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 *Ingénierie De Logiciel*

Thème

Une Approche De transformation et D'analyse Des Diagrammes D'activités

Présenté par :

- **CHERIEF Saliha**
- **MELIANI Toufik**

Devant le jury composé de :

- Examinateur 1 :
- Examinateur 2 :
- Encadreur : **HACHICHI Hiba**

Année Universitaire : 2019/2020

Dédicaces

Ma mère, qui a œuvré pour ma réussite, de par son amour, son soutien, tous les sacrifices consentis et ses précieux conseils, pour toute son assistance et sa présence dans ma vie, reçois à travers ce travail aussi modeste soit-il, l'expression de mes sentiments et de mon éternelle gratitude.

Mon père, qui peut être fier et trouver ici le résultat de longues années de sacrifices et de privations pour m'aider à avancer dans la vie. Puisse Dieu faire en sorte que ce travail porte son fruit ; Merci pour les valeurs nobles, l'éducation et le soutient permanent venu de toi.

A mes chère sœurs Fatima, Houria et Faiza pour leurs encouragements permanents, et son soutien moral.

A mon cher frère, Ismail (rabi yhfdo), pour son appui et son encouragement

A ma petite fille Chaima.

A mon grand père Ali et mes oncles.

A toute ma famille CHERIEF pour leur soutien tout au long de mon parcours universitaire.

A tous mes ami(e)s et collègues Nadjet, Zhour, Kaouter, Zineb, Saida, Cherifa Houda.

A toute la promotion d'informatique 2019/2020.

Que ce travail soit l'accomplissement de vos vœux tant allégués, et le fuit de votre soutien infaillible.

Merci d'être toujours là pour moi.

SALIHA

Dédicaces

Ma mère, qui a œuvré pour ma réussite, de par son amour, son soutien, tous les sacrifices consentis et ses précieux conseils, pour toute son assistance et sa présence dans ma vie, reçois à travers ce travail aussi modeste soit-il, l'expression de mes sentiments et de mon éternelle gratitude.

Mon père, qui peut être fier et trouver ici le résultat de longues années de sacrifices et de privations pour m'aider à avancer dans la vie. Puisse Dieu faire en sorte que ce travail porte son fruit ; Merci pour les valeurs nobles, l'éducation et le soutient permanent venu de toi.

Toufik

Remerciements

On remercie DIEU le tout puissant qui nous a donné la force, la volonté et le courage pour accomplir ce modeste travail.

On adresse également nos remerciements à nos parents Pour le soutien inconditionnel qui nous ont apporté au cours des années de nos études supérieurs. Leur appui nous a été très précieux et nous leur en témoigne aujourd'hui notre plus grande reconnaissance On tient également à remercier très chaleureusement notre encadreur HIBA HACHICHI. Pour son soutien, sa disponibilité, sa patience, et son aide qui nous ont permis de mener à bien ce travail.

Nous ne saurions oublier dans ces remerciements MONSIEUR LE Recteur de l'Université de Djilali Bounaama, qui nous a donné la chance d'accéder au master2.

On adresse également nos remerciements à nos amies et nos collègues de la cellule site web et centre systèmes et réseaux qui nous ont soutenu durant cette année.

Enfin merci à ceux et celles qui de près ou de loin nous ont permis d'arriver à bout dans ce long périple.

ملخص

أدى التطور التكنولوجي في مجال الشبكات والاتصالات إلى تطوير تطبيقات تناسب التطورات الجديدة ، مثل بروتوكولات الاتصال والأنظمة الموزعة. مع أخذ بالاعتبار الجانب الزمني.

في الواقع ، يجب أن تتضمن النمذجة آليات إدارة الوقت والأحداث التي تؤخذ في الاعتبار فكرة التطورات المتزامنة للإجراءات. ونتيجة لذلك نحن بحاجة إلى نماذج تتحمل إجراءات هيكلية ومؤقتة.

علما ، RPTT يملك إجراءات يمكن أن تستمر مع مرور الوقت ولها الفرصة للتعبير عن هذا النوع من السلوك.

العمل المقدم في هذا المشروع هو مساهمة في مجال الهندسة التي تعتمد على النموذج (IDM). هدفها الرئيسي هو تطبيق تقنيات تحويل النموذج ، وبشكل أكثر دقة تحويلات الرسم البياني ، لتكون قادرة على تطبيق أدوات التحليل والتحقق أثناء عملية تطوير الأنظمة المعقدة. لذلك ، نقترح نهجاً وأداة لنمذجة وتحويل والتحقق من خلال عملية تطوير الأنظمة المعقدة.

الكلمات الرئيسية : هندسة البرامج بواسطة النماذج ، الطرق التحليلية ، تحويل الرسم البياني ،القواعد النحوية البيانية،التصميم المتعدد النماذج .

Résumé

L'évolution technologique dans le domaine des réseaux et télécommunications nous poussait à faire développer des applications bien appropriées à ces nouveaux progrès, telles que les protocoles de communication et les systèmes repartis. Ces derniers appréhendent l'aspect temporel.

En effet, la modélisation doit faire intervenir des mécanismes de gestion du temps et d'évènements où la notion d'évolutions simultanées d'actions est prise en compte. Par conséquent, nous avons besoin d'un modèle sémantique du parallélisme qui supporte l'expression d'actions non atomiques structurellement et temporellement, c'est-à-dire d'actions divisibles et non nécessairement de durée nulle. Cependant, les réseaux de Pétri temporellement temporisés (RPTT) dans lesquels les actions peuvent durer dans le temps ont apporté la possibilité d'exprimer ce type de comportements.

Le travail présenté dans ce projet est une contribution dans le domaine de l'ingénierie dirigée par les modèles (MDI). Son objectif principal est l'application de techniques de transformation de modèles, et plus précisément de transformations de graphes, pour pouvoir appliquer des outils d'analyse et de vérification au cours du processus de développement de systèmes complexes. Par conséquent, nous proposons une approche et un outil pour la modélisation et la transformation des diagrammes d'activité en modèles de réseaux de pétri à temporisation temporelle.

Mots clés : MDE, ATL, GMF, transformation, Réseaux de Petri, UML, OMG, EMF, Ecore .diagramme d'activité.

Abstract

The technological evolution in the field of networks and telecommunications led us to develop applications that are well suited to these new advances, such as communication protocols and distributed systems. The latter apprehend the temporal aspect.

Indeed, the modeling must involve time management mechanisms and events where the notion of simultaneous evolutions of actions is taken into account. Therefore, we need models that support the expression of non-atomic actions structurally and temporally, that is to say divisible actions and not necessarily of zero duration.

However, temporally timed petri net in which actions can last over time have provided the opport unity to express this type of behavior. The work presented in this project is a contribution in the field of Model Driven Engineering (MDI). Its main objective is the application of model transformation techniques, and more precisely graph transformations, to be able to apply analysis and verification tools during the development process of complex systems.

Therefore, we propose an approach and a tool for the modeling and the transformation of the activity diagrams into temporally timed models of Petri nets.

Keywords : MDA, ATL, GMF, transformation, Petri nets , UML, OMG, EMF, Ecore.activity diagram.

Table des matières

Table des figures

Introduction générale

Les systèmes logiciels sont présentés aujourd'hui, dans tous les domaines de l'activité humaine (industrie, transports, communications, construction, etc.). Avec le temps, ces systèmes sont devenus de plus en plus complexes. Les systèmes complexes sont caractérisés, non seulement par un grand nombre de composants, mais aussi par la diversité de ces composants. Ceci implique que ces composants sont modélisés dans différents langages de modélisation ou formalismes.

L'Ingénierie dirigée par les modèles (IDM) est une pratique d'Ingénierie des systèmes qui utilise les capacités des technologies informatiques pour décrire à la fois le problème posé et sa solution en utilisant des modèles, des concepts, et des langages. Les transformations de ces modèles spécifient des raffinements d'un niveau d'abstraction vers un autre.

Pour valider les avantages de ce processus et de la méthode qu'il incarne, il a été choisi de tenter une partie de son implémentation dans le langage ATL spécialement conçu pour la transformation de modèles. Ce langage est développé dans le cadre du projet ATLAS mené à l'université de Nantes par une équipe de chercheurs .

Le travail présenté ici est une expérimentation et une évaluation de l'utilisation d'ATL pour réaliser une transformation d'un diagramme d'activité vers les RPTT on se basant sur l'outil EMF qui nous permet de construire des applications basées sur des modèles.

Pour cela ce mémoire est organisé en quatre chapitres :

Le premier chapitre est consacré à l'étude des concepts de base de l'ingénierie dirigée par les modèles (IDM) et leur approche l'architecture dirigée par les modèles (MDA), suivi par la transformation des modèles et le processus de la vérification dans l'IDM, Nous présentons par la suite la transformation des modèles et enfin nous expliquons la transformation de graphe en donnant une vision de langage de transformation utilisé avec les outils associés.

Le chapitre 2 est consacré aux présentations de l'UML2.0, où nous avons éclairci un bref historique suivi par une simple définition de l'UML ainsi que ses différents diagrammes on basant sur le diagramme d'activité. On a présenté par la suite, ce diagramme et ses éléments qui constituent le modèle de base dans notre travail.

Dans le chapitre 3, nous avons présenté les méthodes formelles utilisées dans l'approche au niveau d l'IDM pour la vérification et et décrit ses diagrammes et leurs concepts, en effet ; nous présentons les diagrammes d'activités et leurs principaux élé-

ments qui constituent le modèle de base dans notre travail.

Dans le quatrième chapitre, nous avons adopté un mécanisme de transformation automatique de diagramme d'activité àréseaux de pétri temporellement temporisés). La méthode proposée est basée sur le langage de transformation ATL, et utilise l'outil de modélisation et de méta-modélisation EMF et GMF.nous terminons par une conclusion générale qui résume les contributions de ce mémoire, et discute des limites et des perspectives de notre travail.

Chapitre 1

L'Ingénierie Dirigée par les Modèles

1.1 Introduction

L'Ingénierie Dirigée par les Modèles (IDM) ou Model Driven Engineering (MDE) fait partie des domaines de l'informatique et de l'ingénierie système, elle fournit les outils et concepts nécessaires pour créer ou modifier des langages de modélisation. Ce domaine relativement récent à fait son apparition dans les années 80 avec un premier outil qui supportait les concepts de l'IDM, "the Computer-Aided Software Engineering" (CASE). La complexité grandissante des systèmes à modéliser à conduit les développeurs à manipuler des concepts de plus haut niveau.

L'IDM permet de réutiliser des modèles de formalisme appelés méta-modèles, qui sont adaptables à toutes les plateformes comme les composants d'une voiture. Certains organismes s'intéressent de plus en plus à l'IDM et cherchent à la normaliser : c'est le cas de l'OMG (Object Management Group). Cette association qui est à l'origine de langage de modélisation comme UML.

Dans ce chapitre , nous commençons par une présentation les principes et les concepts de base de l'IDM , ensuite nous entreprenons les approches de l'IDM et plus particulièrement l'architecture dirigée par les modèles (MDA), Nous présentons par la suite la transformation des modèles et enfin nous expliquons la transformation de graphe.

1.2 Les principes de l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles se base sur le principe « tout est modèle ».

Un modèle est une abstraction de la réalité (le système). Il aide à répondre aux questions que l'on peut se poser sur le système modélisé. Pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine.

Cette nouvelle approche peut être considérée à la fois en continuité et en rupture avec les précédents travaux [1, 2, 3, 4].

Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier

en fonction de leurs différentes origines (objets métiers, techniques, etc.).

L'IDM vise donc, à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Alors que l'approche objet est fondée sur deux relations essentielles, « Instance De » et « Hérite De», l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension. A partir des travaux de l'OMG1, de Bézivin et al. Et de Seide witz [1, 2, 5, 6, 4].

1.3 Concepts de base de l'ingénierie dirigée par les modèles

Dans cette section nous définissons les **Concepts** fondamentaux de l'Ingénierie Dirigée par les Modèles.

a. Modèle :

En réalité il n'existe pas à ce jour de définition universelle : "A model represents reality for the given purpose ; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality" [7, 8].

Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé [1, 6].

ont défini un modèle comme étant « une simplification d'un système créé avec un but spécifique. Le modèle doit être capable de répondre aux demandes à la place du système étudié.[6][10] Les réponses fournies par le modèle doivent être les mêmes que celles fournies par le système, à condition qu'elles restent dans la limite du domaine défini par le but général du système » [9].

Pour qu'un modèle donne une représentation efficace d'un système qu'il modélise, il doit respecter quelques caractéristiques du modèle parmi lesquels : [1, 9]

- **Abstraction :** un modèle doit donner une abstraction correcte du système ; c.-à-d. il doit assurer les besoins souhaités de la même façon que le système les assure.
- **La compréhensibilité :** le modèle ne doit pas être ambigu, facile a comprendre par les utilisateurs. Pour cela il doit être formalisé dans un langage compréhensif.
- **La substituabilité :** le modèle doit être précis et suffisant c'est-à-dire en substitution avec le système qu'il modélise, le modèle doit contenir les propriétés à étudier par le système.
- **L'économie :** le modèle doit être peu coûteux par rapport aux coûts du développement du système réel.

b. Méta-modèles et méta-modélisation :

Selon Jean Marie Favre « *est un modèle d'un langage de modélisation. Il définit les concepts ainsi que les relations entre ces concepts nécessaires à la description des modèles* » [9, 1, 6].

La définition de méta-modèle mène à l'identification d'une autre relation entre le modèle et le méta-modèle, appelée « **conforme à** ».

On dit qu'un modèle est conforme à un méta-modèle si tous les éléments du modèle sont instance du méta-modèle ; et les contraintes exprimées sur le méta-modèle sont respectées [1].

Dans l'IDM il existe deux relations de base :

- La première relation lie le modèle et le système modélisé, elle s'appelle "**Représentation de**".
- La seconde relation lie le modèle et le méta-modèle du langage de modélisation, c'est la relation "**Conforme à**" [1].

La figure 1.1 représente la relation entre le système et le modèle, ainsi que, la relation entre le modèle et le méta-modèle. [3]

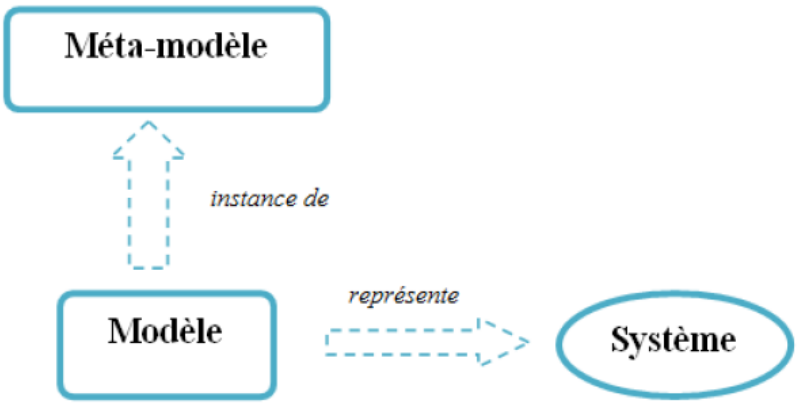


Figure 1.1: Les relations de bases dans l'IDM [3]

Dans la **figure** 1.1, la relation représente dénote d'un modèle est une représentation d'un système, tant disque la relation instance de dénote qu'un modèle est conforme à un méta-modèle si ce modèle appartient à l'ensemble modélisé par ce méta-modèle. [3]

La **figure** 1.2 montre un exemple des relations de l'IDM.

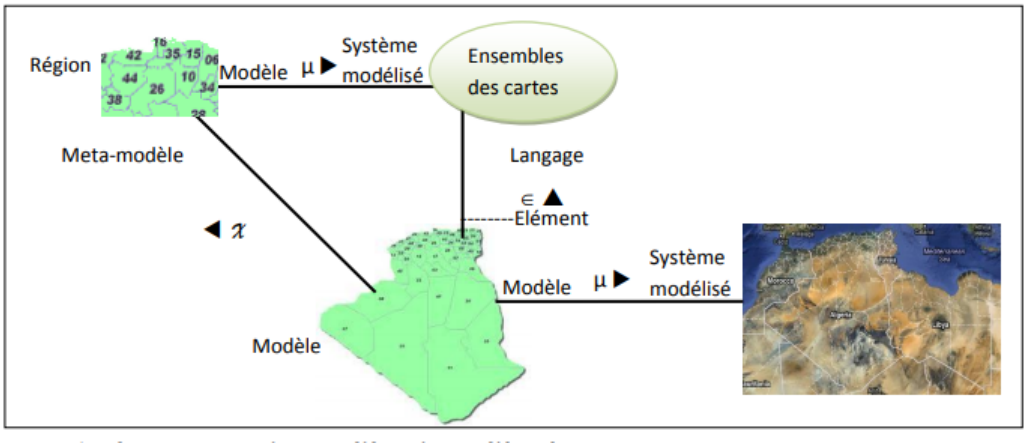


Figure 1.2: Un exemple montrant les relations de base dans l'IDM [10]

En général, La métamodélisation consiste à fournir la définition du langage de modélisation sous forme de modèle également. Cette définition abstraite et rigoureuse représente une spécification pour les différentes implantations du langage défini.

c. Langage de modélisation :

Un langage de modélisation pour un domaine a plusieurs aspects. Il doit définir les concepts du domaine c'est à dire représenter ces concepts sous forme de diagramme ou de texte ; il doit définir les moyens par lesquels un utilisateur peut interagir avec la langue ; et définir ce qu'est un modèle valide et ce qu'il n'est pas ; et comment les modèles seront échangés [11] objectif de langage de modélisation peut être utilisé pour exprimer de l'information ou de la connaissance ou des systèmes dans une structure qui est définie par un ensemble cohérent de règles.

d. Méta-méta-modèle :

Selon Jean Marie Favre [12] Un méta-méta-modèle est un modèle qui décrit un langage de méta-modélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même [4].

Un méta-méta-modèle est le modèle d'un langage permettant d'écrire des langages de modélisation. Il est conforme à lui-même, devenant ainsi le sommet de la pile de méta-modélisation (voir FIGURE 1.3). Ainsi, chaque plateforme de modélisation est basée sur un méta-méta-modèle. On peut citer par exemple MOF et sa version simplifiée EMOF définis par l'OMG [13].

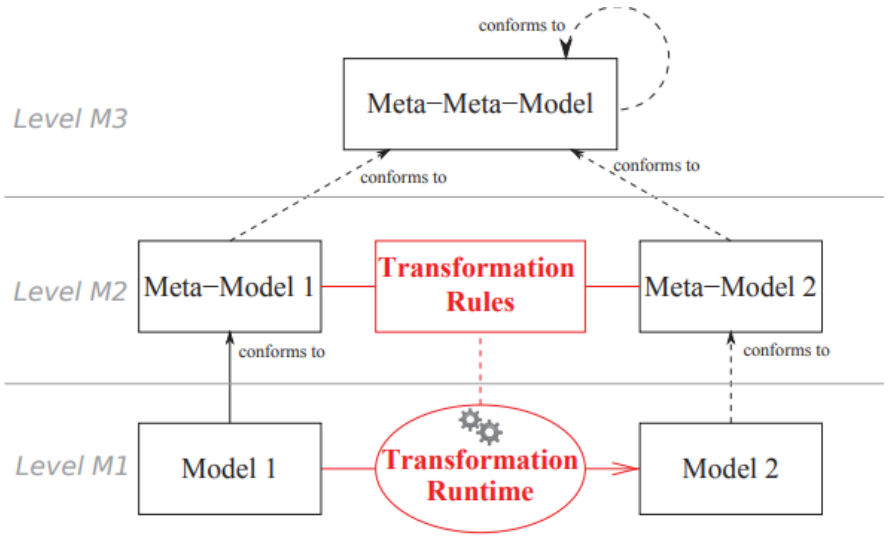


Figure 1.3: Pile de méta-modélisation représentant les différents niveaux de modélisation [13]

e. Plateforme :

Une plate-forme est un système offrant des services nécessaires à la construction, La réalisation ou l'exécution d'autres systèmes [4].

On distingue deux types de plateformes : Dans la pratique : les plates-formes logicielles et les plates-formes matérielles.

1.4 Transformation de modèles :

Les concepts le plus importants de l'IDM est la transformation de modèle.

Une transformation de modèle est une technique qui génère un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources conformément à un ensemble de règles de transformations. Ces règles décrivent essentiellement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible [1].

D'après la définition de la transformation, La transformation est un ensemble de règles de transformation qui décrivent comment un modèle dans un langage source est transformé en modèle dans un langage cible. [1]

1.4.1 Classification des approches de transformation de modèles :

Les approches de transformation de modèle ont été classées selon plusieurs axes. Chaque axe mène à une classification particulière. La classification des approches de transformation proposée par Czarnecki et Helsen [14] se base sur les techniques de transformation utilisées dans les approches et les facettes qui les caractérisent. Selon cette classification on peut distinguer deux types de transformation de modèles : les transformations de type *modèle vers code* et les transformations de type *modèle vers modèles*.

Généralement, le premier type de transformation peut être vu comme un cas particulier du deuxième type, nous avons seulement besoin de fournir un méta-modèle pour le langage de programmation cible. Cependant, pour des raisons pratiques de réutilisation de la technologie des compilateurs existants, souvent, le code est simplement généré en tant que texte, qui est ensuite introduit dans un compilateur. Pour cette raison, nous distinguons entre la transformation de type modèle vers code et la transformation de type modèle vers modèle [15]. La figure 1.4 résume la classification des approches de transformations.

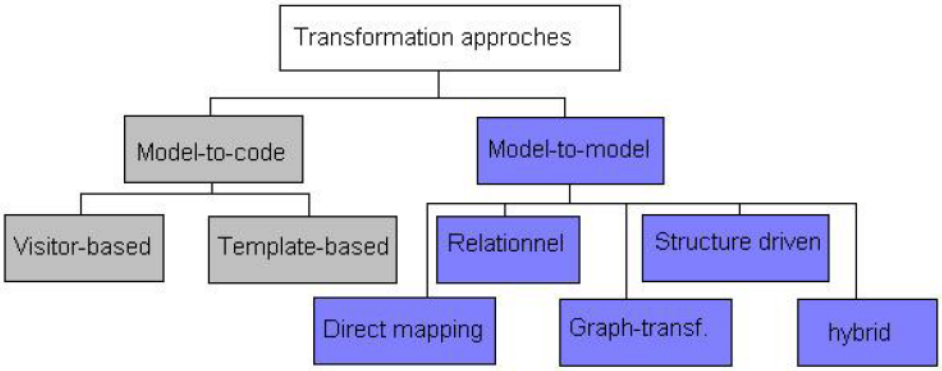


Figure 1.4: Approches de transformation de modèles [16]

1.4.1.1 Transformations de type Modèle vers code (*M2T* : model to text)

Dans cette catégorie, on distingue entre les approches basées sur le principe du visiteur (**Visitor-based approach**) et celles basées sur le principe des patrons (**Template-based approach**).

a. Approche basée sur le visiteur (*Visitor-based*) :

Approche de base pour la génération de code, elle consiste à fournir un mécanisme de visiteur pour traverser la représentation interne d'un modèle et créer le code. On peut citer comme exemple le Framework *Jamda* qui fournit un ensemble de classes pour représenter les modèles UML, une API pour manipuler les modèles, et un mécanisme de visiteur pour générer le code [15, 17].

b. Approche basée sur les templates (*Template-based*) :

Actuellement, ce sont les approches les plus utilisées et la majorité des outils MDA disponibles les supportent. La structure d'un Template ressemble au code à générer. On utilise le méta-code du code cible pour accéder aux informations du modèle source. Parmi les outils basés sur ce principe, on peut citer : JET, ArcStyler et Andro MDA (un générateur de code qui se repose notamment sur la technologie ouverte), Optimal J, XDE (les deux derniers sont aussi des transformations de type modèle vers modèle) [15, 17].

