buche, Modèle pour L'ingénierie des Systèmes _ UML _, Ecole Nationale des Ingénieurs de Brests (ENB), 8/1/2014

[25] Diagramme états transitions, Classe de première SI, S diagramme_ etats.odt.

[26] Laurent AUDIBERT, UML 2.0, Institut Universitaire de Technologie de Villetaneuse

[27] André MIRALLES, INGENIERIE DES MODELES POUR LES APPLICATIONS ENVIRONNEMENTALES, U N I V E R S I T E D E M O N T P E L L I E R I I,11/12/2006.

[28] Halim Bou Chaboub, Transformation d'une ontologie OWL-S vers un reseau petri, Université L'arbi Ben M'hidi Oum El Bouaghi, 2012.

[29] N. Belala, D.E. Saidouni, R. Boukharrou, A.C. Chaouche, A. Seraoui, A. Chachoua, TIME PETRI NETS WITH ACTION DURATION : A TRUE CONCURRENCY REAL TIME MODEL,Université Constantine.

[30] Asma CHACHOUA, Radja BOUKHARROU, Mise en oeuvre distribuée de la sémantique opérationnelle des réseaux de Petri temporellement temporisés : Approche Orientée Objet, Université Mentouri — Constantine, 2005-2011.

[31] Christian ATTIOGBÉ, Contributions aux approches formelles de développement de logiciels Intégration de méthodes formelles et analyse multifacette, Université de Nantes - UFR Sciences, 13/9/2007.

[32] TRUONG Ninh Thuan, Utilisation de B pour la v'erification de sp'ecifications UML et le d'enveloppement formel orienté objet, Ecole doctorale IAEM Lorraine, 5/5/2006.

[33] https ://isabelle.in.tum.de/ 13.00 10/7/2020.

[34] Said Meghzili1, Allaoua Chaoui1, Martin Strecker, Elhillali Kerkouche, Verification of Model Transformations Using Isabelle/HOL and Scala, Springer Science+Business Media, LLC, part of Springer Nature 2018.

[35] Clemens Ballarin, Stefan Berghofer, Jasmin Blanchette, Timothy Bourke, Lukas Bulwahn, Amine Chaieb, Lucas Dixon, Florian Haftmann, Brian Huffman, Lars Hupel, Gerwin Klein, Alexander Krauss, Ondrej Kuncar, Andreas Lochbihler, Tobias Nipkow, Lars Noschinski, David von Oheimb, Larry Paulson, Sebastian Skalberg, Christian Sternagel, Dmitriy Traytel, The Isabelle/Isar Reference Manual Part III, 9/6/2019.