1.4.1.2 Transformations de type Modèle vers modèle (*M2M* : model to model) :

Les transformations de type modèle vers modèle consistent à transformer un modèle Source en un modèle cible, ces modèles peuvent être des instances de différents méta-modèles. Elles offrent des transformations plus modulaires et faciles à maintenir. Dans les cas où on trouve un grand espace d'abstraction entre PIMs et PSMs, il est plus facile de générer des modèles intermédiaires qu'aller directement vers le PSM cible [17]. Les modèles intermédiaires peuvent être utiles pour l'optimisation ou bien pour des fins de débogage. De plus, les transformations de type modèle vers modèle sont utiles pour le calcul des différentes vues du système et leur synchronisation [15].

a. Approche de manipulation directe :

Cette approche est basée sur une représentation interne des modèles source et cible, en plus des API pour les manipuler. La combinaison JMI [13] (Java Metadata Interface) et Java sont souvent utilisées dans la mise en œuvre de cette approche [16, 9].

b. Les approches relationnelles :

Cette catégorie regroupe les approches déclaratives et utilise les relations mathématiques. L'idée de base est de spécifier les relations entre les éléments des modèles source et cible par des contraintes, ces derniers peuvent avoir une sémantique exécutable comme dans la programmation logique. Cependant, les relations ne sont pas exécutables mais peuvent être lues dans les deux sens [15].

c. Approches basées sur les transformations de graphes :

Ces approches sont déclaratives, il s'agit d'un formalisme mathématique qui applique la théorie des graphes pour la transformation de modèles, en considérant ces derniers comme des graphes. La stratégie de transformation dans cette approche consiste en un remplacement et une mise en correspondance entre le modèle source et cible, qui s'appuie sur une syntaxe de règles de graphes pour prendre un LHS (Left Hand Side) et le transformer en un RHS (Right Hand Side). La complexité de cette approche réside dans l'aspect non-déterministe de la stratégie d'application des règles de transformations, ce qui implique que les solutions basées sur ce paradigme sont très peu utilisées dans le concret [18].

D'une manière générale, les systèmes de réécriture de graphes combinent une notation graphique et une notation textuelle afin d'exprimer ces transformations [16]. Cette catégorie d'approches est mise en œuvre dans : VIATRA [19], UMLX [20] etATOM3 [21] par exemple [16].

d. Approches hybrides :

Cette approche est la plus récente parmi les autres approches de transformation, elle combine l'approche déclarative et impérative, L'approche déclarative est généralement utilisé pour la définition et la sélection des transformations qui peuvent être appliquées, tandis que l'approche impérative est bien adapté à la description de la stratégie de transformation [18].

e. Approches dirigées par la structure :

Elles offrent une représentation interne d'un modèle et des API (Application Programming Interface) pour les manipuler. Elles sont généralement implémentées comme des cadres structurants orientés objet qui fournissent un ensemble minimal de concepts – sous forme de classes abstraites par exemple [8].

f. Autres approches :

Pour accomplir, deux approches doivent être mentionnées : CWM et XSLT.

- **CWM :** fournit un mécanisme permettant de relier les éléments source et cible, mais la dérivation des éléments cibles doit être implémentée dans un langage concret qui n'est pas prescrit par CWM.
- **XSLT :** technologie standard pour transformer XML [15, 22].

1.4.2 Critères des approches de transformation

Classification des approches de transformation de modèles **huit critères** :

- **les règles de transformation :** les règles de transformations peuvent être présentées de différentes manières : de manière classique sous forme de règles avec une prémisse et une conclusion, mais aussi des fonctions ou des templates peuvent aussi être considérés comme des formes de règles ;
- **Le contrôle d'application des règles :** il s'agit ici de l'ordonnancement des règles et des parties du modèles à traiter ;
- **Organisation des règles :** les structures d'organisation des règles telles que des modules ou des packages ;

- **Relation source-cible :** cela concerne les relations entre les modèles source et cible : dans certains cas le modèle cible est le modèle source sur lequel on a appliqué des modifications tandis que dans d'autres cas les deux modèles sont différents ;
- **Incrémentalité :** cela décrit si l'approche de transformation permet de modifier le modèle cible lorsqu'on modifie le modèle source ;
- **Directionalité :** si la transformation n'est applicable que dans un seul sens elle est considérée comme unidirectionnelle, tandis qu'elle sera considérée comme multidirectionnelle dans le cas contraire ;
- **Traçabilité :** cela concerne les mécanismes permettant d'enregistrer les différentes étapes de l'exécution d'une transformation.

1.4.3 Type de transformation des modèles

Les deux principaux artefacts de l'ingénierie dirigée par les modèles sont les modèles et les transformations de modèles. Rendre les modèles productifs consiste à leur appliquer des transformations pour obtenir des résultats utiles au développement. Un concept clé de l'IDM, consiste à manipuler les modèles à travers des transformations, ces transformations assurent le passage d'un ou plusieurs modèles sources a un ou plusieurs modèles cibles Dans la suite nous citons les différents types de transformation de modèles.

1.4.3.1 Transformation horizontale de modèle

Une transformation de modèle horizontal est une transformation, ou le modèle source et le modèle cible appartiennent au même niveau d'abstraction. Un exemple pour ce type particulier de transformation est le refactoring, ou le modèle cible, par rapport au modèle source, change dans sa structure interne sans changer le comportement. Par exemple le renommage d'un élément dans le modèle ne change rien au comportement du modèle [23] voir figure :

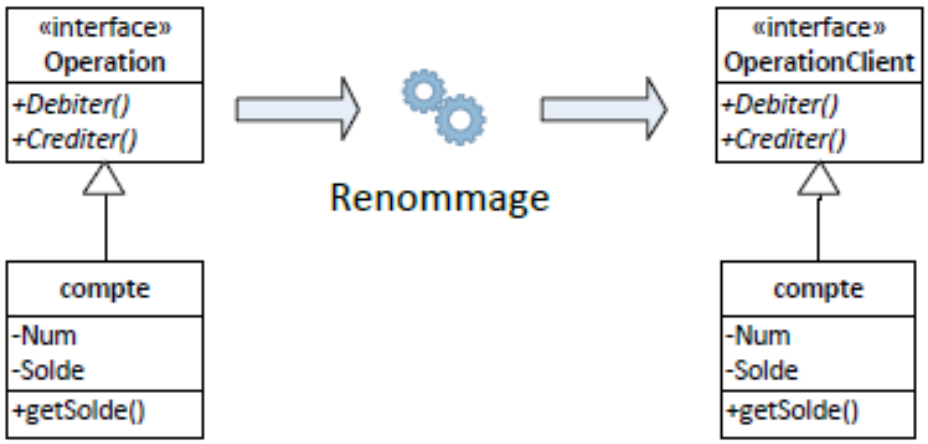


Figure 1.5: Renommage de l'interface opération [23]

1.4.3.2 Transformation verticale de modèle

Contrairement à la transformation de modèle horizontale, différents niveaux d'abstraction sont utilisés dans le modèle source et cible. Cette transformation est appelée aussi raffinement parce que des informations supplémentaires sont ajoutées [23]

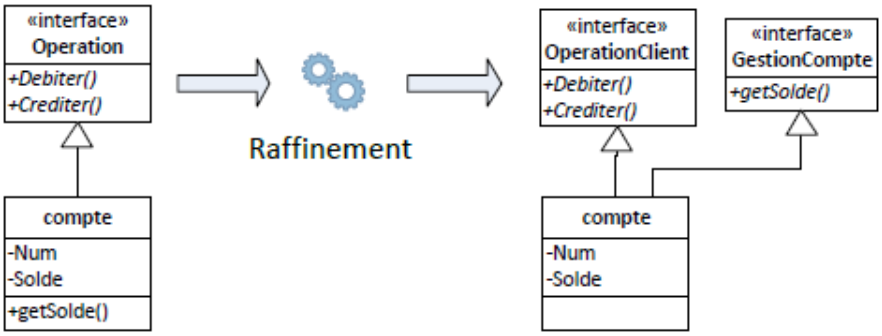


Figure 1.6: Raffinement : Ajout d'interface [23]

1.4.3.3 Transformation endogène

La transformation de modèles est qualifiée d'endogène si les modèles sources et cibles sont conformes au même méta-modèle [9].

Exemples de transformations endogènes, l'optimisation comme la fusion ou l'élimination de code mort ou encore le refactoring [23].

1.4.3.4 Transformations exogènes

Contrairement aux transformations endogènes, les transformations exogènes sont exprimées entre les modèles conformes à des différents méta-modèles. Les transformations endogènes peuvent également être appelées translation. Un exemple pour les Transformations exogènes est la génération de code ou de documentation ou l'ingénierie inverse Par exemple la décompilation. Par exemple, un diagramme de classes UML peut être traduit en Code Java. La traduction d'un code Java en diagramme de classe UML est un exemple pour le Reverse engineering [23].

Notons une convention implicite, les transformations sont toujours nommées par : nom-source2nom-cible (ActivityDiagram2PetriNet).

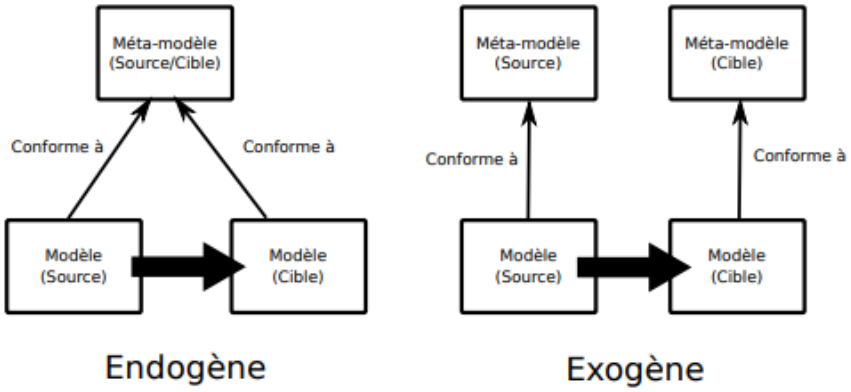


Figure 1.7: Transformation endogène et exogène [24]

1.4.4 Caractéristique des transformations de modèles

- **Une organisation :** on peut les organiser de façon modulaire Par : la réutilisation par le biais de mécanisme d'héritage entre les Règles, la composition par le biais d'un ordonnancement explicite, Ou Enfin, par une structure dépendante du modèle source ou du Modèle cible [17].
- **Ordonnancement et orientation :** il y a deux types d'ordonnancement des règles ; l'ordonnancement implicite qui est défini par l'outil de transformation lui-même, et l'ordonnancement explicite. Il existe des mécanismes permettant de spécifier l'ordre d'exécution des règles [17].
- **Traçabilité :** il permet de renforcer la technique du modèle source qui a été appariés à des éléments du modèle cible. La fonctionnalité de traçage peut être intégrée dans l'outil ou mise en œuvre dans le cadre de la description du mapping. Les traces peuvent être stockées dans le modèle source, dans le modèle cible ou dans un endroit séparé.
  Pour garantir la traçabilité des mappings, les langages de transformations offrent la possibilité de définir un sens d'exécution de la transformation. De ce fait, on distingue :
  - **les langages unidirectionnels** qui interprètent le mapping dans un seul sens (du modèle source vers le modèle cible) ;
  - **les langages bidirectionnels** pour lesquels les notions de modèles source et cible sont relatives au sens d'interprétation du mapping. En effet, un même mapping étant utilisé dans les deux cas d'interprétation, un modèle sera source s'il est en entrée dans l'interprétation du mapping. Dans le cas contraire, il est dit cible.
- **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 ; [25]
- **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 [25].

1.5 L'Architecture Dirigée par les Modèles (MDA)

L'IDM peut être considérée comme un domaine qui a émergé avec les technologies liées à l'instrumentation des modèles. Il existe différentes approches concrétisant différentes façons d'utiliser les modèles dans leur processus de développement des systèmes. L'approche la plus connue et peut-être la plus développée est l'approche MDA. Nous présentons cette approche dans la sous-section suivante.

1.5.1 Présentation générale

En novembre 2000, l'OMG (Object Management Group), initie l'approche MDA (Model Driven Architecture). Cette approche a pour but d'apporter une nouvelle vision unifiée de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique.

Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux couts excessifs de migration technologique [4].

L'initiative MDA a donné lieu à une standardisation des approches pour la modélisation sous la forme d'une structure en 4 niveaux de modélisation (appelée communément Pile de modélisation). La proposition initiale était d'utiliser le langage UML et ses différentes vues comme unique langage de modélisation. Cependant, il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin d'exprimer de nouveaux concepts relatifs à des domaines d'application spécifiques.

Ces extensions devenant de plus en plus importantes, la communauté MDA a élargi son point de vue en considérant les langages de modélisation spécifiques à un domaine (DSML en Anglais). Figure 1.8 donne une vue générale d'un processus [25].

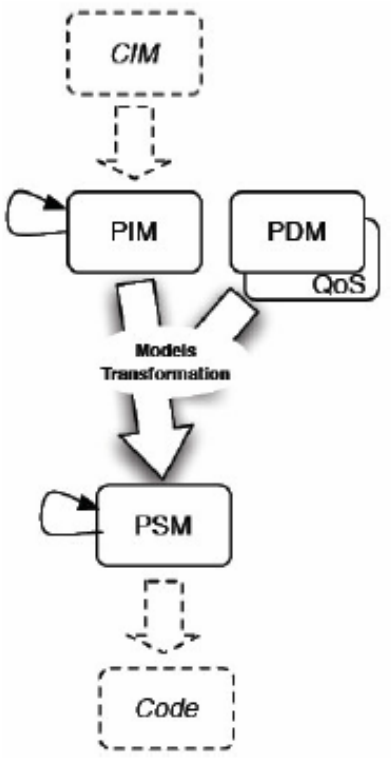


Figure 1.8: Principe du Processus MDA [4]

Le noyau de l'architecture est basé sur les techniques (uml, mof, cwm), autour quelques-unes des plates-formes supportées (Java, Web, Corba, ...), en surface on trouve les services systèmes et enfin les domaines pour lesquels des composants métiers doivent être définis (Domain Facilities), parmi ces domaines on peut citer : E-Commerce, Finance, Télécommunication [15].

1.5.2 Architecture MDA à quatre niveaux

L'OMG a défini une architecture à quatre niveaux d'abstraction, comme carde général pour l'intégration des méta-modèles, en se basant sur l'MOF comme le montre la figure 1.9. Dans cette architecture, les modèles de deux niveaux adjacents sont liés par une relation d'instanciation : [4]

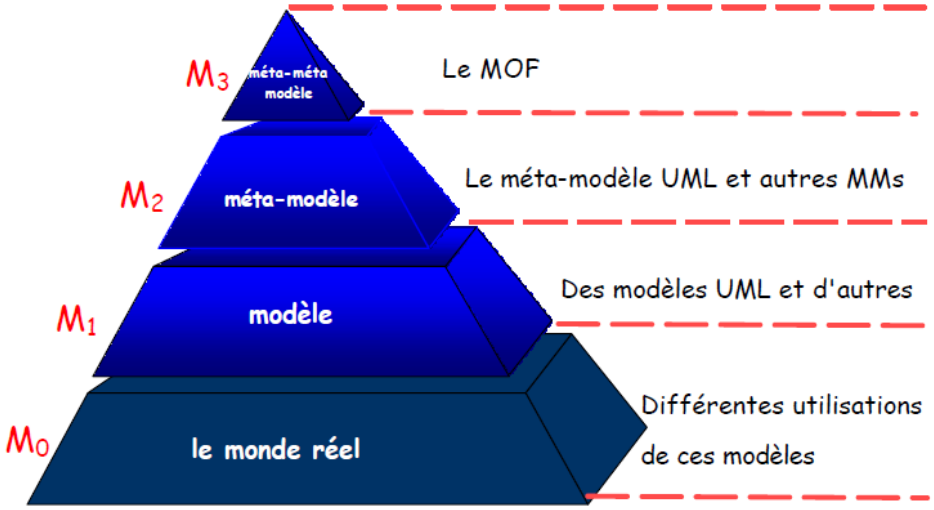


Figure 1.9: Les quatre niveaux d'abstraction pour MDA [26]

- ***Le niveau M0*** **:** qui est le niveau des données réelles, est composé des informations que l'on souhaite modéliser. Ce niveau est souvent considéré comme étant le monde réel [26].
- ***Le niveau M1*** **:** Ce niveau représente toutes les instances d'un méta-modèle. Les Modèles du niveau M1 doivent être exprimés dans un langage défini au niveau M2. UML est un exemple de modèles du niveau M1.
- ***Le niveau M2*** **:** Ce niveau représente toutes les instances d'un méta-méta-modèle. Il est composé de langages de spécifications de modèles d'information. Le méta-modèle UML qui est décrit dans le standard UML et qui définit la structure interne des modèles UML, appartient au niveau M2.
- ***Le niveau M3*** **:** Ce niveau définit un langage unique pour la spécification des Méta-modèles. Le MOF élément réflexif du niveau M3, définit la structure de tous les Méta-modèles du niveau M2 [25].

1.5.3 Les standards de L'OMG

L'OMG a déjà défini plusieurs standards pour le MDA : nous en dressons ici une liste des plus importants.

- **UML (Unified Modeling Language) :** La notation UML est décrite sous forme d'un ensemble de diagrammes. La première génération d'UML (UML 1.x), définit neuf diagrammes pour la spécification des applications. Dans UML 2.0, quatre nouveaux diagrammes ont été ajoutés : il s'agit des diagrammes de structure composite (Composite structure diagrams), les diagrammes de paquetages (Packages diagrammes), les diagrammes de vue d'ensemble d'interaction (Interaction overview diagrams) et les diagrammes de synchronisation (Timing diagrams). Les diagrammes UML sont regroupés dans deux classes principales :
  - **Les diagrammes dynamiques :** regroupent les diagrammes de séquence, les diagrammes de communication (nouvelle appellation des diagrammes de collaboration d'UML), les diagrammes d'activités, les machines à états, les diagrammes de vue d'ensemble d'interaction, et les diagrammes de synchronisation.
  - **Les diagrammes statiques :** regroupent les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, et les diagrammes de paquetages..[4]
- **XMI, (XML Metadata Interchange) :** C'est un format destiné pour l'échange d'informations de métadonnées UML basé sur XML. Il a également été proposé par l'OMG en tant que standard et représente un procédé de sérialisation d'objets MOF permettant de décrire des objets sous forme XML.
- **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 [8].
- **QVT (Query/View/Transformation) :** Il s'agit d'un langage standardisé par l'OMG pour permettre la transformation de modèles. Il propose un cadre formel pour décrire les règles de transformations qui seront appliquées sur les modèles en question. QVT représente le pivot technologique de la transformation de modèle et sa syntaxe s'appuie sur la norme OCL (Object Constraint Language) [18].

- **OCL (Object Constraint Language) :** C'est un langage d'expression de contraintes. Il est utile pour la définition précise d'un langage ou d'une transformation de modèles ; une requête OCL peut spécifier les éléments ciblés par une règle de transformation.OCL étend l'expressivité de UML. Il permet de spécifier :
  - Les valeurs initiales des attributs ;
  - Les règles de dérivation des associations et attributs ;
  - Les destinations des messages envoyés ;
  - Les conditions de garde pour une machine à états finis (FSM) ;
  - Les requêtes de l'utilisateur sur un modèle [27].
- **CWM (Common Warehouse Metamodel) :** (OMG-CMW, 2003) est le standard de l'OMG pour les techniques liées aux entrepôts de données. Il couvre le cycle de vie complet de modélisation, de construction et de gestion des entrepôts de données. CWM définit un métamodèle qui représente les métadonnées aussi bien au niveau métier qu'au niveau technique. CWM définit actuellement les métamodèles des principaux types d'entrepôts de données (Relationnel, Objet, XML,....) et propose des règles de transformation entre ceux-ci. Les métamodèles de données permettent de modéliser des ressources comme les bases de données relationnelles et les bases de données orientées objets

1.5.4 Les différents Modèles de MDA

Le MDA est composé de plusieurs modèles, « descriptions abstraites d'une entité du monde réel utilisant un formalisme donné » qui vont servir dans un premier temps à modéliser l'application, puis par transformations successives à générer du code. Aujourd'hui, la frontière entre les différents modèles n'est pas encore bien explicitée, ni formalisée. Néanmoins, il est possible de donner une description de chacun d'eux [28].

1.5.4.1 Le CIM (Computation Independent Model)

Il est indépendant de tout système informatique. C'est le modèle métier ou le modèle du domaine d'application. Le CIM permet la vision du système dans l'environnement où il opèrera, mais sans rentrer dans le détail de la structure du système, ni de son implémentation. Il aide à représenter ce que le système devra exactement faire. Il est utile, non seulement comme aide pour comprendre un problème, mais également comme source de vocabulaire partagé avec d'autres modèles. L'indépendance technique de ce modèle lui permet de garder tout son intérêt au cours du temps et il est modifié unique-

ment si les connaissances ou les besoins métier changent. Le savoir faire est recentré sur la spécification CIM au lieu de la technologie d'implémentation. Dans les constructions des PIM* (Platform Independent Model) et des PSM* (Platform Specifique Model), il est possible de suivre les exigences modélisées du CIM qui décrivent la situation dans lequel le système est utilisé, et réciproquement.

1.5.4.2 Le modèle PIM (Platform Independent Model)

Les modèles appelés PIM (*Platform Independant Model*), désignés aussi par « modèle d'analyse et de conception abstraite » spécifient la logique métier de l'application et ne doivent pas contenir d'informations sur les technologies qui seront utilisées pour déployer l'application. Même si MDA préconise l'utilisation d'UML pour réaliser les modèles d'analyse et de conception, il n'exclut pas que d'autres langages puissent être utilisés. En effet, l'IDM, au contraire, favorise la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Languages – DSL*) offrant ainsi aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Dans ce contexte, l'OMG a proposé des langages standards dédiés à des domaines d'applications, tel que EDOC (*Enterprise Distributed Object Computing*) pour les systèmes distribués à base de composants, CWM pour les bases données, etc. A ce titre, nous proposons à travers cette thèse un DSL pour la modélisation d'interfaces d'application multimodales et adaptatives. L'objectif est d'aboutir à un langage de description permettant de décrire les différents aspects d'une interface d'application : aussi bien graphique qu'interactif, pouvant s'adapter aux différents dispositifs d'interaction existants. [29]

1.5.4.3 Le modèle PSM (Platform Specific Model)

Cette phase de l'approche MDA concerne la génération de code. Elle consiste en l'élaboration du modèle PSM (Platform Specific Model), appelé aussi le modèle de code ou de conception concrète. MDA considère que le code final d'application peut être facilement obtenu à partir du PSM. En effet, le code d'une application se résume à une suite de lignes textuelles, alors qu'un modèle de code est plutôt une représentation structurée. Une caractéristique importante des modèles de code est qu'ils intègrent les concepts des plateformes d'exécution. Pour élaborer des modèles de code, MDA propose, entre autres, l'utilisation des profils UMl .Un profil UML est une adaptation du langage UML à un domaine particulier. [29]

1.5.4.4 PDM (Platform Description Model)

Ce modèle est désigné par l'acronyme PDM pour Platform Description Model. Il correspond à un modèle de transformation du PIM vers un PSM d'implémentation. L'architecte doit choisir une ou plusieurs plates-formes pour l'implémentation du système avec les qualités architecturales désirées. Ce modèle propre à la plate-forme est utile pour la transformation du PIM en PSM. La démarche MDA est ainsi basée sur le détail des modèles dépendant de la plate-forme. Il représente les particularités de chaque plate-forme. Il devrait être fourni par le créateur de la plate-forme, [25].

1.5.5 La transformation de modèle au niveau de l'architecture MDA

Le MDA identifie plusieurs transformations pendant le cycle de développement. Il est possible de faire quatre types de transformations différentes. [28]

1.5.5.1 De PIM vers PIM

Ces transformations sont utilisées pour enrichir, filtrer ou spécialiser les informations des modèles sans rajouter aucune information liée à la plate-forme. Un exemple de transformation PIM vers PIM est démasquer des éléments afin de s'abstraire des détails fonctionnels. Un autre exemple est le passage du modèle d'analyse à celui de conception. Cependant, ces transformations ne sont pas toujours automatisables. Le fait de passer d'un PIM à un autre PIM est appelé raffinement. Ce processus consiste à introduire des détails supplémentaires dans le modèle. Il est aussi utilisé pour le passage de PSM [28].

1.5.5.2 De PIM vers PSM

Un fois le PIM suffisamment raffiné pour pouvoir être spécialisé vers une plate-forme donnée, il peut alors être transformé en PSM. Cette opération consiste à ajouter au PIM des informations propres à une plate-forme technique. Les principales plates-formes visées sont J2EE, .NET ou CORBA, ... C'est le PDM,) ou le MOF (Meta-Object Facility) qui contient les caractéristiques de transformation. Il est alors possible de passer d'un modèle indépendant à un modèle dépendant. Les règles de transformation devront être généralisées et capitalisées pour obtenir dans le futur une automatisation importante [28].

1.5.5.3 De PSM vers PSM

Une transformation PIM vers PSM n'est pas toujours suffisante pour permettre la génération de code d'où la nécessité de passer de PSM à PSM en utilisant des formalismes intermédiaires. Par exemple, pour générer un code C++, à partir d'un formalisme en UML, un passage d'UML vers SDL* (Specification and Description Language) puis de SDL vers C++ pourrait être utilisé. La transformation PSM à PSM (raffinement) s'effectue lors de phases de déploiement, d'optimisation ou de reconfiguration.

1.5.5.4 De PSM vers PIM

Cette transformation est utilisée pour revenir à un modèle indépendant de plate-forme (PIM) à partir d'un modèle spécifique de plate-forme (PSM) ou éventuellement du code. C'est une opération de rétro ingénierie (reverse engineering) qui est assez complexe à réaliser et difficilement automatisable. Ces transformations sont néanmoins nécessaires pour permettre l'intégration d'applications existantes dans le processus MDA. [28]

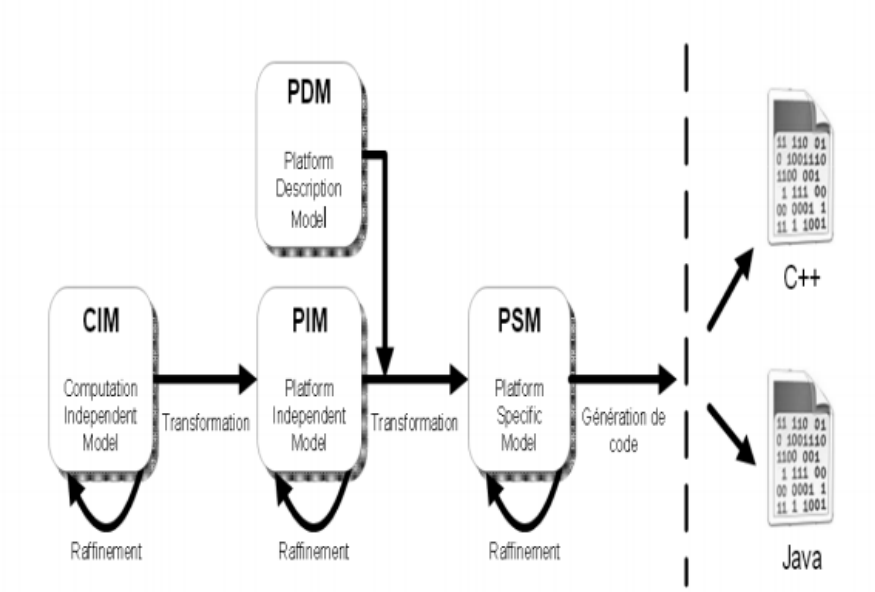


Figure 1.10: Transformations des modèles du MDA [1]

1.6 La transformation des graphes

Les transformations de graphes [30] ont évolué dans la répercussion à l'imperfection dans l'expressivité des approches de réécriture classique comme les grammaires de Chomsky et la réécriture de termes pour s'y prendre avec les structures non linéaires.

Une transformation de graphe d'après **Karsai [31], Andries [32], Rozenberg [30]** Consiste en l'application d'une règle à un graphe et itérer ce processus. Chaque application de règle transforme un graphe par le remplacement d'une de ses parties par un autre graphe. Autrement dit, la transformation de graphe est le processus de choisir une règle d'un ensemble indiqué, appliquer cette règle à un graphe et réitérer le processus jusqu'à ce qu'aucune règle ne puisse être appliquée. La transformation de graphe est spécifiée sous forme d'un modèle de grammaires de graphes. Ces dernières sont une généralisation des grammaires de Chomsky pour les graphes. Elles sont composées de règles. Une règle est constituée de deux parties, le *Left Hand Side (LHS)* et le *Right Hand Side (RHS)* [33].

Le LHS est la partie gauche de la règle, destinée à être mise en concordance avec les parties du graphe (appelé *host graph*) où on veut appliquer la règle. La partie droite de la règle, Le RHS, décrit la modification qui sera effectuée sur le *host graph*, elle substitue dans le *host graph* la partie identifiée par la partie gauche de la règle .

Il existe plusieurs formalismes pour représenter les règles de transformations. Dans ce manuscrit, nous utilisons des règles qui sont exprimées visuellement. La Figure 1.11 montre le principe général de l'application d'une règle sur un graphe. [34]

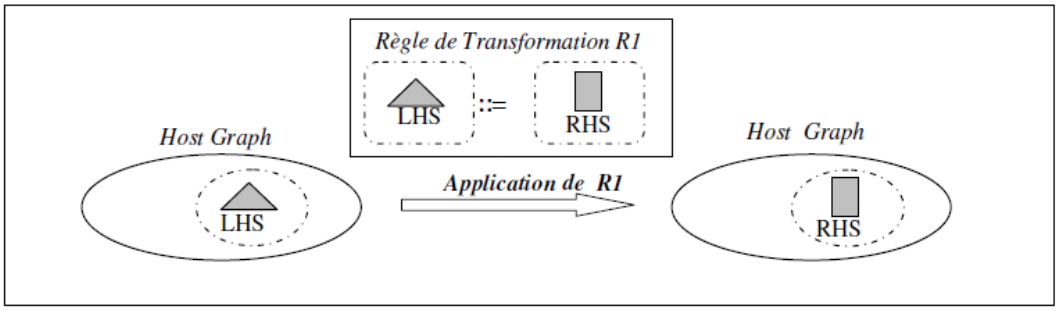


Figure 1.11: Principe de l'application d'une règle [34]

1.6.1 Notion de graphe

Il existe deux types de graphes : les graphes non orientés et les graphes orientés.

- **Graphes non orientés :** Un **graphe** est constitué de **sommets (nœuds)** qui sont reliés par des **arêtes**. Deux sommets reliés par une arête sont **adjacents**. Le nombre de sommets présents dans un graphe est appelé **ordre du graphe**.
- **Graphes orientés :** Graphe orienté est un graphe dont les arêtes sont orientées : on parle alors de l'origine et de l'extrémité d'une arête. Dans un graphe orienté une arête est dénommée **arc**.
- **Graphe étiqueté :** c'est un graphe orienté, dont les arcs possèdent des étiquettes. Si toutes les étiquettes sont des nombres positifs, on parle de **graphe pondéré** [33, 34].

1.6.2 Grammaires de graphes

Andries définis Une grammaire de graphe généralement 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. [35, 33]

1.6.2.1 Le principe de règles

Une règle de transformation de graphe est définie par :
$R = (LHS, RHS, K, glue, emb, cond)$

- LHS graphe de partie gauche.
- RHS graphe de partie droite.
- Un sous graphe K de LHS.

- Une occurrence *glue* de K dans RHS qui relie le sous graphe avec le graphe de partie droite.
- Une relation d'enfoncement *emb* qui relie les sommets du graphe de la partie gauche et ceux du graphe de la partie droite [33, 34, 35].

1.6.2.2 Application des règles

Un ensemble *cond* qui indique les conditions d'application de la règle.

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

1. Choisir une occurrence du graphe de partie gauche LHS dans G.
2. Vérifier les conditions d'application d'après *cond*.
3. Retirer l'occurrence de 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. 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.
5. 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*.

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.

Plusieurs formalismes permettent de représenter les règles de transformation.

1.6.2.3 Système de transformation de graphe

D'après **Rozenberg** [30] Un *système de transformation de graphe* est défini comme un système de réécriture de graphes qui applique les règles de la Grammaire de Graphes sur son graphe initial jusqu'à ce que plus aucune règle ne soit applicable [34, 35].

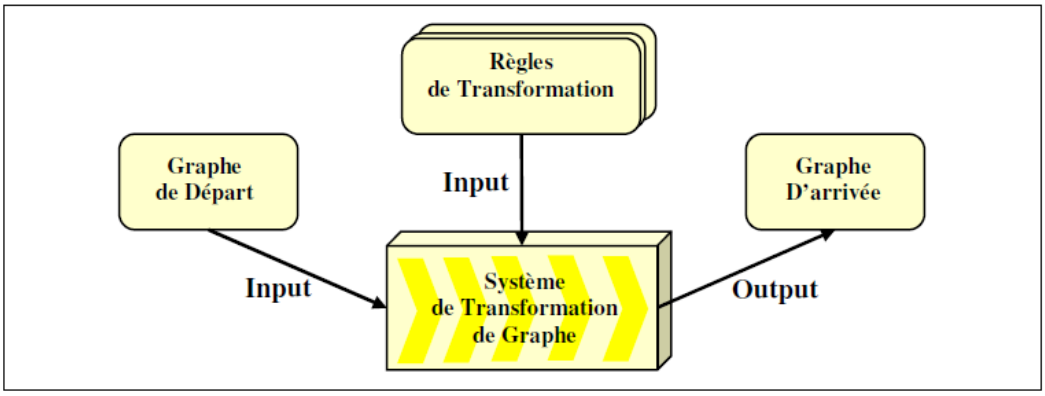


Figure 1.12: Système de réécriture de graphes [34]

Cette approche de transformations de modèles a plusieurs avantages par rapport aux autres Approche :

- Les grammaires de graphes sont un formalisme naturel, visuel, formel et de haut niveau pour décrire les transformations.
- Les fondements théoriques des systèmes de la réécriture de graphes permettent d'aider à vérifier certaines propriétés des transformations telles que la terminaison ou la correction.

1.6.3 Outils de transformation de graphes

Il existe plusieurs systèmes implantant des systèmes de réécriture de graphes. Dans cette section, nous présentons quelques outils actuels de transformation de graphes :

EMF (Eclipse Modeling Framework) : Le projet EMF est un plugin Eclipse. C'est une plateforme de modélisation et de génération de code pour la construction d'outils et d'applications basés sur un modèle de données structurées. À partir d'un modèle décrit dans la spécification XMI.

ATL (Langage de Transformation ATLAS) : ATL est un outil et un langage de transformation de modèles sous l'environnement Eclipse. Dans le cadre de l'Ingénierie Dirigée par les Modèles (IDM), ATL propose une méthode pour produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources. Le programme de transformation est composé de règles qui définissent la manière dont les éléments du modèle source sont traités et analysés afin d'obtenir des éléments du modèle cible [35].

GMF (Graphical Modeling Framework) : un framework au sein de la plateforme Eclipse. Il fournit un composant génératif et une infrastructure d'exécution pour le développement d'éditeurs graphiques basés sur Eclipse Modeling Framework (EMF) et Graphical Editing Framework (GEF). Le projet vise à fournir ces composants, en plus d'outils exemplaires pour sélectionner des modèles de domaine qui illustrent ses capacités.

VIATRA2 (Visual Automated model transformations) : C'est un plugin Eclipse développé en 2005 à Budapest. Le logiciel utilise un langage de modélisation particulier pour représenter les modèles appelé VPM. Le développeur utilise des motifs récursifs et des motifs de négations représentant les conditions d'application négatives pour définir les règles de transformation.

AToM3 : AToM3 [36] est un outil visuel pour la modélisation et la méta-modelisation multi formalismes. Comme il a été implémenté en Python, il peut être exécuté, sans aucun changement, sur toutes les plateformes où un interpréteur de Python est disponible (Linux, Windows et MacOs) [33].

1.7 Conclusion

Dans ce chapitre, nous avons présenté les principes et les concepts de base de l'Ingénierie Dirigée par les Modèles (IDM) qui permet une amélioration significative du développement logiciel et la transformation. Nous nous sommes intéressés beaucoup plus sur l'Architecture Dirigée par les Modèles (ADM) en introduisant les modèles et le formalisme du modèle, les méta-modèles, nous avons présenté par la suite la transformation des modèles et enfin la transformation de graphe.

Chapitre 2

Modélisation semi-formelle avec UML 2.0

2.1 Introduction

Actuellement, l'UML est devenu un standard largement accepté dans l'industrie de développement de logiciels orienté objet. Certains diagrammes d'UML sont utilisés pour modéliser la structure d'un système, d'autres sont utilisés pour modéliser son comportement.

L'approche orientée objet considère le logiciel comme une collection d'objets dissociés et identifiés, définis par des propriétés. Une propriété est soit un attribut soit une entité élémentaire comportementale de l'objet. La fonctionnalité du système émerge alors de l'interaction entre les différents objets qui le constituent. L'une des particularités de cette approche est qu'elle encapsule les données et leurs traitements associés au sein d'un unique objet. Un objet est caractérisé par plusieurs notions, et c'est dans ce contexte que nous présentons quelques éléments du langage UML 2.0 (UnifiedModelingLanguage), qui s'est imposé comme un standard que rencontrent tous les développeurs dans l'industrie du génie logiciel.

Dans ce chapitre, nous rappellerons quelques notions et quelques concepts sur la modélisation et UML, puis nous intéressons particulièrement aux diagrammes d'activités, nous présentons leurs intérêts et nous donnons une description détaillée à ses composants.

2.2 La modélisation

La modélisation d'un système est le processus de développement des modèles. Selon **MarvinL** et **Minsky** [37], un modèle est une représentation abstraite qui contient un ensemble restreint d'informations sur un système réel et un point de vue différent ou perspective de ce système. 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. En informatique la modélisation est vue comme une séparation entre les différent besoins fonctionnels et non fonctionnels (tels que : la sécurité, la

fiabilité, l'efficacité, la performance, la flexibilité, ..., etc.).

Par ailleurs La modélisation d'un système est venue à signifier ce qui représente un système en utilisant une sorte de notation graphique, qui est presque toujours basée sur des notations dans UML [17].

2.3 Les types de modélisation

La modélisation peut se classer selon le degré du formalisme des langages ou des méthodes employées dans le processus de la modélisation. La modélisation peut être considérée comme étant formelle, semi-formelle ou informelle. La table de la figure 2.1 ci-dessous présente une définition des catégories de langages ainsi que des exemples de langages ou de méthodes qui l'utilisent.

La modélisation peut se classer selon le degré du formalisme des langages ou des méthodes employées dans le processus de la modélisation. La modélisation peut être considérée comme étant formelle, semi-formelle ou informelle La table de la figure 2.1 ci-dessous présente une définition des catégories de langages ainsi que des exemples de langages ou de méthodes qui l'utilisent [17].

2.3.1 Modélisation Informelle

Le processus de modélisation informelle à base d'un langage informel, se justifie selon **Celso** [38] pour plusieurs raisons :

- La facilité de compréhension d'un langage permet des consensus entre les personnes qui spécifient et celles qui commandent un logiciel.
- Elle représente une manière familière de communication entre personnes. Le caractère informel de cette approche rend difficile toute tentative de standardisation. D'un autre côté, l'utilisation d'un langage informel rend la modélisation imprécise et parfois ambigüe.

2.3.2 Modélisation semi-formelle

Le processus de modélisation semi-formelle est basé sur un langage textuel ou graphique pour lequel une syntaxe précise est définie **Celso** [38] avec une sémantique. La sémantiqued'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 puissance expressive du modèle graphique est utilisée dans la plupart des méthodes

de modélisation semi-formelles. Par ailleurs, la modélisation semi-formelle s'appuie sur des langages graphiques, tels que : UML qui permet la production de modèles assez faciles à interpréter.

2.3.3 Modélisation formelle

La modélisation formelle est un processus de développement rigoureux basé sur des notations formelles avec une sémantique précise, ainsi que sur des vérifications formelles. Le principal avantage des spécifications formelles est leur capacité à exprimer une signification précise, en permettant de cette manière des vérifications de la cohérence et de la complétude d'un système. Par exemple J. P. Bowen et **M. C. Hinchey** [39] montrent qu'avec une traduction appropriée, les méthodes formelles peuvent aider à la compréhension d'un système par un utilisateur.

Catégories de Langages			
Langage Informel		**Langage Semi-Formel**	**Langage Formel**
Simple	*Standardisé*		
Langage qui n'a pas un ensemble complet de règles pour restreindre une construction	Langage avec une structure, un format et des règles pour la composition d'une construction.	Langage qui a une syntaxe définie pour spécifier les conditions sur lesquelles les constructions sont permises.	Langage qui possède une syntaxe et une sémantique définies rigoureusement. Il existe un modèle théorique qui peut être utilisé pour valider une construction.
Exemples de Langages ou Méthodes			
Langage Naturel.	Texte Structuré en Langage Naturel.	Diagramme Entité-Relation, Diagramme à Objets.	Réseaux de Petri, Machines à états finis, VDM, Z.

Figure 2.1: Classification et utilisation de langages ou de méthodes [17]

2.4 Une notion sur la programmation orientée objet

La programmation orientée objet consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel (que l'on appelle domaine) en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objet. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, la couleur, ...).

L'approche objet est une idée qui a désormais fait ses preuves. Simula a été le premier langage de programmation à implémenter le concept de classes en 1967 ! En 1976, Smalltalk implémente les concepts d'encapsulation, d'agrégation, et d'héritage (les principaux concepts de l'approche objet). D'autre part, de nombreux langages orientés objets ont été mis au point dans un but universitaire (Eiffel, Objective C, Loops...). Un objet représente une entité du monde réel, ou de monde virtuel dans le cas d'objets immatériels, qui se caractérisent par une identité, des états significatifs et par un comportement. L'identité d'un objet permet de distinguer les objets les uns par rapport aux autres. Son état correspond aux valeurs de tous les attributs à un instant donné. Ces propriétés sont définies dans la classe d'appartenance de l'objet. Enfin, le comportement d'un objet se défini par l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés (un message = demande d'exécution d'une opération) par les autres objets. Ces opérations sont définies dans la classe d'appartenance de l'objet. A partir de cette année, Rumbaugh et Booch (rejoints en 1995 par Jacobson) ont unis leurs efforts, pour mettre au point la méthode unifiée (UnifiedMethod 0.8), incorporant les avantages de chacune des méthodes précédentes. La méthode unifiée à partir de la version 1.0 devient UML, soumis à l'OMG (Object Management Group) en Janvier 1997, et acceptée en novembre 1997 dans sa version 1.1, date à partir de laquelle UML devient un standard international. La version qui à vue nos jours est la version 2.0 d'UML [40].

2.4.1 Intérêt d'une méthode objet

Les langages orientés objet constituent chacun une manière spécifique d'implémenter le paradigme objet. Ainsi, une méthode objet permet de définir le problème à haut niveau sans rentrer dans les spécificités d'un langage. Il représente ainsi un outil permettant de définir un problème de façon graphique, afin par exemple de le présenter à tous les acteurs d'un projet (n'étant pas forcément des experts en un langage de programmation). De plus, le fait de programmer à l'aide d'un langage orienté objet ne

fait pas d'un programmeur un concepteur objet. En effet il est tout à fait possible de produire un code syntaxiquement juste sans pour autant adopter une approche objet. Ainsi la programmation orientée objet implique [17].

- en premier lieu une conception abstraite d'un modèle objet (c'est le rôle de la méthode objet)
- en second plan l'implémentation à l'aide d'un langage orienté objet (tel que C++/Java/...)

2.5 Notion sur l'UML

2.5.1 Définition

Tout à fait simplement, l'UML « UnifiedModelingLanguage » ou « langage de modélisation unifié » est une langue visuelle pour modéliser et communiquer au sujet de systèmes à travers l'usage de diagrammes et texte secondaire.

UML n'est pas une méthode ou un processus, est un langage de modélisation objet, UML a été adopté par toutes les méthodes Objet [4].

2.5.2 Historique

Durant les années 70 et 80, on comptait tout au plus dix langages de spécification d'applications orientées objet. Afin d'enrayer la multiplication des langages de spécification, G. Booch et J. Rumbaugh ont décidé en 1994 d'unifier leurs méthodes respectives OOADA (Object-Oriented Analysis and Design with Applications) et OMT (Object Management Technique) au sein de la société Rational Software Corporation. La première version de ces travaux est sortie en octobre 1995 sous le nom d'UML0.8. Après 1995, l'initiative de Rational Software Corporation a intéressé d'autres industriels, qui ont vite compris les avantages qu'ils pouvaient tirer de l'utilisation d'UML. C'est ainsi qu'une RFP (Request For Proposal) a été émise à l'OMG en 1996 pour la standardisation d'UML. Des débuts de l'initiative UML jusqu'à la version 1.4 élaborée par l'OMG, l'objectif fondamental était de résoudre le problème de l'hétérogénéité des spécifications. L'approche envisagée a consisté à proposer un langage unifiant tous les langages de spécification. La version 2.0 vise à faire entrer ce langage dans une nouvelle ère en faisant en sorte que les modèles UML soient au centre de MDA. Cela revient à rendre les modèles UML pérennes et productifs et à leur permettre de prendre en compte les plates-formes d'exécution [4].

La figure 2.2 ci-dessous, présente l'évolution des versions d'UML.

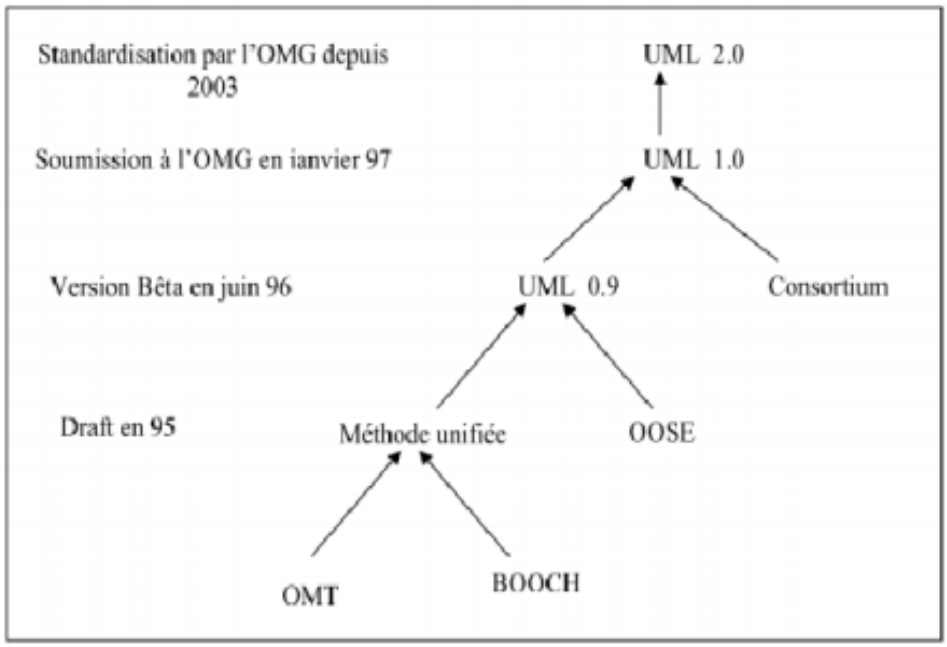


Figure 2.2: Évolution des versions d'UML [17]

2.5.3 A quoi sert UML ?

UML utilise l'approche objet en présentant un langage de description universel. Il permet grâce à un ensemble de diagrammes très explicites, de représenter l'architecture et le fonctionnement des systèmes informatiques complexes en tenant compte des relations entre les concepts utilisés et l'implémentation qui en découle [1].UML est avant tout un support de communication performant, qui facilite la représentation et la compréhension de solutions objet :

- Sa notation graphique permet d'exprimer visuellement une solution objet, ce qui facilite la comparaison et l'évaluation de solutions.
- L'aspect formel de sa notation, limite les ambiguïtés et les incompréhensions.
- Son indépendance par rapport aux langages de programmation, aux domaines d'application et aux processus, en fait un langage universel.

UML est donc bien plus qu'un simple outil qui permet de "dessiner" des représentations mentales... Il permet de parler un langage commun, normalisé mais accessible, car visuel. Il représente un juste milieu entre langage mathématique et naturel, pas trop

complexe mais suffisamment rigoureux, car basé sur un méta modèle. Une autre caractéristique importante d'UML, est qu'il cadre l'analyse. UML permet de représenter un système selon différentes vues complémentaires : les diagrammes [40].

2.5.4 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 2.3, pour avoir une idée utilisable en pratique sans risque d'erreur grave [35].

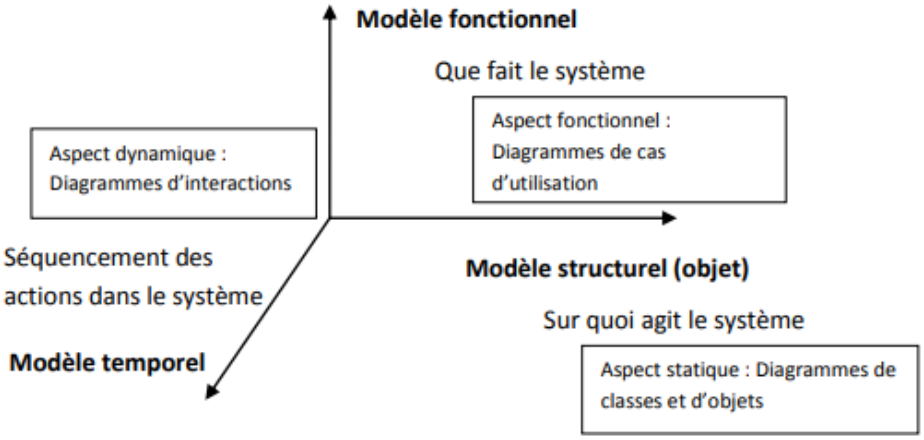


Figure 2.3: Les aspects d'un système [35]

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 :

1. **La vue structurelle, ou statique :**

   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 paquetages.
- Diagramme de structures composites.

2. **La vue comportementale :**

   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'interblocage.

3. **La vue dynamique ou d'interaction :**

   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 2.4 montre un concept UML 2.0 avec le lien entre les différentes vues et les abstractions abstractions.

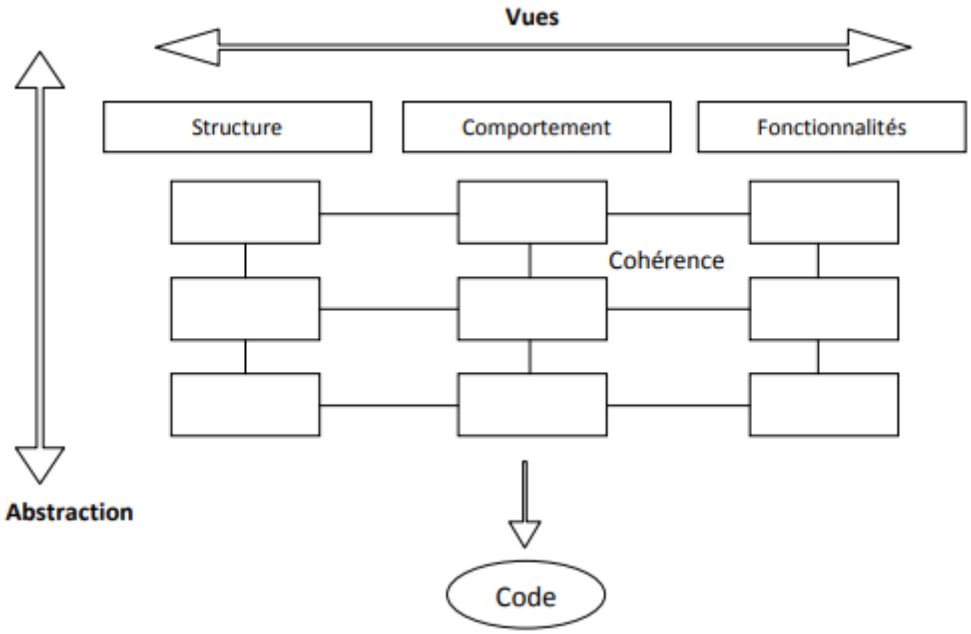


Figure 2.4: Différentes vues dans un concept UML 2.0 [35]

Dans notre projet on s'intéresse seulement au diagramme d'activités, pour cela la section suivante sera consacrée au diagramme d'activités.

2.6 Les Diagrammes d'Activités

2.6.1 Définition

UML permet de représenter graphiquement les aspects dynamiques des systèmes à l'aide de plusieurs diagrammes comportementaux. Parmi eux, on distingue les diagrammes d'activité qui permettent de mettre l'accent sur les traitements. Ils sont utilisés pour représenter le comportement d'une méthode ou le déroulement d'un cas d'utilisation [15, 9].

Les diagrammes d'activités sont relativement proches des diagrammes d'états-transitions dans leur présentation, mais leur interprétation est sensiblement différente. Les diagrammes d'états-transitions sont orientés vers des systèmes réactifs, mais ils ne donnent pas une vision satisfaisante d'un traitement faisant intervenir plusieurs classeurs et doivent être complétés, par exemple, par des diagrammes de séquence. Au contraire, les diagrammes d'activités ne sont pas spécifiquement rattachés à un classeur particulier. On peut attacher un diagramme d'activités à n'importe quel élément de modélisation

afin de visualiser, spécifier, construire ou documenter le comportement de cet élément [41].

Les diagrammes d'activités d'UML constituent un outil de modélisation des systèmes workflows, des modèles orientés service et des processus métiers (ils montrent l'enchaînement des activités qui concourent au processus). Un modèle d'activité consiste en activités liéespar des flux de données et de contrôle. Une activité peut varier d'une tâche humaine à une tâche complètement automatisée [15, 9].

La différence principale entre les diagrammes d'interaction et les diagrammes d'activité, est que les premiers modélisent le flot de contrôle entre objets, alors que les seconds sont utilisés pour modéliser le flot de contrôle entre activités [41, 9].

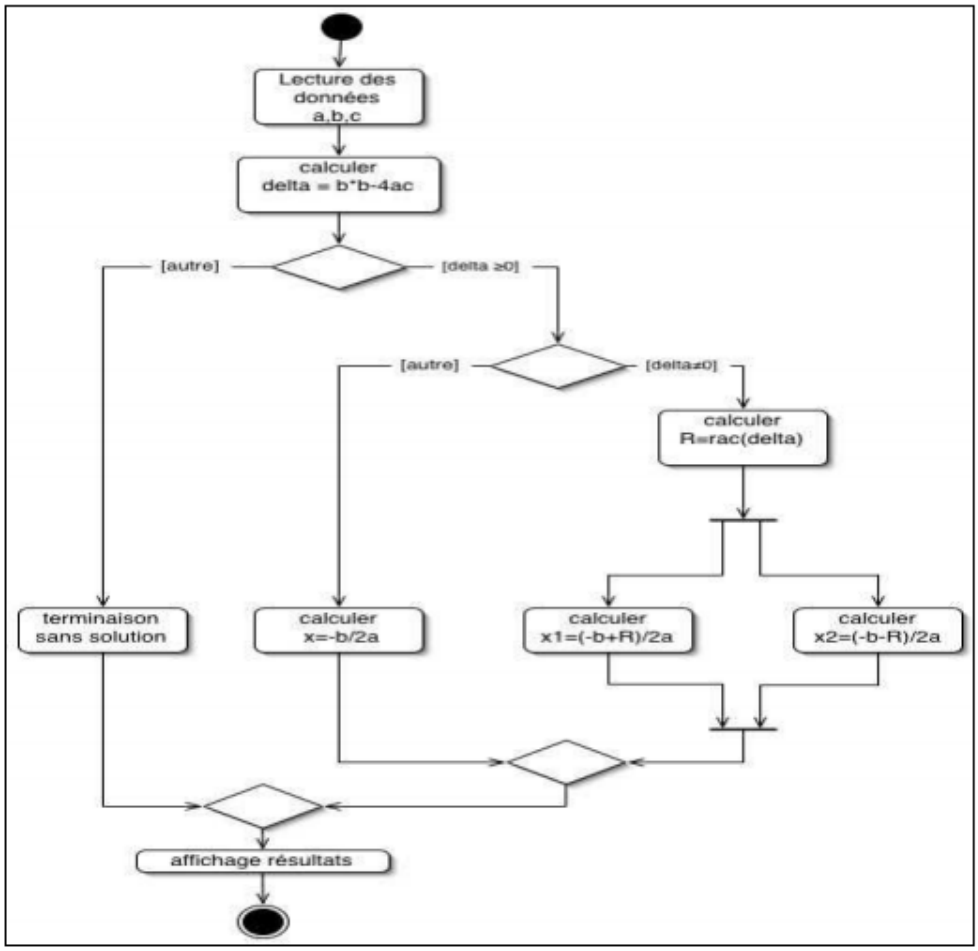


Figure 2.5: Exemple de diagramme d'activités [15]

2.6.2 Utilisation courante

Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisation. Plus précisément, ils viennent illustrer et consolider la description textuelle des cas d'utilisation De plus, leur représentation sous forme d'organigrammes les rend facilement intelligibles et beaucoup plus accessibles que les diagrammes d'états-transitions. On parle généralement dans ce cas de modélisation de workflow. On se concentre ici sur les activités telles que les voient les acteurs qui collaborent avec le système dans le cadre d'un processus métier. La modélisation du flot d'objets est souvent importante dans ce type d'utilisation des diagrammes d'activités.

Les diagrammes d'activités permettent de spécifier des traitements a priori séquentiels et offrent une vision très proche de celle des langages de programmation impératifs comme C++ ou Java. Ainsi, ils peuvent être utiles dans la phase de réalisation, car ils permettent une description si précise des opérations qu'elle autorise la génération automatique du code. La modélisation d'une opération peut toutefois être assimilée à une utilisation d'UML comme langage de programmation visuelle, ce qui n'est pas sa finalité. Il ne faut donc pas y avoir recours de manière systématique, mais la réserver à des opérations dont le comportement est complexe ou sensible [41, 42].

2.6.3 Notation de diagrammes d'activité

Nous présentons dans cette sous section seulement quelques éléments de modélisation de base des diagrammes d'activité [42].

2.6.3.1 Action

Une action est l'unité fondamentale de la spécification de comportement. L'action prend un ensemble d'entrées et les convertit en un ensemble de résultats, l'une ou les deux ensembles peuvent être vide. Une action ne peut être décomposée en actions plus simples. Une action peut être, par exemple :

- une affectation de valeur à des attributs
- la création d'un nouvel objet ou lien
- l'émission d'un signal
- la réception d'un signal

La notation d'une action est un rectangle avec des coins arrondis.

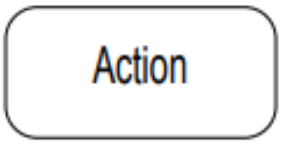


Figure 2.6: Notation d'action [42]

2.6.3.2 Nœud d'action

Un nœud d'action est un nœud d'activité exécutable qui constitue l'unité fondamentale de fonctionnalité exécutable dans une activité. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé. Les actions sont généralement liées à des opérations qui sont directement invoquées. Un nœud d'action doit avoir au moins un arc entrant. Graphiquement, un nœud d'action est représenté par un rectangle aux coins arrondis qui contient sa description textuelle. Cette description textuelle peut aller d'un simple nom à une suite d'actions réalisées par l'activité. UML n'impose aucune syntaxe pour cette description textuelle, nous pouvons donc utiliser une syntaxe proche de celle d'un langage de programmation particulier ou du pseudo-code. Certaines actions de communication ont une notation spéciale comme montré ci-contre :

Dans l'exemple ci-contre, nous détectons l'arrivée du train ; cette action représente l'action "accepte vent" c'est-à-dire que le système reçoit le signal de l'arrivée du train. Deuxièmement, "faire clignoter les feux" est une action "send signal", cela veut dire que le système envoie un signal qui est transmis à un objet cible sans attendre que ce dernier ait bien reçu ce signal. Ensuite, l'action "time event" est un événement temporel déclenché après l'écoulement d'une certaine durée. Enfin, "abaisser la barrière" est une action "send signal", un message est envoyé et transmis à la cible.

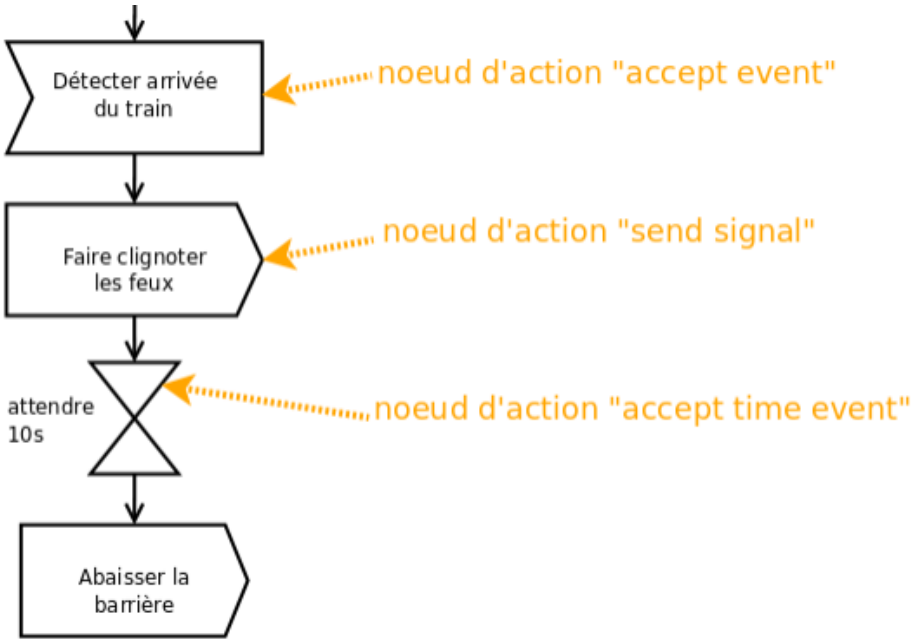


Figure 2.7: Représentation particulière des noeuds d'action de communication [41]

2.6.3.3 Activité (activity)

Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions. Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions). Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

Une activité est un comportement (behavior en anglais) et à ce titre peut être associée à des paramètres [41, 15].

2.6.3.4 Groupe d'activités (activity group)

Un est une activité regroupant des nœuds et des arcs. Les nœuds et les arcs peuvent appartenir à plus d'un groupe. Un diagramme d'activités est lui-même un groupe d'activités

2.6.3.5 Un nœud d'activité

Un nœud d'activité est un type d'élément abstrait permettant de représenter les étapes le long du flot d'une activité [43]. Un nœud d'activité peut être l'exécution d'un comportement subordonné, comme un calcul arithmétique, un appel à une opération,

ou la manipulation du contenu d'un objet. Les nœuds d'activité comprennent également le flux de contrôle des constructions, tel que la synchronisation, la décision et la concurrence. Il existe trois types de nœuds d'activités [15] :

- le nœud de contrôle (control nodes).
- les nœuds objets (objectnode).
- les nœuds d'exécutions (executable node).

La figure ci-dessous représente graphiquement les nœuds d'activité.

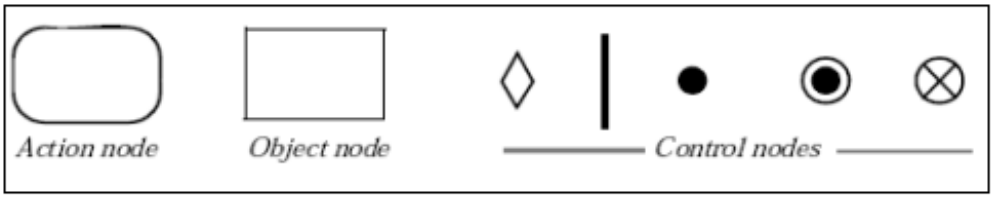


Figure 2.8: Notation nœuds d'activité [9]

- **Représentation graphique des nœuds d'activité :** De la gauche vers la droite, nous trouvons : le nœud représentant une action, qui est une variété de nœud exécutable, un nœud objet, un nœud de décision ou de fusion, un nœud de bifurcation ou d'union, un nœud initial, un nœud final et un nœud final de flot. [41]

2.6.3.5.1 Les nœuds de contrôle (control nodes)

On utilise les nœuds de contrôle pour guider le flux de contrôle (et le flux d'objets) à travers un groupe d'activités et d'actions. Les nœuds de contrôle viennent dans une variété de formes, en fonction de ce qu'on a besoin, ils servent comme un agent de circulation pour le flux de contrôle et les flux d'objets. Les nœuds de contrôle sont les suivants : [42]

- **Initial :** Un nœud initial est l'endroit où le flux de contrôle commence quand une activité est invoquée. La Figure 2.9 montre la notation d'un nœud initial.

Figure 2.9: Notation de nœud initial [42]

- **Final :** Un nœud final est un nœud de contrôle dans laquelle un ou plusieurs flux au sein d'une activité donnée s'arrêtent. Il existe deux types de nœuds finaux :

  (a) nœud de flux final.

  (b) nœud d'activité final.

  La figure 2.10 montre la notation de nœud final :

(a) Nœud d'activité final (b) Nœud de flux final

Figure 2.10: Notation de nœud final [42]

- **Décision :** offre un choix entre deux ou plusieurs activités sortantes, dont chacune a une expression booléenne qui doit résoudre à Vrai avant la prise de chemin. Un nœud de décision apparaît comme un diamant, comme le montre la figure 2.11.

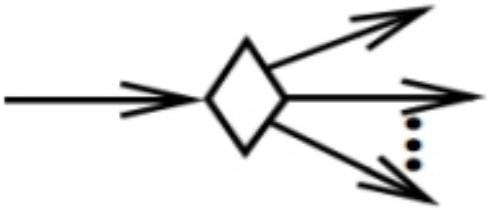


Figure 2.11: Notation de nœud de décision [42]

- **Fusion (Merge) :** Un nœud de fusion est un nœud de contrôle, il rassemble plusieurs flots alternatifs entrants en un seul flot sortant. L'utilité de ce nœud n'est pas pour synchroniser des flux concurrents mais pour accepter un flux (en sortie) parmi plusieurs flux entrants. [15].
- **Nœud de bifurcation ou de débranchement (forknode) :** Divise un flux en plusieurs flux simultanés. La figure 2.12 montre sa notation.

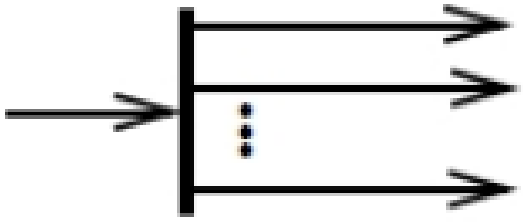


Figure 2.12: Notation de nœud de bifurcation [42]

- **Nœud d'union ou de jointure (join node) :** Un nœud d'union (nœud de jointure) est un nœud de contrôle qui synchronise des flots multiples. Il possède plusieurs arcs entrants et un seul arc sortant. Ce dernier ne peut être activé que lorsque tous les arcs entrants sont activés. L'exemple suivant illustre l'utilisation des nœuds de contrôle [15, 9].

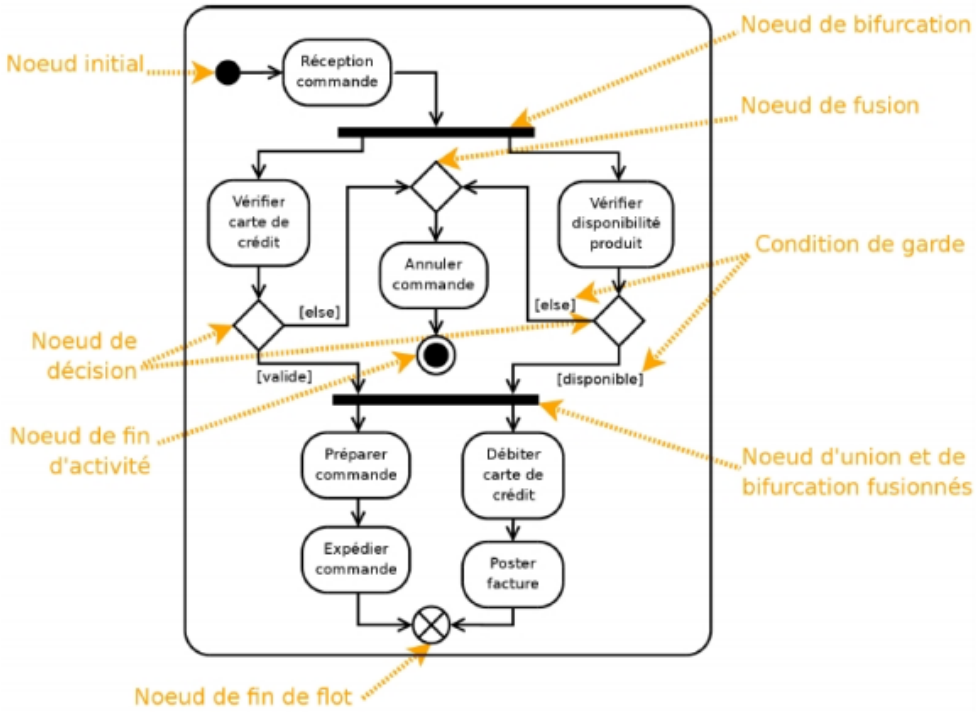


Figure 2.13: Exemple de diagramme d'activité illustrant l'utilisation de nœuds [41]

2.6.3.5.2 Nœud d'objet

Jusqu'ici, nous avons montré comment modéliser le comportement du flot de contrôle dans un diagramme d'activités. Or, les flots de données n'apparaissent pas et sont pourtant un élément essentiel des traitements (arguments des opérations, valeurs de retour...). un nœud d'objet permet de définir un flot d'objets (i.e. un flot de données) dans un diagramme d'activités. Ce nœud représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions [41].

Graphiquement, un tel nœud d'objet est représenté par un rectangle dans lequel est mentionné le type de l'objet. Des arcs viennent ensuite relier ce nœud d'objet à des activités sources et cibles. Le nom d'un état, ou d'une liste d'états, de l'objet peut être précisé entre crochets après ou sous le type de l'objet. Nous pouvons également préciser des contraintes entre accolades, soit à l'intérieur, soit en dessous du rectangle du nœud d'objet [43].

La figure ci-dessous représente graphiquement les nœuds d'activité.

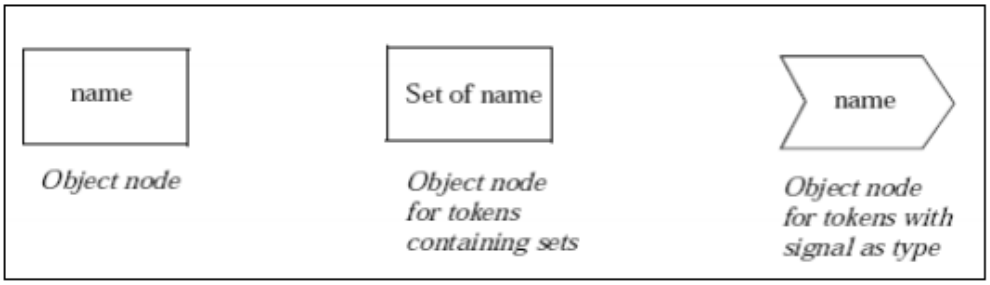


Figure 2.14: Notation nœud d'objet [15]

- **Pins d'entrée/sortie :**
  Pour spécifier les valeurs passées en argument à une activité et les valeurs de retour, nous utilisons des nœuds d'objets appelés pins (pin en anglais) d'entrée ou de sortie. L'activité ne peut débuter que si nous affectons une valeur à chacun de ses pins d'entrée. Quand l'activité se termine, une valeur doit être affectée à chacun de ses pins de sortie.
  Les valeurs sont passées par copie : une modification des valeurs d'entrée au cours du traitement de l'action n'est visible qu'à l'intérieur de l'activité.
  Graphiquement, un pin est représenté par un petit carré attaché à la bordure d'une activité (figure 2.15). Il est typé et éventuellement nommé. Il peut contenir

des flèches indiquant sa direction (entrée ou sortie) si l'activité ne permet pas de le déterminer de manière univoque [43].

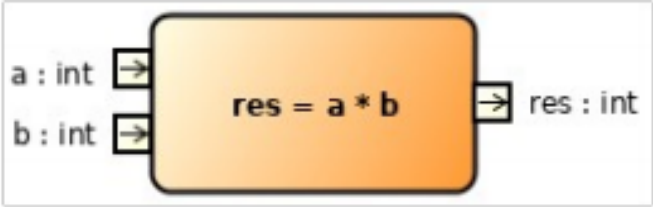


Figure 2.15: Représentation des pins d'entrée et de sortie sur une activité [43]

- **Pin de valeur :**
  Un pin valeur est un pin d'entrée qui fournit une valeur à une action sans que cette valeur ne provienne d'un arc de flot d'objets. Un pin valeur est toujours associé à une valeur spécifique. Graphiquement, un pin de valeur se représente comme un pin d'entrée avec la valeur associée écrite à proximité.
- **Flot d'objets :**
  Un flot d'objets permet de passer des données d'une activité à une autre. Un arc reliant un pin de sortie à un pin d'entrée est, par définition même des pins, un flot d'objets dans cette configuration, le type du pin récepteur doit être identique ou parent (au sens de la relation de généralisation) du type du pin émetteur.
  Il existe une autre représentation possible d'un flot d'objets, plus axée sur les données proprement dites, car elle fait intervenir un nœud d'objet détaché d'une activité particulière.
  Graphiquement, un tel nœud d'objet est représenté par un rectangle dans lequel est mentionné le type de l'objet (souligné). Des arcs viennent ensuite relier ce nœud d'objet à des activités sources et cibles. Le nom d'un état, ou d'une liste d'états, de l'objet peut être précisé entre crochets après ou sous le type de l'objet. On peut également préciser des contraintes entre accolades, soit à l'intérieur, soit en dessous du rectangle du nœud d'objet [41].
  Un flot d'objets peut porter une étiquette stéréotypée mentionnant deux comportements particuliers :
  - « transformation » indique une interprétation particulière de la donnée véhiculée par le flot ;
  - « selection » indique l'ordre dans lequel les objets sont choisis dans le nœud pour le quitter.

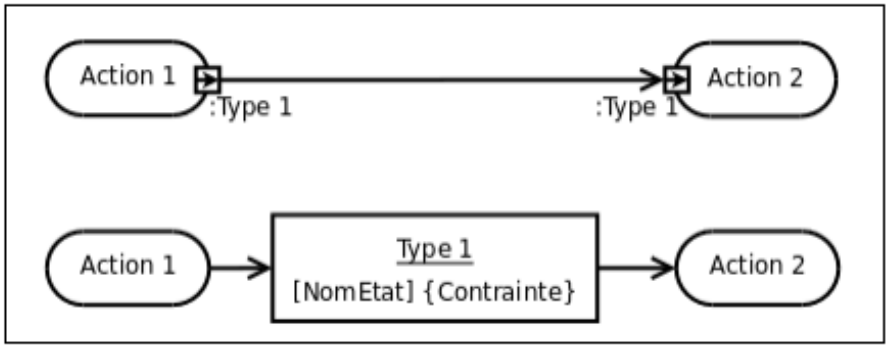


Figure 2.16: Deux notations possibles pour modéliser un flot de donné [41]

- **Nœuds de stockage de données (data store node) :**
  Un nœud de stockage des données est un nœud tampon central particulier qui assure la persistance des données. Lorsqu'une information est sélectionnée par un flux sortant, l'information est dupliquée et ne disparaît pas du nœud de stockage des données. Lorsqu'un flux entrant véhicule une donnée déjà stockée par le nœud de stockage des données, cette dernière est écrasée par la nouvelle [43].
  Ce nœud est représenté comme un nœud d'objet détaché avec le stéréotype « datastore ». Voici ci-dessous un exemple de nœud de stockage de donnée :

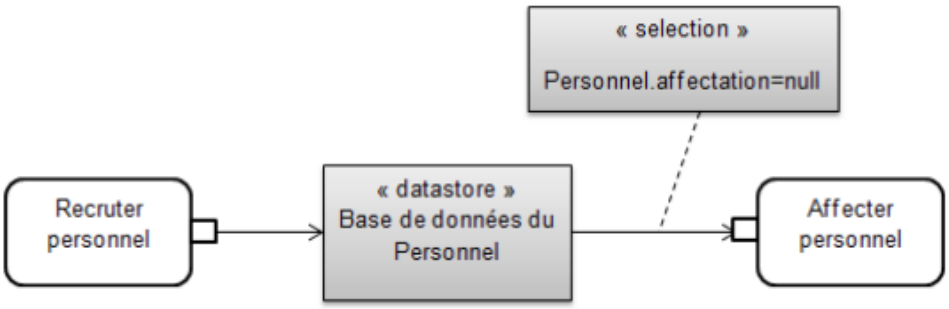


Figure 2.17: Un exemple de nœud de stockage de donnée [43]

Après avoir recruté le personnel, il est stocké dans le nœud de stockage des données de façon permanente, appelé dans ce cas "Base de données du Personnel". Ceux qui n'ont pas été affectés sont disponibles pour être affectés par l'activité "Affecter personnel". L'étiquette "selection : personnel.affectation=null" permet de sélectionner ceux qui n'ont pas été affectés [43].

- **Nœud central de mémoire tampon (central buffer node) :**
  Un nœud central de mémoire tampon est un nœud d'objet, destiné pour la gestion des flux provenant de multiples sources. Il peut avoir plusieurs arcs entrants et plusieurs arcs sortants.
  Graphiquement, on utilise le mot clé <centralBuffer> associé à la notation d'un nœud objet [15].

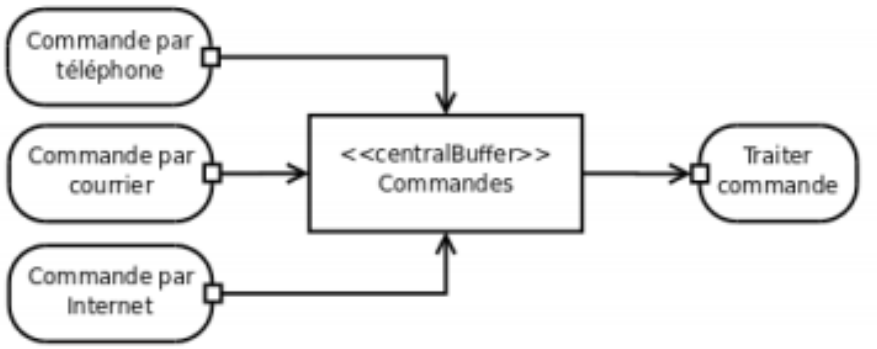


Figure 2.18: Exemple d'utilisation d'un nœud tampon centrale [41]

2.6.3.6 Les arcs (edges)

2.6.3.6.1 Arc d'activité (Activity Edge)

Contexte Un arc d'activité (activity edge en anglais) est une classe abstraite pour les connexions dirigées entre deux nœuds d'activités. Nous détaillons deux types particuliers d'arcs d'activités. [44]

Figure 2.19: Arc d'activité [15]

2.6.3.6.2 Flux de contrôle (Control Flow)

Contexte un arc de flot de contrôle (control flow en anglais) est un arc qui permet de d'écrire le séquencement de deux nœuds objets [9]. Ce type d'arc transmet uniquement

des jetons de contrôle (par opposition aux jetons de données). Les jetons offerts en entrée par le nœud source sont tous transmis au nœud de contrôle cible [44].

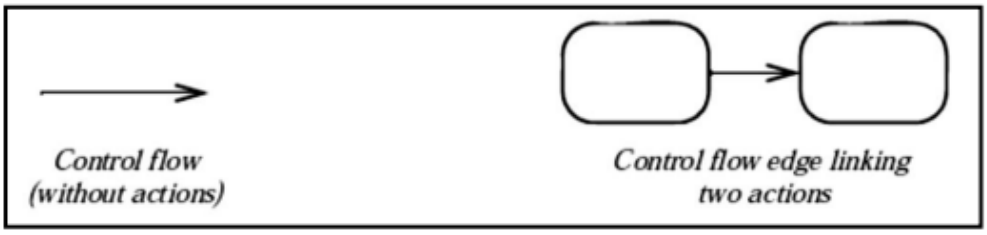


Figure 2.20: Notation flux de contrôle [9, 15]

2.6.3.6.3 Flux d'objet (Object Flow)

Contexte un arc de flot d'objets (Object flow en anglais) est un arc qui permet de décrire le séquencement de deux nœuds objets et transmet un jeton objet du nœud source au nœud cible. Ceci permet de transmettre des données d'une action à une autre [44].

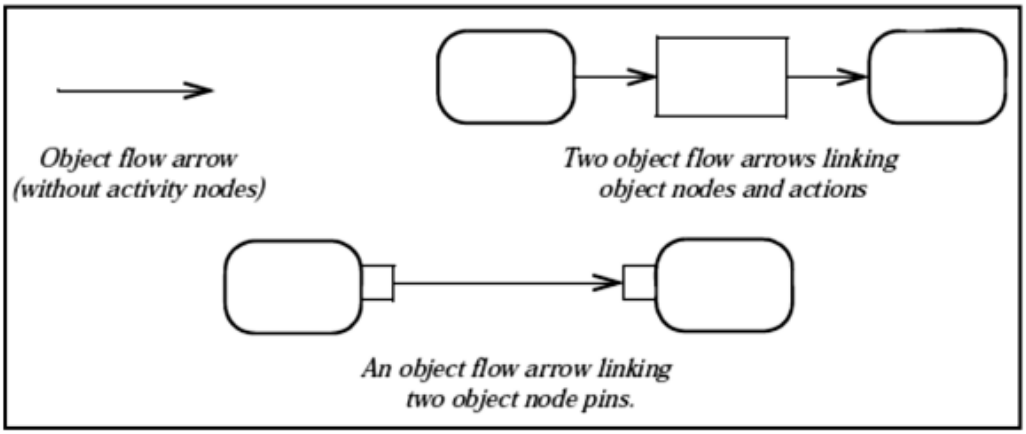


Figure 2.21: Notation flux d'objet [9]

2.6.3.6.4 Handler d'exception (Exception Handler)

Un handler d'exception (appelé aussi gestionnaire d'exception) est une activité qui spécifie un organisme à exécuter. Il possède un pin d'entrée du type de l'exception qu'il gère, et lié à l'activité protégée par un arc. La figure suivante montre les deux

représentations possibles du gestionnaire d'exception [15].

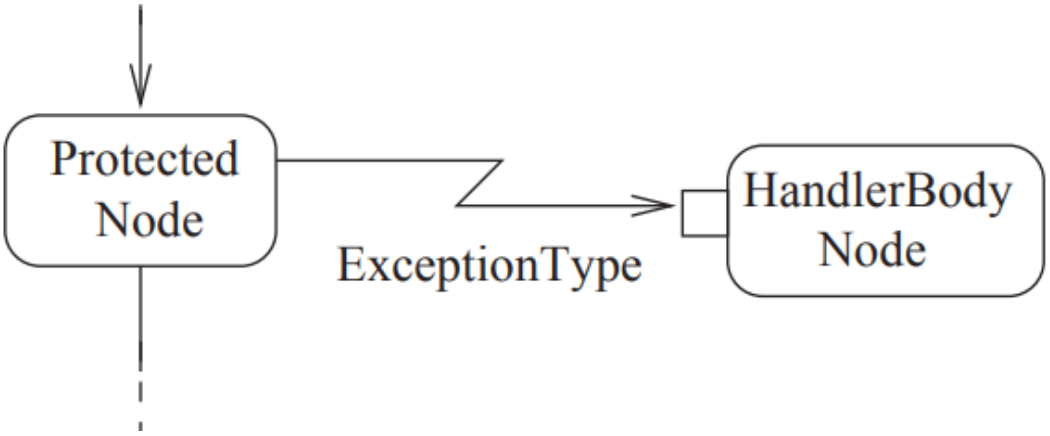


Figure 2.22: Notation d'un handlerd'exception [44]

2.7 Conclusion

Dans ce chapitre, nous avons parlé de la modélisation orientée objet comme une approche ayant fait ses preuves. Ensuite, nous avons donné une description détaillée des diagrammes d'activité d'UML 2.0. Nous avons insisté sur les nœuds d'activité et les arcs des diagrammes d'activité, qui seront modélisés dans le formalisme source de la transformation constituant l'objet de notre travail.Dans le chapitre qui va suivre nous présenterons un formalisme des réseaux de Pétri.

Chapitre 3

Méthodes formelles et modèles RPTT

3.1 Introduction

Dans l'informatique, spécifiquement l'ingénierie dirigée par les modèle, les méthodes formelles sont une sorte particulière de techniques mathématiques pour la spécification, le développement, la vérification et la validation de système logicielle.

Les méthodes formelles sont de plus en plus utilisées pour répondre aux exigences des systèmes critiques, en mettant l'accent sur une technique formelle couramment utilisée, réseaux de Pétri, Qui représentent un outil de modélisation universellement connu et reconnu pour les possibilités d'analyse, de validation et de vérification dont ils font preuve. L'exploitation de la théorie associée aux RdP permet, par la recherche des Places et Transitions invariants de répondre à de nombreux problèmes.

Les réseaux de Pétri représentent une technique formelle largement utilisée. Un intérêt particulier sera porté sur eux .Le réseau de Pétri est un outil graphique et mathématique pour modéliser et analyser les systèmes discrets, particulièrement les systèmes concurrents, parallèles, non-déterministes, etc. En étant rôle d'outil graphique, il nous aide à comprendre facilement le système modélisé, et plus il nous permet de simuler les activités dynamiques et concurrentes. Avec le rôle d'outil mathématique, il nous permet d'analyser le système modélisé grâce aux modèles de graphes, aux équations algébriques, etc..

Dans ce chapitre, nous présentons un panorama sur les méthodes formelles et leur classification ainsi que leur intérêts d'utilisation, ensuite, nous abordons l'utilisation et l'intégration de ces méthodes formelles à la méthodologie IDM, tout en présentant un rappel sur le Réseau de Pétri, ses concepts fondamentaux et ses propriétés.

3.2 Introduction aux méthodes formelles

Les méthodes formelles sont de plus en plus utilisées pour répondre aux exigences des systèmes critiques. Ces nouvelles méthodes sont apparues, les méthodes formelles. L'ajout de formalisme aux méthodes d'analyse et de conception à objets permet une meilleure structuration et une organisation des spécifications. D'un autre coté, les objets

seront plus surs et la conception plus rigoureuse..[45].

Elles sont utilisées dans le développement des logiciels les plus critiques pour donner une spécification du système que l'on souhaite développer [9].

3.2.1 Définition

En informatique, les méthodes formelles sont des techniques permettant de raisonner rigoureusement, à l'aide de logique mathématique, sur des programmes informatiques ou des matériels électroniques, afin de démontrer leur validité par rapport à une certaine spécification [9].

Ces méthodes permettent d'obtenir une très forte assurance de l'absence de bug dans les logiciels, c'est-à-dire d'acquérir des niveaux d'évaluation d'assurance élevés, elles sont basées sur les sémantiques des programmes, c'est-à-dire sur des descriptions mathématiques formelles du sens d'un programme donné par son code source (ou parfois, son code objet). Cependant, elles sont généralement coûteuses en ressources (humaines et matérielles) et actuellement réservées aux logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique [46].

3.2.2 Spécification formelle

Les méthodes de spécification formelles utilisent des techniques mathématiques pour décrire un problème. L'utilisation de notations formelles résoudre le problème de la variété d'interprétations par la rigueur du formalisme, l'abstraction, la syntaxe et la sémantique mathématiques bien définies. Elles engendrent des spécificationsprécises que l'on peut vérifier de manière systématique à l'aide d'outils. En identifiant tôt les problèmes du système, il est encore possible de les corriger avec un coût minimal. Ceci peut réduire le coût et la durée du développement. Ces améliorations pourraient aboutir à un processus prévisible qui produit un logiciel ayant un nombre réduit de fautes. Des études ont montré que l'addition de méthodes formelles au cycle de développement améliore le processus de développement et la qualité du logiciel, en imposant un contrôle dès les premières phases du développement, notamment la spécification.

Les méthodes formelles ne sont pas utiles si elles ne sont pas compréhensibles et faciles à utiliser. Différentes notations formelles ont différents niveaux de compréhension, mais elles sont relativement faciles à apprendre. Depuis qu'on a reconnu le potentiel des méthodes formelles pour produire de meilleures spécifications, plusieurs notations

appuyées par des outils ont été introduites. Les notations prennent différentes formes, comme tabulaire, graphique, mathématique, etc. Différents outils ont été développés, comme des éditeurs, des animateurs et des vérificateurs, pour manipuler ces notations [47].

3.2.3 Intérêts d'utilisation des méthodes formelles

L'avantage principal des méthodes formelles est l'utilisation de concepts de la logique et de la technique mathématique. Ces concepts fournissent des outils effectifs qui organisent les pensées des concepteurs et qui facilitent la communication entre toutes les personnes concernées par le développement. De plus, ils nous permettent de décrire de manière précise, non ambiguë, les demandes énoncées par l'utilisateur du système logiciel à réaliser. Les notions d'ensemble, de relation, de fonction et leurs différentes propriétés et opérations, avec les quantifications universelles et existentielles, nous permettent d'établir une spécification d'une manière simple et claire et de démontrer mathématiquement les propriétés de la spécification.

Les principaux avantages techniques d'une spécification formelle, par rapport à une spécification informelle, sont la précision et la clarté. Des imprécisions et des ambiguïtés peuvent facilement se glisser dans les spécifications informelles. Ceci peut ouvrir la voie à plusieurs interprétations. Par contre, les termes de spécifications formelles n'ont qu'une seule interprétation.

Un autre avantage des spécifications formelles est que les questions sont posées et répondues avec précision et d'une manière scientifique. De plus, les méthodes formelles fournissent des spécifications qui peuvent être rigoureusement vérifiées, analysées et testées dès les premières étapes du cycle de développement, ce qui n'est pas le cas dans les méthodes informelles. Cela signifie qu'il est possible de détecter et de corriger des fautes dès les premières étapes, ce qui réduit le coût et la durée du développement et améliore la qualité du logiciel [9].

Les méthodes formelles nous permettent de spécifier ce qui est nécessaire à un niveau d'abstraction particulier. Certains comportements et propriétés peuvent être volontairement exclus s'il est préférable que leurs élaborations soient remises aux prochaines phases du cycle de développement [47].

Selon **Hohan** et **Stopper** [38] les spécifications rigoureuses jouent un triple rôle dans le développement du logiciel. D'abord, les spécifications documentent avec précision les décisions de conception, indépendamment de l'implantation, et servent de

base pour la revue de cette conception. Durant l'implantation, les mêmes spécifications supportent le développement en parallèle. D'une part, elles renseignent les utilisateurs de ce qu'ils peuvent s'attendre et, d'autre part, elles renseignent les programmeurs de ce qu'ils doivent faire, et servent de base pour la phase de test. Finalement, durant la maintenance, ces mêmes spécifications supportent l'analyse de changements et aident à la formation de nouveaux personnels [47].

3.2.4 Classification des méthodes formelles

On distingue différentes classifications pour les méthodes formelles Selon J.M. Wing [39], peuvent être classées en trois approches figure 3.1.

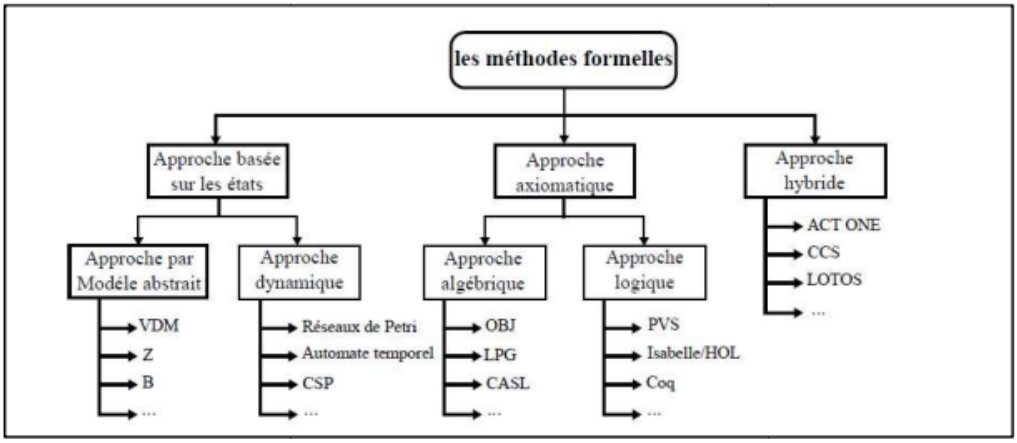


Figure 3.1: La classification des méthodes formelle [48]

3.2.4.1 Méthodes orientées propriétés ou axiomatiques

Cette approche traite de façon indirecte le comportement du système. Elle est basée sur la construction d'une axiomatisation qui permet d'obtenir les propriétés comportementales. Nous procédons à cette axiomatisation par approches algébriques ou logiques [9].

Le traitement axiomatique du logiciel pourrait bien être le type le plus ancien de méthodes formelles. Un exemple de ces méthodes la logique **Floyde-Hoare** [49] qui est généralement appelée la logique de Hoare .elle basée sur le traitementaxiomatique (utilisation de la sémantique axiomatique .Le but de cette logique est de formaliser la preuve de la correction des programmes.Elle utilise le triplet de Hoare qui se présent de la manière suivant Q PR.

Où P Est programme et le Q c'est la pré-condition et le R est la poste-condition.

Si Q est vrai au début de P, alors R sera vrai à la fin du programme P [8, 50].

L'approche logique exprime des systèmes transformationnels avec la logique temporelle. Elle exprime des propriétés dynamiques de sûreté et de vivacité des systèmes réactifs. Dans cette approche, nous nous intéressons à la démonstration de programmes en utilisant les théories de la démonstration automatique ou semi-automatique de théorèmes. Parmi les langages de spécification logiques nous citons : PVS Isabelle/HOL et Coq [35].

Méthodes Algébriques de type abstrait de donnée, ou type abstrait algébrique, est la donnée d'une signature et d'un ensemble d'axiomes. La signature comprend des noms de types (les sortes) et des opérations définies par leur profil. La signature permet de construire toutes les valeurs des types de données par application des opérations. Les axiomes sont des formules logiques. Une spécification algébrique est un système formel dont le langage est donné par la signature et le système d'inférence est basé sur les axiomes et la déduction équationnelle et dont l'interprétation est donnée en termes d'algèbres [45].

3.2.4.2 L'approche basée sur les états

Cette approche s'intéresse aux données du système. Elle construit un modèle en termes de structures mathématiques tout en gardant les propriétés du système. Le modèle obtenu servira à l'étude du fonctionnement du système, en lui appliquant des approches dynamiques et des approches par modèle abstrait [9].

Les méthodes dynamique sont utiliser dans le domaine des protocoles qui se base sir l'interaction antre les processus .ces méthodes peuvent être utiles pour le traitement des systèmes de transition comme les automates de pétri, les algèbres de processus (CSP) et la logique temporelle [51].

3.2.4.3 L'approche hybride

Les approches hybrides modélisent les systèmes par des processus ou des systèmes de transitions. Les données manipulées sont définies par des spécifications algébriques. Les aspects fonctionnels sont soit explicites dans les processus séquentiels soit implicites par transformation des données dans les systèmes de transition. LOTOS est un langage basé sur l'algèbre de processus CCS enrichie par certains mécanismes de CSP. Les données sont décrites par le langage de spécification algébrique ACT-ONE. LOTOS présente

l'intérêt d'être une norme ISO et d'être outille (simulation, exécution, vérification). Les réseaux algébriques hiérarchiques définissent des réseaux de Pétri dont les jetons sont des valeurs de types de données. Les places sont des stocks de valeurs et les transitions des transformations de données [45].

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 Pétri 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 Pétri

3.2.5 Combinaison d'IDM avec les Méthodes Formelles

Aujourd'hui l'utilisation de méthodes formelles (MFs) est essentielle pour le développement de systèmes complexes, en particulier pour les systèmes critiques où les questions liées à la sûreté/fiabilité sont fondamentales. D'autre part, l'IDM a atteint un bon niveau de maturité et est devenue une nouvelle démarche en génie logiciel qui conçoit l'intégralité du cycle de développement en se basant sur la méta-modélisation et transformation de modèles.

Bien sûr, chacune des deux approches a des points forts et des points faibles. Dans ce qui suit, nous examinons comment ces deux approches peuvent être combinées en montrant comment les inconvénients des méthodes formelles peuvent être surmontés grâce aux apports de l'IDM et réciproquement. La Figure 3.2 représente brièvement les avantages et les inconvénients de l'IDM et des MFs [52].

	Avantages	*Inconvénients*
IDM	√ Notations conviviales et souvent graphiques √ Génération automatique d'outils de développement √ Transformations automatiques de modèles	✗ Manque de sémantiques formelles ✗ Analyse de modèles impossible
MFs	√ Fondements mathématiques rigoureux √ Analyse formelle de modèles	✗ Notations complexes ✗ Absence d'outils de développement ✗ Manque d'intégration

Figure 3.2: Avantages et les inconvénients de l'IDM et des MFs [52]

- ***Avantages des MFs.*** L'utilisation des méthodes formelles dans l'ingénierie des systèmes devient indispensable, surtout dans les phases amont du développement. Un modèle abstrait du système peut servir de support pour s'assurer que le système en cours de développement répond aux exigences des clients par simulation ou tests, et de garantir certaines propriétés liées à son comportement par l'analyse formelle (validation et vérification).
- ***Inconvénients des MFs.*** Bien qu'il existe plusieurs applications des méthodes formelles dans le monde industriel et avec de bons résultats, les développeurs hésitent encore à adopter les MFs. En plus de leur difficulté d'utilisation et d'apprentissage, ce scepticisme est principalement dû :

  1. Aux notations complexes que les techniques formelles utilisent par rapport à d'autres notations intuitives et graphiques utilisées par les langages semi-formels comme le langage UML.
  2. A l'absence de support outillé qui permet d'aider et d'assister les développeurs dans leurs démarches de développement de manière transparente.
  3. Au manque d'intégration entre les différentes méthodes formelles et leurs outils d'analyse adjacents.

- ***Avantages de l'IDM.*** L'approche IDM met davantage l'accent sur l'automatisation du processus de développement des systèmes. Cette approche est basée principalement sur des représentations du système à un haut niveau d'abstraction qui sont les modèles. Le processus de développement, dans cette approche, revient à raffiner, maintenir, et éventuellement de transformer en d'autres modèles ou de générer le code exécutable. Il est important à noter que ces différentes activités se font d'une manière automatique. En plus, la méta-modélisation, qui est aussi un concept clé de l'approche IDM, permet de donner à un langage de modélisation une notation abstraite, ce qui permet de générer automatiquement son éditeur. La métamodélisation de langages est de plus en plus adoptée pour des domaines spécifiques afin de réduire la complexité et d'exprimer efficacement les concepts du domaine [34].
- ***Inconvénients de l'IDM.*** Malgré que la définition d'une syntaxe abstraite d'un langage par un méta-modèle est bien maîtrisée et supportée par de nombreux environnements de métamodélisation, la définition de la sémantique de ces langages reste une question ouverte et cruciale. Actuellement, les environnements

de méta-modélisation sont en mesure de faire face à la plupart des problèmes de définition syntaxique, mais ils manquent de tout support rigoureux permettant de fournir la sémantique des méta-modèles. La sémantique est généralement donnée en langage naturel, cela implique que les langages définis par métamodélisation ne sont pas encore aptes à l'analyse formelle de leurs modèles [34].
L'absence de notations conviviale, d'intégration des déférentes techniques formelles et d'interopérables de leurs outils sont des défis importants pour les MFs. L'IDM permet, par le biais des notions de méta-modèle et de transformation de modèles, de concevoir des supports outillés pour les méthodes formelles tout en assurant leur interopérabilité. En se sens, l'IDM n'apporte rien de nouveau sur le plan théorique aux concepts d'analyse formelle, mais elle peut leur permettre une meilleure intégration afin de profiter pleinement de leurs avantages [34].

3.3 Langages formels

Pour définir une spécification d'un système, une méthode formelle utilise un langage formel doté d'une sémantique mathématique rigoureuse, basée sur des règles d'interprétation et des règles de déduction.

Les règles d'interprétation garantissent l'absence d'ambigüité d'interprétation (dans un langage informel ou semi-formel, une description peut avoir plusieurs interprétations). Les règles de déduction raisonnent sur des spécifications afin de détecter les manques et les inconsistances, et aussi pour vérifier des propriétés attendues [35].

3.4 Techniques d'analyse

Comme les systèmes deviennent de plus en plus complexes, il est crucial non seulement d'assurer certaines performances, mais également de garantir l'absence de risques de dysfonctionnement ou au moins limiter leur impact. Pour cela, de nombreuses méthodes et techniques ont été développées durant les dernières décennies pour analyser ce type de systèmes [34]. Parmi ces méthodes nous citons :

3.4.1 La vérification

La vérification répond à la question "Construisons-nous correctement le modèle ?" (*"Is the system being built right ?"*). La vérification est l'ensemble des actions de revue, inspection, test, simulation, preuve automatique, ou autres techniques appropriées

permettant d'établir et de documenter la conformité des artéfacts du développement vis-à-vis des critères préétablis [34]. La vérification est définie comme étant *"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"* [8].

3.4.2 Validation

La validation cherche à répondre à la question "Construisons nous le bon modèle ?" (*"Is the right system being built ?"*). La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs. Par définition la validation est 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"* [34].

3.4.3 Qualification

La qualification consiste à s'assurer que le modèle peut servir à la communication sans ambiguïtés ni interprétation parasite possible entre les activités du projet et/ou entre les acteurs d'un groupe de travail [34].

3.4.4 Certification

Elle consiste à s'assurer que le modèle respecte une norme et peut servir de base à l'établissement d'un référentiel réutilisable et générique à un domaine. Cela sous entend la nécessaire implication et la responsabilité d'un organisme tiers qui reconnaît la pertinence, la rigueur, l'intérêt du modèle et garantit ces qualités lors de sa diffusion [34].

3.5 Les techniques de vérification formelle

Désignent un ensemble de méthodes basées sur les mathématiques et la logique pour assurer qu'un produit final est conforme à ses spécifications. Ces dernières doivent être implémentées dans un langage défini mathématiquement (syntaxe et sémantique), et que nous présentons quelques-unes dans ce qui suit [50].

3.5.1 Vérification formelle par "Model checking"

Le modèle-checking consiste à vérifier si un modèle d'un système ou une abstraction, satisfait une propriété à l'aide d'un traitement de toutes les exécutions possibles en parcourant d'une manière exhaustive le graphe d'état correspondant au système [1].

Nous montrerons que la vérification des modèles est basée sur des paradigmes bien connus de la théorie des automates, des algorithmes de graphes, de la logique et des structures de données. Sa complexité est analysée à l'aide de techniques standard de la théorie de la complexité [53].

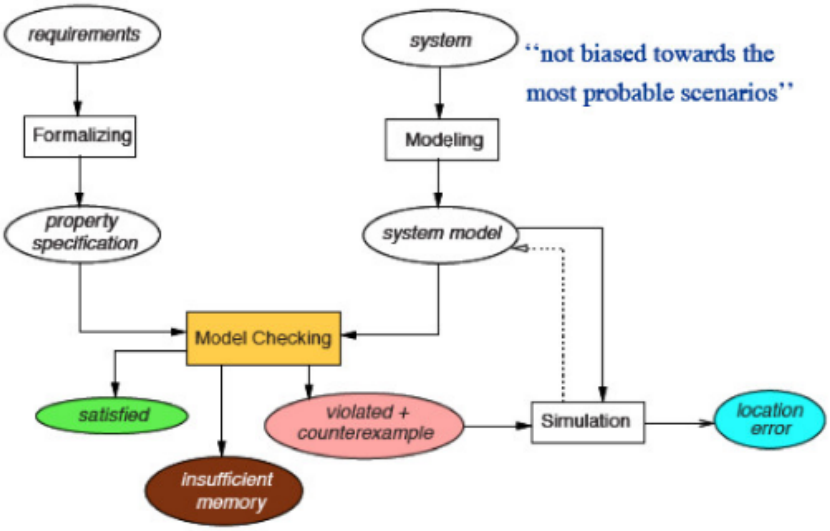


Figure 3.3: Principe du model checking [53]

3.5.2 Vérification formelle par Théorème de preuve

Théorème de preuve nécessite que le système soit spécifié sous forme d'une théorie mathématique, ou devrait être transformé en une telle forme. En utilisant un ensemble d'axiomes (le théorème de base), un démonstrateur (le logiciel) tente de construire.

Le principal avantage de la Théorème de preuve est qu'il peut agir avec des espaces d'états infinis et peut vérifier la validité des propriétés pour des valeurs de paramètres arbitraires [9].

3.5.3 Vérification formelle par la simulation

En pratique, l'une des techniques de vérification la plus connue et utilisée est la simulation. Le simulateur permet à l'utilisateur d'étudier le comportement du système. Ceci passe par la détermination, sur la base du modèle du système, des réactions que doit avoir le système vis-à-vis de certains scénarios spécifiques. Elle est cependant moins adaptée à détecter les erreurs subtiles, car pour le simulateur, il est impossible de générer tous les scénarios possibles du système [9, 54].

3.5.4 Vérification formelle par le test

La technique de vérification basée sur le test est applicable dans le cas où il est difficile, voire impossible, d'obtenir un modèle du système à vérifier. Avec le test, des séquences d'actions. Les tests sont plutôt improvisés et pas très systématiques. Comme résultat, le test est une activité très élaborée, prédisposant aux erreurs, et difficilement gérable. Similaire au model checking [55, 9].

3.6 Les Réseaux de Pétri

Il est clair que les systèmes dynamiques ne peuvent pas être décrits en référant seulement à leurs états initiaux et finaux. Une description adéquate doit tenir compte de leur comportement permanent qui est une séquence (peut-être infinie) d'états. Plusieurs techniques formelles ont déjà été proposées pour spécifier, analyser et vérifier ce genre de systèmes. Les réseaux de Pétri représentent une technique formelle largement utilisés [34].

Le formalisme des réseaux de Pétri (RdP) a été introduit par Carl Adam Pétri, en 1962 à l'université Darmstadt dans sa thèse intitulée : " Communication avec des automates" comme un outil de modélisation graphique et mathématique permettant la modélisation et l'analyse des systèmes dynamiques à événements discrets. En tant qu'outil graphique, les réseaux de Pétri permettent la visualisation du comportement dynamique et les activités concurrentes du systèmeEn tant qu'outil mathématique, ils permettent l'analyse des propriétés du système concernant sa structure et son comportement. Ce formalisme bénéficie d'une riche panoplie de techniques d'analyse et d'outils. Les résultats de cette analyse sont utilisés pour évaluer le système et en permettre la modification ou l'amélioration. La Figure 3.4 montre la méthode générale basée sur le formalisme des réseaux de Pétri pour la modélisation et l'analyse des systèmes [34].

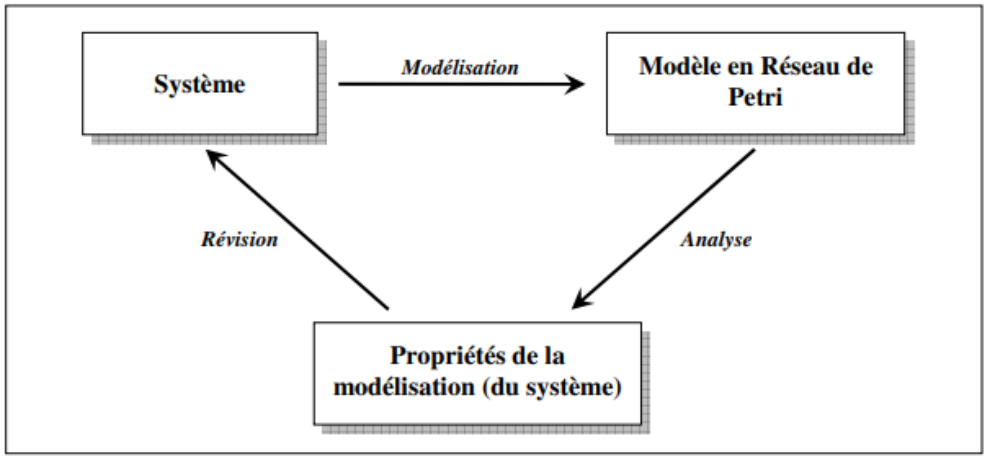


Figure 3.4: Méthodes générale de modélisation et d'analyse basée sur les réseaux de Pétri [34]

Grâce à leur généralité et à leur souplesse, les réseaux de Pétri sont utilisés dans une large variété de domaines tels que les protocoles de communication, les systèmes distribués, l'architecture des ordinateurs, etc. [34]

3.6.1 Concepts de base & définition

3.6.1.1 Définition Graphique

Un réseau de Pétri (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 [6].
- Chaque arc est étiqueté par une valeur (ou un poids), qui est un nombre entierpositif. L'arc ayant k poids peut être interprété comme un ensemble de k arcsparallèles. Un arc qui n'a pas d'étiquette est un arc dont le poids est égal à 1.

La figure 3.5 illustre la notation graphique d'un Réseau de Pétri [42].

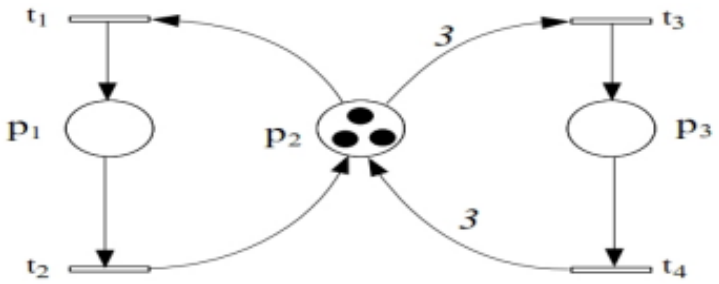


Figure 3.5: Exemple d'un Réseau de Pétri [42]

3.6.1.2 Définition informelle

Comme l'illustre le RdP de la Figure 3.6, un réseau de Pétri est un graphe biparti orienté (ayant deux types de nœuds) : des places représentées par des cercles et des transitions représentées par des rectangles. Les arcs du graphe ne peuvent relier que des places vers des transitions, ou des transitions vers des places. Un réseau de Pétri décrit un système dynamique à événements discrets. Les places permettent la description des états possibles du système et les transitions permettent la description des événements ou les actions qui causent le changement de l'état. Un réseau de Pétri est un graphe muni d'une sémantique opérationnelle, c'est-à-dire qu'un comportement est associé au graphe, ce qui permet de décrire la dynamique du système représenté. Pour cela un troisième élément est ajouté aux places et aux transitions : les jetons [34].

Une répartition des jetons dans les places à un instant donné est appelée marquage du réseau de Pétri. Un marquage donne l'état du système. Le nombre de jetons contenus dans une place est un entier positif ou nul. Pour un marquage donné, une transition est

franchissable si chacune de ses places d'entrée contient au moins un jeton. L'ensemble des transitions franchissables pour un marquage donné défini l'ensemble des changements d'états possibles du système depuis l'état correspondant à ce marquage. C'est un moyen de définir l'ensemble des événements auxquels ce système est réceptif dans cet état [34].

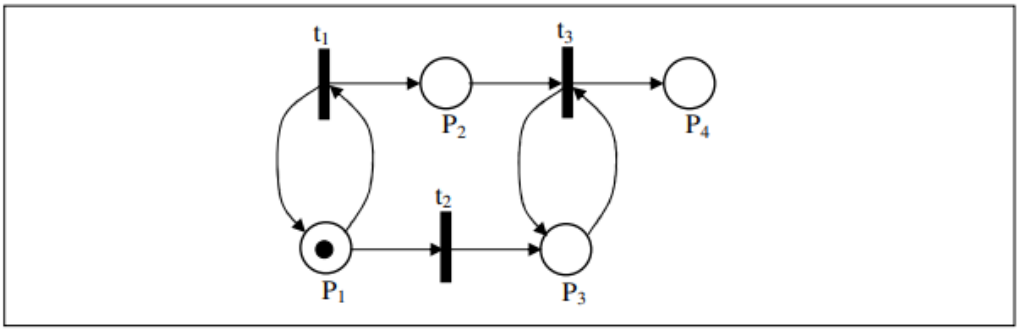


Figure 3.6: Exemple de réseau de Pétri marqué [34]

3.6.1.3 Définition formelle

De manière plus formelle, nous dirons qu'un réseau de Pétri est un quadruplet $R = (P, T, I, O)$ où :

$P = P_1, P_2, ..., P_n$ est un ensemble fini et non vide de places,

$T = T_1, T_2, ..., T_m$ est un ensemble fini et non vide de transitions,

$I : P \times T \rightarrow N$ est l'application d'incidence avant, où N est l'ensemble des entiers non négatifs,

$O : P \times T \rightarrow N$ est l'application d'incidence arrière.

Notons que $P \cap T =$ *et* $P \cup T \neq \emptyset$, c'est à dire que les ensembles P et T sont disjoints.

Le marquage du réseau est donné par une application Mar définie par :

$Mar : P \rightarrow N$ tel que $Mar(P_i) = mp_i$ qui indique le nombre de jetons dans la place P_i.

Le marquage à un instant t donnée d'un réseau de Pétri est donc un vecteur dimension n donné par : $m(t) = [mP_1(t), mp_2(t), ..., mp_n(t)]^T$.

Un réseau de Pétri marqué est le couple $N = (R, m_0)$ formé d'un réseau de Petri R et m_0 [56].

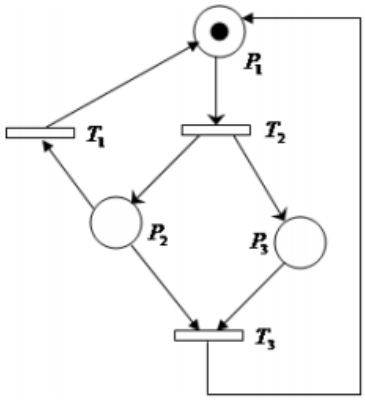


Figure 3.7: Exemple de réseau de Pétri ordinaire [57]

Le marquage initial du réseau i.e. à $t = 0$. La Figure 3.7 montre un exemple de réseau de Pétri. A chaque arc est associé un poids entier positif (défini par les applications I et O). Lorsque ce poids n'est pas porté sur l'arc, par convention il est égal à 1. Lorsque tous les poids des arcs sont égaux à 1, le réseau de Pétri est dit ordinaire (c'est le cas du réseau de Pétri de la Figure 3.7). Dans ce qui va suivre, les notations suivantes sont adoptées [56],

${}^{o}T_j$: l'ensemble des places d'entrée de la transition T_j,

$T {}^{o}_{j}$: l'ensemble des places de sortie de la transition T_j,

$P {}^{o}_{i}$: l'ensemble des transitions de sortie de la place P_i,

${}^{o}P_i$: l'ensemble des transitions d'entrée de la place P_i.

Le marquage d'un réseau de Pétri permet de définir l'état d'un système modélisé par ce réseau. Le marquage consiste à placer initialement un nombre mi entier (positif ou nul) de jetons dans chaque place P_i du réseau. Le marquage du réseau sera défini par un vecteur $M = \{mi\}$ (cf. la figure 3.8).

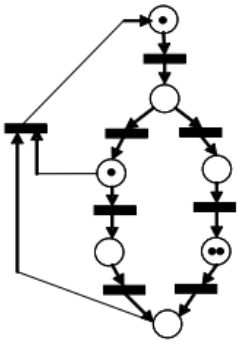


Figure 3.8: Un réseau de Pétri marqué avec un vecteur de marquage M : M = (1,0,1,0,0,2,0) [58]

3.6.1.4 Représentation matricielle

Il est possible de représenter les réseaux de Pétri par des matrices. - La matrice $Pré(pi, tj)$: c'est le poids k de l'arc reliant une place à une transition [57]

$$Pré(P_i, T_j) = \begin{cases} k, & \text{si l'arc } W(pi, tj) \text{ existe} \\ 0, & \text{sinon} \end{cases}$$

La matrice $Post(tj, pi)$: c'est le poids k de l'arc reliant une transition à une place.

$$Post(T_j, P_i) = \begin{cases} k, & \text{si l'arc } W(tj, pi) \text{ existe} \\ 0, & \text{sinon} \end{cases}$$

La matrice $C = Post - Pré$ est appelée matrice d'incidence du réseau

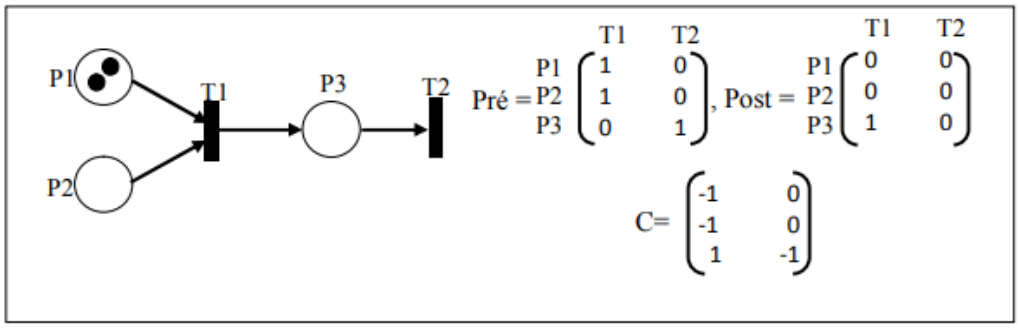


Figure 3.9: Représentation matricielle d'un Rdp [57]

3.6.1.5 Évolution d'un Réseau de Pétri

L'évolution d'un Réseau de Pétri 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 [42].

3.6.1.5.1 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.

3.6.1.5.2 Règle de Franchissement

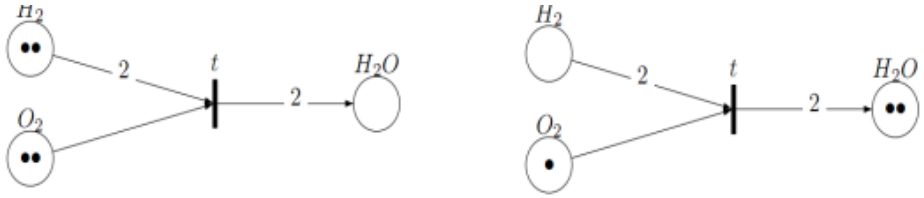
Si la transition est validée, on peut effectuer le franchissement de cette transition : on dit alors que la transition est franchissable [9, 42].

Le franchissement consiste à :

- retirer W(p, t) jetons dans chacune des places en entrée p de la transition t.
- ajouter W(t, p) jetons à chacune des places en sortie p de la transition t. La règle de franchissement est illustrée par la figure 3.10 en utilisant la réaction chimique connue

$$2H_2 + O_2 \rightarrow 2H_2O$$

La présence des deux jetons dans chaque place d'entrée (figure 3.10 a), indique que 2 unités de H_2 et 2 unités de O_2 sont disponibles, donc la transition t est franchissable. Après avoir franchi t, le marquage va changer et on obtient le réseau ayant le marquage comme celui de la (figure 3.10 b), maintenant t n'est plus franchissable [9, 42].



(a) Marquage avant de franchir la transition t.

(b) Marquage après avoir franchi t

Figure 3.10: Exemple de règle de franchissement de transition [9]

3.6.2 Modélisation Avec les Réseaux de Pétri

Les réseaux de Pétri ont été conçus et utilisés principalement pour la modélisation. Plusieurs systèmes peuvent être modélisés par les réseaux de Pétri, ces derniers peuvent être de natures très diverses : matériel informatique, logiciels informatiques, les systèmes physiques, les systèmes sociaux, etc. [42].

Dans les sections suivantes nous donnerons quelques exemples de modélisation des problèmes informatiques et mathématiques avec les réseaux de Pétri.

3.6.2.1 Parallélisme

Dans le réseau de Pétri représenté par la figure **??** 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)

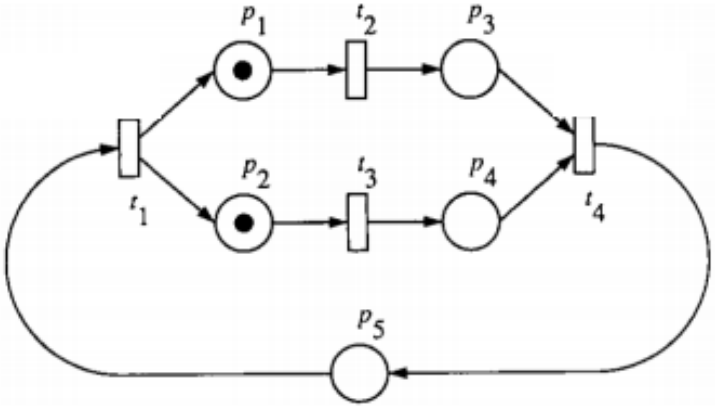


Figure 3.11: Parallélisme dans les Réseaux de Pétri [59]

3.6.2.2 Synchronisation

Les réseaux de pétri ont été utilisés pour modéliser une variété de mécanismes de synchronisation, y compris les problèmes de l'exclusion Mutuelle, producteur/consommateur, lecteurs/écrivains... etc. [59].

Exemple 1 : Problème du producteur/consommateurs Considérons le problème du producteur-consommateur où le producteur ne reprend que si le Buffer est vide. La transition produire ne pourra être tirée que si la place buffer ne contient pas de jetons. [59]

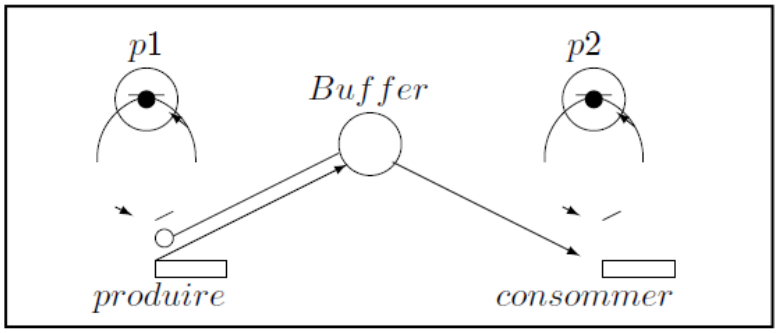


Figure 3.12: Problème du producteur et consommateurs [9]

Exemple 2 : Exclusion mutuelle

Ce problème peut être résolu par un réseau de Pétri comme celui de la figure 3.13 La place m représente la permission d'entrer dans la section critique. Pour qu'un processus entre dans la section critique, il doit avoir un jeton dans p_1 ou p_2 pour signaler qu'il souhaite entrer dans la section critique et il doit y avoir un jeton dans la place m pour avoir la permission d'entrer dans la section critique. Si les deux processus souhaitent entrer simultanément, les transitions *t1* et *t2* seront en conflit. Seulement l'une d'eux peut être franchie. Le franchissement de *t1* désactive *t2*, ce qui oblige le processus *2* à attendre la sortie de processus *1* de sa section critique et à mettre un jeton à la place m. [42]

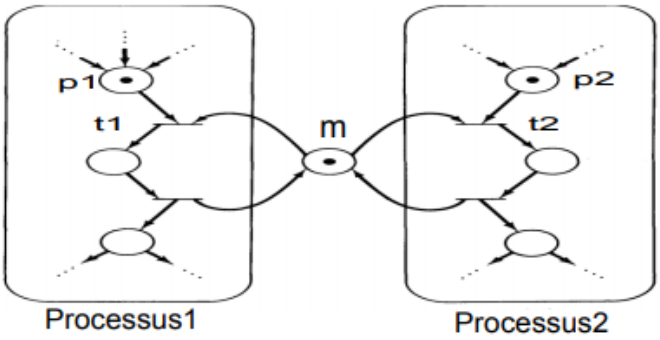


Figure 3.13: Exclusion mutuelle [42]

3.6.2.3 Calcul De Flux De Données

Un calcul de flux de données est celui dans lequel les instructions sont activées pour l'exécution par l'arrivée de leurs opérandes [59]. Ils peuvent être exécutées simultanément. Les réseaux de Pétri peuvent être utilisés pour représenter non seulement le flux de contrôle, mais également le flux de données. Les jetons dénotent les valeurs des données en cours ainsi que la disponibilité des données [9, 42].

Le réseau de Pétri de la figure 3.14 représente un calcul de flux de données de la formule suivante :

$$x = \frac{a+b}{a-b}$$

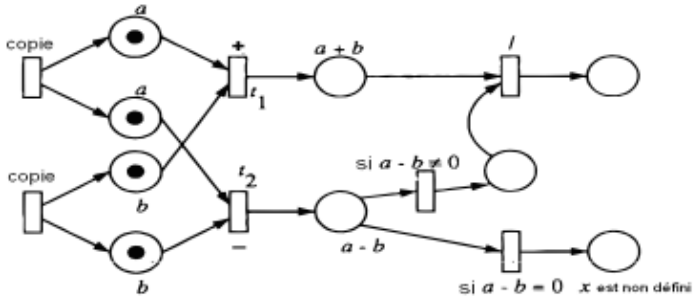


Figure 3.14: Exemple d'un calcul de flux de données par un Rdp [9, 42]

3.6.3 Propriétés des réseaux de Pétri

En tant qu'outil mathématique, les réseaux de Pétri possèdent un certain nombre de propriétés. Quand ces propriétés sont interprétées dans le contexte du système modé-

lisé, elles permettent d'identifier la présence ou l'absence des propriétés fonctionnelles spécifiques au domaine d'application du système. Les propriétés définies dans cette sectionsont l'accessibilité, l'état d'accueil, la bornitude et la vivacitéect. Ces propriétés dépendent du marquage initial du réseau de Pétri. [56]

1. **Accessibilité :** Un marquage m_k est dit accessible à partir du marquage m_0, s'il existe une séquence de franchissement $\sigma = T_1T_2...T_k$ qui transforme m_0 en m_k. Dans ce cas, on écrit m_0 $[\sigma > m_k]$ ce qui signifie que m_k est accessible à partir de m_0 par σ. L'ensemble de tous les marquages accessibles à partir de m_0 dans un réseau de Pétri N est noté R(m_0). L'ensemble de toutes les séquences de franchissement possibles à partir de m_0 dans un réseau de Petri N est noté $L(m_0)$.[56]
2. **Caractère Borné :** Une place « p » du réseau marqué (R, M_0) est k-bornée ($k \in IN$, avec $k > 0$) si pour tout marquage M accessible depuis M_0, $M(p) \leq k$ (le nombre de marques dans Pi est fini). Dans le cas contraire la place « p » est dite non-bornée. Un RdP est borné pour un marquage initial M_0 si toutes ses places sont bornées [57].

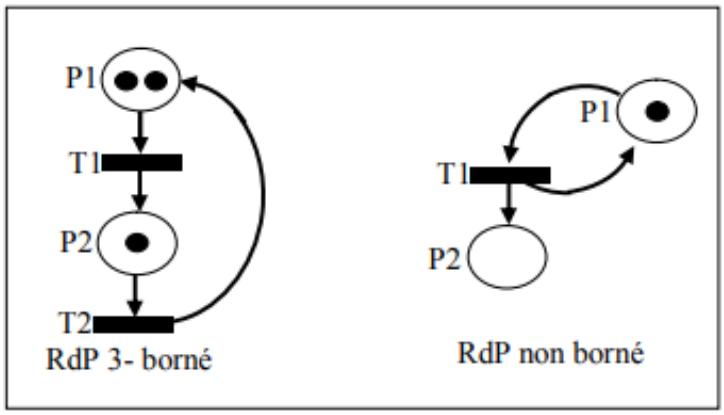


Figure 3.15: RdP 3-borné / non borné [57]

3. **Absence de blocage :** Cette propriété implique seulement que le réseau a toujours la possibilité d'évoluer et de progresser [60].

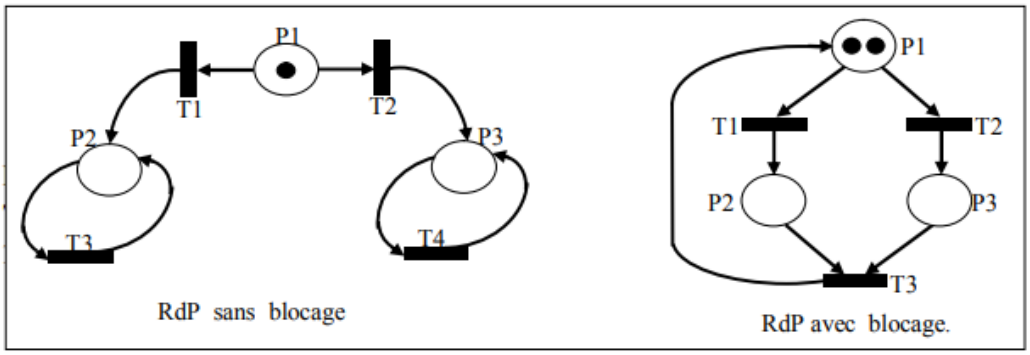


Figure 3.16: RdP sans / avec blocage [60]

4. **Conflits :** Un conflit structurel a précédemment été défini comme l'existence d'une place Pi qui a au moins deux transitions de sortie T_j ,T_k, etc.. Notation : $< P_i, T_j, T_k, ... >$.

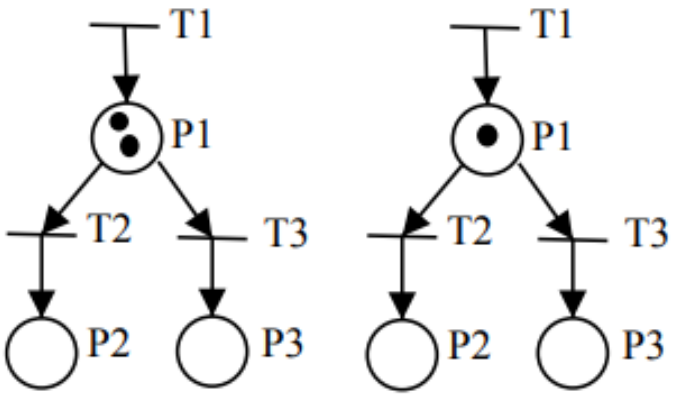


Figure 3.17: Conflit effectif ou pas [60]

Un conflit effectif est l'existence d'un conflit structurel k, et d'un marquage M, tel que le nombre de marques dans p est inférieur au nombre de transitions de sortie de p qui sont validées par M [35].

3.6.4 Extensions des Réseaux de Pétri

Malgré ces qualités, Les RdPs classiques d'Adam Pétri souffrent de certains reproches en termes d'expressivité tels que la distinction entre les jetons, l'expression de dimension temporelle ou le manque de structuration. Par conséquent, l'utilisation des RdPs

produirait des modèles dont la taille croît rapidement avec la complexité du système, et qui deviennent rapidement difficiles à manipuler et/ou à analyser. Pour augmenter le niveau d'expressivité, les RdPs ont connu plusieurs extensions au fils des années. Les extensions les plus connues sont les réseaux de Pétri colorés, Les ECATNets, les réseaux de Pétri algébriques, les réseaux de PetriPredicate/Transition, Les réseaux de PetriNested Net, les G-Nets, etc. [34].

Dans la section suivante, nous présentons un aperçu de quelques extensions des Rdps.

3.6.4.1 Les réseaux de Pétri colorés

Afin d'augmenter l'expressivité d'un RdP, les jetons placés dans les états du réseau seront colorés, ce qui permettra de les distinguer. Ce procédé de marquage permet de distinguer les jetons d'un même état. L'exemple du producteur consommateur sera agréablement modélisé par un réseau de Pétri coloré. Chaque processus sera représenté par un jeton de couleur différente dans un même état du réseau, ce qui facilitera le tir de la transition [58].

3.6.4.2 Les réseaux de Pétri temporisés

Dans ce modèle de réseau de Pétri, la durée d'une activité est explicitement intégrée. La temporisation peut concerner les places (réseaux de Pétri P-temporisés) ou bien les transitions (réseaux de Pétri T-temporisés) selon les événements modélisés. Beaucoup d'autres extensions des réseaux de Pétri sont développées. Dans cette thèse, nous nous intéressons aux systèmes à base d'agents mobiles. Nous focaliserons dans les sections suivantes sur les réseaux de Pétri étendus pour modéliser la mobilité [58].

3.6.4.3 Réseaux de Pétri à arcs étiquetés ou généralisé

Un exemple de RdP généralisé et de RdP ordinaire équivalent est présenté Figure 3.18. L'intérêt de l'utilisation de RdP généralisés est évident : il permet d'obtenir des modèles RdP plus concis. Les RdPs généralisés font partie des classes de modèles RdPs appelées abréviations. Ces classes de modèles RdPs permettent de représenter les mêmes systèmes que les RdPs mais avec une représentation graphique beaucoup plus concise. Une seconde classe d'abréviations est les RdPs colorés [51].

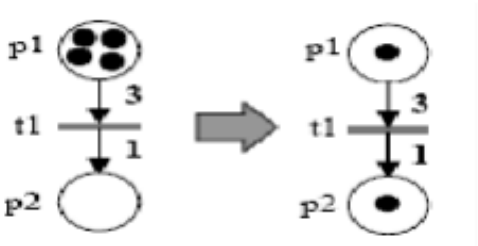


Figure 3.18: RdP à arc étiquetés [51]

3.6.4.4 Réseaux de Pétri autonomes

Un RdP autonome décrit le fonctionnement d'un système de façon autonome, c'est-à-dire dont l'évolution est régi par ses propres lois. La succession des quartes saisons peut être ainsi modélisée par un RdP autonome [51].

3.6.4.5 Réseaux de Pétri ordinaire

L'arc de poids $n = 1$ est un arc ordinaire. Donc si tous les arcs d'un réseau sont ordinaires le réseau sera dit ORDINAIRE.

La description formelle de ce modèle est définie par un 6-uplet :

$PN =< P, T, M_0, A, Pre, Post >$ avec :

- P, un ensemble fini de places,
- T, un ensemble fini de transitions,
- M_0, le marquage initial du réseau,
- A, un ensemble fini d'arcs tel que $P \setminus T = P \setminus A = T \setminus A = 0$
- Pre, indique combien de jetons sont consommés depuis une place vers une transition,
- $Post$, indique combien de jetons sont produits par une transition dans la place avale [8].

3.6.5 Réseaux de Pétri temporels et temporisés

3.6.5.1 Introduction

Il existe deux principales familles d'extension temporelle des réseaux de Pétri : les réseaux de Pétri temporisés introduits par **Ramchandani** [61] et les réseaux de Pétri temporels introduits par **Merlin** [62] Pour les réseaux de Pétri temporisés, les temporisations ont d'abord été associées aux transitions (t-temporisés), puis aux places

(p-temporisés). La temporisation représente alors la durée minimale de tir ou le temps de séjour minimum d'un jeton dans une place (ou durée exacte avec une règle de fonctionnement au plus tôt).

Les RdP t-temporisés et p-temporisés sont équivalents et sont une sous-classe des réseaux de Pétri temporels. Concernant les réseaux de Pétri temporels, l'extension temporelle s'exprime sous la forme d'un intervalle associé principalement aux transitions (t-temporel, ou aux places (p-temporel). En ce qui concerne l'expressivité des réseaux de Pétri p-temporels et t-temporels, ces deux modèles sont incomparables. Enfin, les réseaux de Pétri t-temporels forment une sous classe des Time Stream Pétri Nets qui ont été introduits pour modéliser des applications multimédia. Le problème de la bornitude est indécidable pour ce type de RdP et les travaux sur ce modèle reportent les résultats de décidabilité (comme l'accessibilité) sur l'hypothèse de bornitude du réseau. La bornitude (et les autres problèmes) sont alors résolus par le calcul de l'espace d'états (quand celui-ci termine) [63].

3.6.5.2 Définition des réseaux de Pétri temporellement temporisés

Soit $\mathbb{T}$ un domaine temporel. Un RPTT sur $\mathbb{T}$ et de support Act, est un tuple $N = (P, T, F, \lambda, SIM, \pi)$ [9] tel que :

- $N = (P, T, D, B)$ est un RDP marqué
- Un alphabet Act est un ensemble fini ; nous supposons que $\pi \in Act$ (désignera l'action invisible, t dite aussi action silencieuse ou interne).
- L'étiquetage d'un RPTT est une fonction $\lambda : T \to Act \cup \{\pi\}$ Si alors t dite observable ou externe ; dans le cas contraire, t dite silencieuse ou invisible.
- $SIM : \times\mathbb{T}\infty$ est la fonction qui associée à chaque transition un intervalle statique de tir.
- $\pi : Act \to D$ est la fonction de durée statique, qui à chaque action associe sa durée statique
- Soit l'ensemble de tous les intervalles d'un RPTT tel que :
  $I(t) = [min, max]$ est l'intervalle associé à la transition et on note par :
  $I(t) = max$ les fonctions qui donnent respectivement la borne $\uparrow I(t) = min$, $\uparrow$ inférieure et la borne supérieure d'un intervalle [9].

3.6.5.3 Principe

L'idée fondatrice des réseaux de Pétri temporellement temporisé (pour Duration Action Time Pétri 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 « $\theta + min$ » et doit être tirée au plus tard à la date « $\theta + max$ », 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 avec $c\theta + min \leq \vartheta \leq \theta + max$, l'action associée se termine à la date $\vartheta + d$ Le tir de la transition marque le début d'exécution de l'action associée. Dans un réseau de Pétri 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 $\vartheta + d$, le jeton est lié au tir de la transition durant l'intervalle $[\vartheta, \vartheta + d]$ et il devient libre à l'instant $\vartheta + d$ (devient dans la partie droite de cette place) [9, 8, 64].

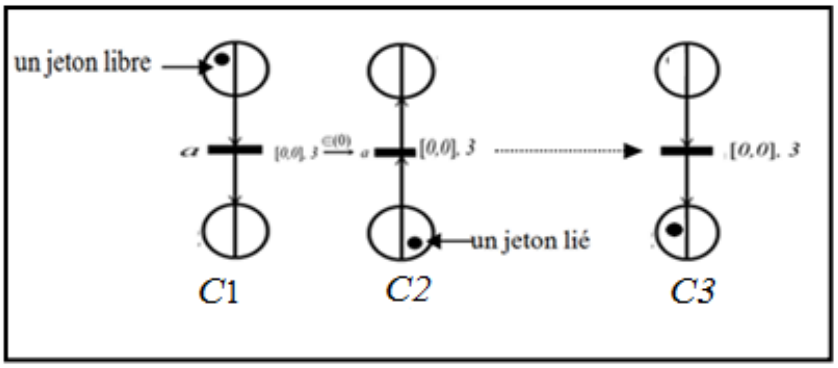


Figure 3.19: Le passage d'un état à un autre pour le jeton [64]

Le jeton qui se trouve dans la place amont de la transition n'est liée 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 3.19) (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 3.19) [9, 8].

3.6.6 Avantages et inconvénients des réseaux de Pétri

Le formalisme des réseaux de Pétri présente les avantages suivants :

- **Définition formelle :** Cette caractéristique efface toute ambigüité dans la spécification, car chaque modèle possède une sémantique bien définie.
- **Les réseaux de Pétri sont exécutables :** Il existe des programmes construits sur la définition formelle de la notation qui permettent d'interpréter les modèles en réseaux de Pétri et de simuler le fonctionnement du système en cours de spécification. Ceci permet une vision dynamique du système.
- **Expression puissante :** Les réseaux de Pétri sont adaptés pour décrire des comportements complexes, réactifs ou concurrents.
- **Support de vérification :** Les réseaux de Pétri disposent de nombreuses techniques de vérification automatique des propriétés génériques du système, comme la vivacité, ou des propriétés spécifique, comme l'existence d'invariants.
- **Représentation graphique :** cette qualité facilite l'interprétation et la compréhension des modèles [35].

Malgré ces multiples qualités, les réseaux de Pétri souffrent de certains points négatifs

- **Manque de structuration :** Plus le système est complexe et plus la taille du modèle produit est importante, et plus leur maîtrise est compliquée.
- **Structure de données :** Les réseaux places/transitions ne permettent pas de décrire la structure des données manipulées par le système.

Ces inconvénients ont été étudiés et commencent à se résoudre graduellement depuis le développement de la théorie des réseaux de Pétri de haut-niveau [35].

3.7 Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux des méthodes formelles, leurs techniques d'analyse et leur classification. Nous avons présenté aussi l'intérêt de leur intégration à l'IDM ,Pour cela plusieurs techniques de vérification sont proposées. Nous avons également présenté les réseaux de Pétri. Ils sont des outils graphiques et mathématiques puissants pour la modélisation, l'analyse et la vérification

des systèmes.

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 Pétri et nous avons également présenté le principe de ce réseau qui fait partie de notre étude.

Chapitre 4

Contribution

4.1 Introduction

Dans ce chapitre, nous présentons notre contribution qui consiste à transformer les diagrammes d'activités vers les RPTT en se basant sur l'approche de transformation de graphes. Nous commencerons, par établir un Meta-modèles associés au diagramme d'activités et un deuxième méta modèle associé au RPTT. Ensuite, nous proposons nos règles de transformation avec le langage de transformation ATL qui permet la transformation de modèles en faisant une présentation graphique avec GMF. À la fin nous avons présenté cas d'études afin d'illustrer notre approche de transformation.

Cette transformation de graphes est réalisée à l'aide de l'outil de modélisation EMF.

4.2 Manipuler des modéles avec EMF

Eclipse Modeling Framework (EMF) EMF existe depuis 2002, est un framework qui traite des modèles c'est-à-dire qu'EMF offre à ses utilisateurs un cadre de travail pour la manipulation des modèles. EMF permet de stocker les modèles sous forme de fichier pour en assurer la persistance. EMF permet également de traiter différents types de fichiers : conformes à des standards reconnus (UML, XML, XMI) et aussi sous des formes spécifiques (code Java) ou tout simplement sur mesure (au bon gré du concepteur). [4]

4.2.1 Objectif d'EMF

L'objectif général d'EMF est de proposer un outillage qui permet de passer du modèle au code Java automatiquement. Pour cela le framework s'articule autour d'un modèle (le Core Model). EMF va proposer plusieurs services :

1. la transformation des modèles d'entrées, présentés sous diverses formes, en Core Model.
2. la gestion de la persistance du Core Model.
3. la transformation du Core Model en code Java. EMF peut de base gérer en entrée des modèles présentés sous trois formats :

- UML
- XMI
- Code Java Annoté

Dans EMF, il est possible de définir un méta-modèle et de générer les interfaces afin de pouvoir manipuler les instances du méta-modèle dans Eclipse.

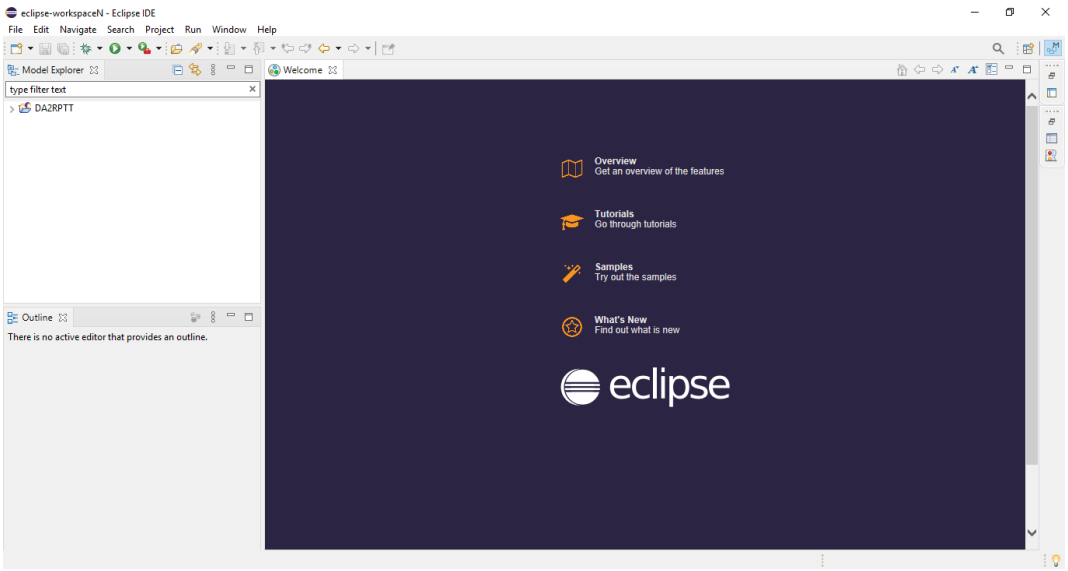


Figure 4.1: Interface EMF

4.3 Les Méta-modèles

Cette section présente les méta-modèles sources qui sont les diagrammes d'activité, et les méta-modèles cibles qui sont les réseaux de Petri temporelles.

4.3.1 Spécification du méta-modèle source

Dans cette partie, nous spécifions notre métamodèle de diagramme d'activité d'UML. La définition des métamodèle sa été réalisée sous l'environnement EMF. Figure 4.2

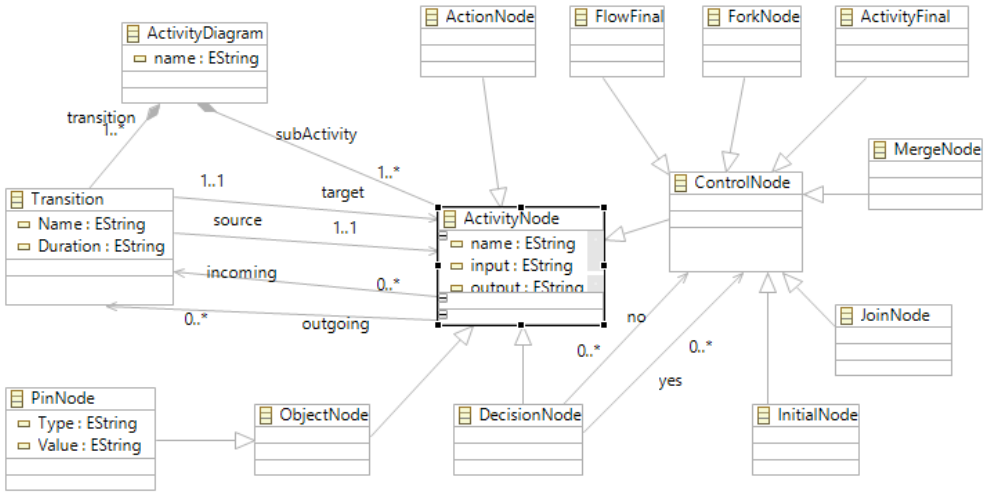


Figure 4.2: Méta-modèle du diagramme d'activité

- **Les classes**
  - **Class ActivityDiagram :** cette classe est composé de Classe Action Node et Transition.
  - **Classe Action Node :** cette classe représente une opération atomique non décomposable.elle est représentée visuellement par un ovale qui contient sa description textuelle.
  - **Classe control Node :** elle représente un nœud d'activité abstrait utilisé pour coordonner les flux entre les nœuds d'une activité. Elle est la classe mère des classes des nœuds finaux (final node, flow final) , initial node, forknode et décision node.
  - **Classe activityNode :** est une classe abstraite permettant de représenter les étapes le long du flux d'une activité . Elle possède un attribut de type String.
  - **Classe initial node :** représente le début d'un diagramme d'activité.
  - **Class transition :** cette classe représente une opération de transition. Elle possède un attribut de type String et duration de type string.
  - **Classe décision Node :** cette classe spécifie les différentes alternatives possibles. elle a un arc entrant et deux ou plusieurs arcs sortants,
  - **Classe forkNode :** elle représente un nœud de synchronisation qui possède un seul arc entrant et plusieurs arcs sortants qui doivent être déclenchés simul-

tanément.
  - **Classe MergeNode :** cette classe rassemble plusieurs flots entrant en un seul flot sortant
  - **Classe JoinNode :** elle représente un nœud de synchronisation qui ne peut être franchit que lorsque toute les transitions en entrée ont été déclenchés.
  - **Classe Final Node :** indique une terminaison avec succès. Elle possède un ou plusieurs arcs entrants et aucun arc sortant.
  - **Classe Object Node :** elle représente une méta-classe abstraite permettant de définir les flux d'objets dans les diagrammes d'activités.
  - **Classe Pin Node :** elle représente un nœud d'objet connecté en entrée ou en sortie d'une activité.
- **Les associations** Chaque association de méta-modèle possède un attribut Name de type String. Elle relie chaque instance de classe source avec tell instance de la classe Les associations :
  1. transition
  2. subactivity
  3. incoming
  4. outgoing
  5. source

4.3.2 Spécification du métamodèle cible

Nous présentons maintenant le métamodèle cible.

- PetriNet est composé de transitions, de Place et d'Arcs. L'entité PetriNet, ainsi que la Transition et Place, héritent de l'entité Elément.
- Arc est une entité abstraite qui est associée à un attribut «weight». Chaque arc est l'un des Genre TransToPlace ou PlaceToTrans
- Un TransToPlaceconnecte une transition à un Place,alors qu'un PlaceToTransition connecte un Place à une Transition. Un espace peut avoir plusieurs PlaceToTransition sortants et plusieurs TransitionToPlace entrants.
- Une transition peut avoir plusieurs PlaceToTrans entrants et plusieurs TransToPlace. Chaque TransToPlace a une transition source et un lieu cible. De la même façon, chaque PlaceToTrans a une place source et une transition cible. Comme l'illustre la figure 4.4

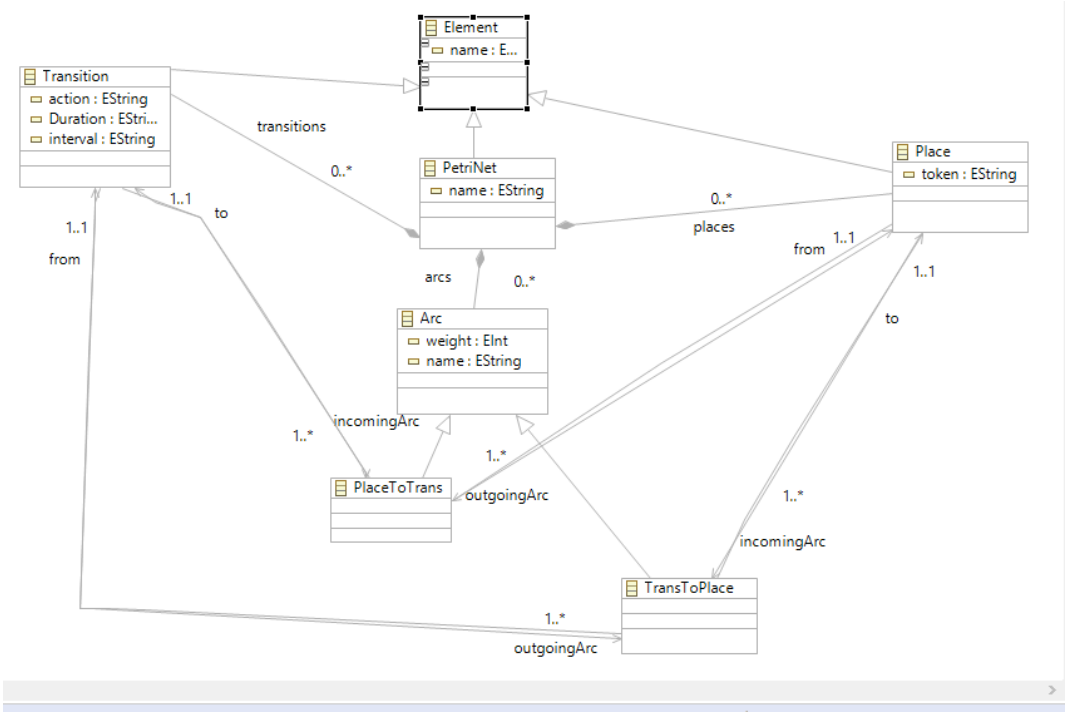


Figure 4.3: Méta-modèle du RPTT

- **Les associations**

1. incomingArc
2. outgoingArc
3. from
4. transition

4.4 La génération d'un outil pour la transformation d'un Diagramme d'activité vers un RPTT

- **Création projet ATL :**
  Après la théorie, commençons à créer le projet.
  – Pour créer un nouveau projet ATL, vous devez aller **File > New > Other**

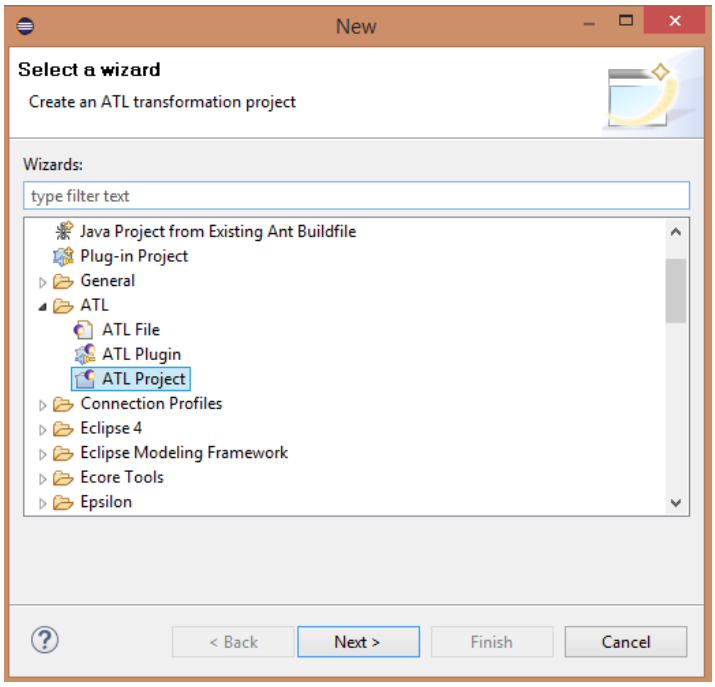


Figure 4.4: Méta-modèle du RPTT

– Nommer le projet (par exemple : AD2RPTT) après cliquer sur **Finish**.

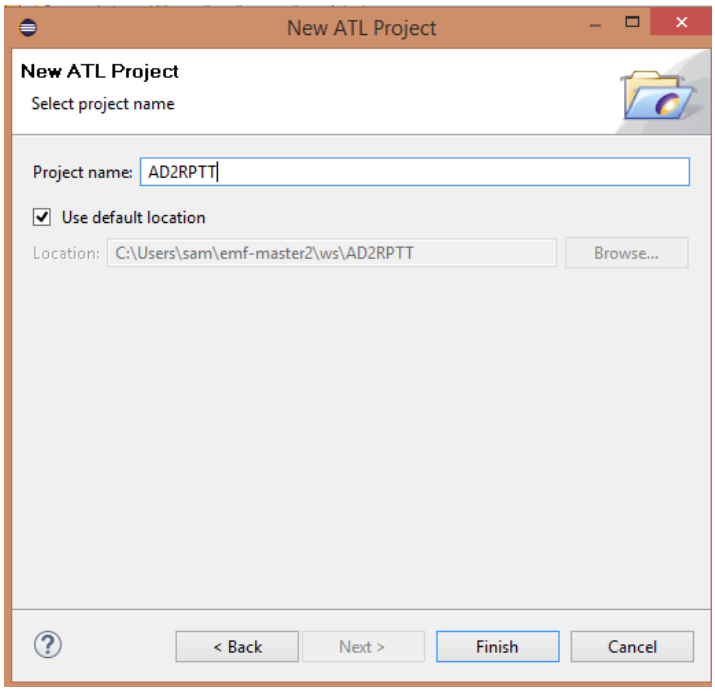


Figure 4.5: Nom de projet ATL

- **Création d'un dossier dans le projet :**
  - Cliquer droit sur le projet > **New** > **Folder**
  - Nomer le dossier et cliquer **Finish**

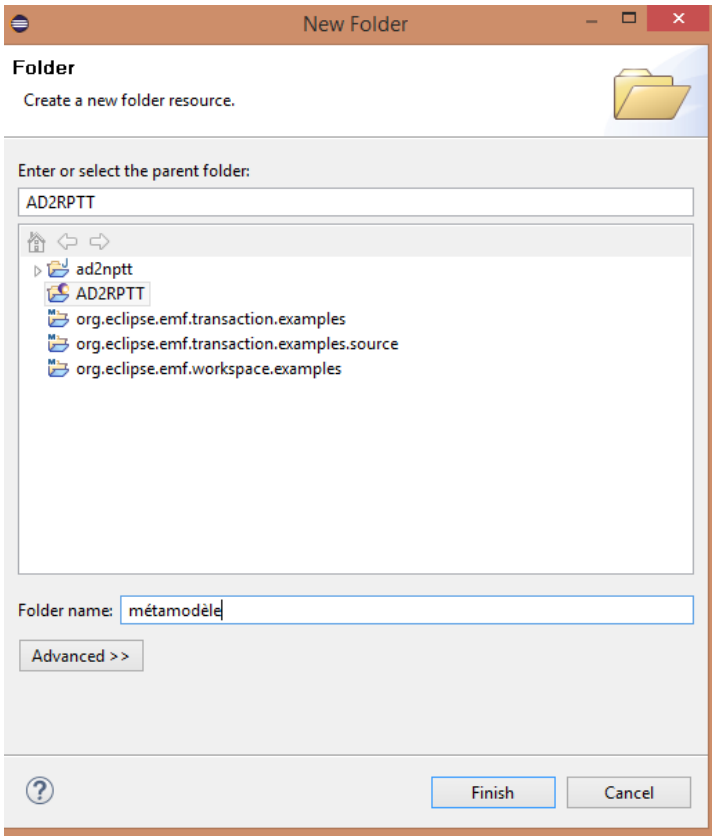


Figure 4.6: Création d'un dossier dans projet

– Répéter la procédure de création des dossiers dans le projet

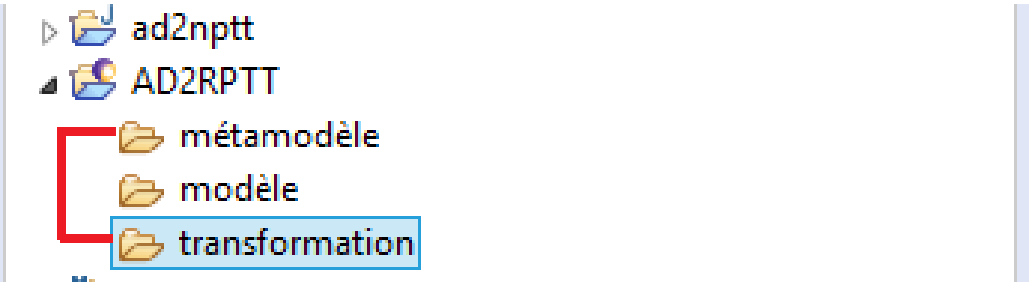


Figure 4.7: Création des dossiers

- **Créer un fichier de type Ecore :**
  - Cliquer droit sur le modèle **ecoreInitializeEcoreDiagrame**
  - Sélectionner métamodèle et nommer le fichier et cliquer sur **Finish**

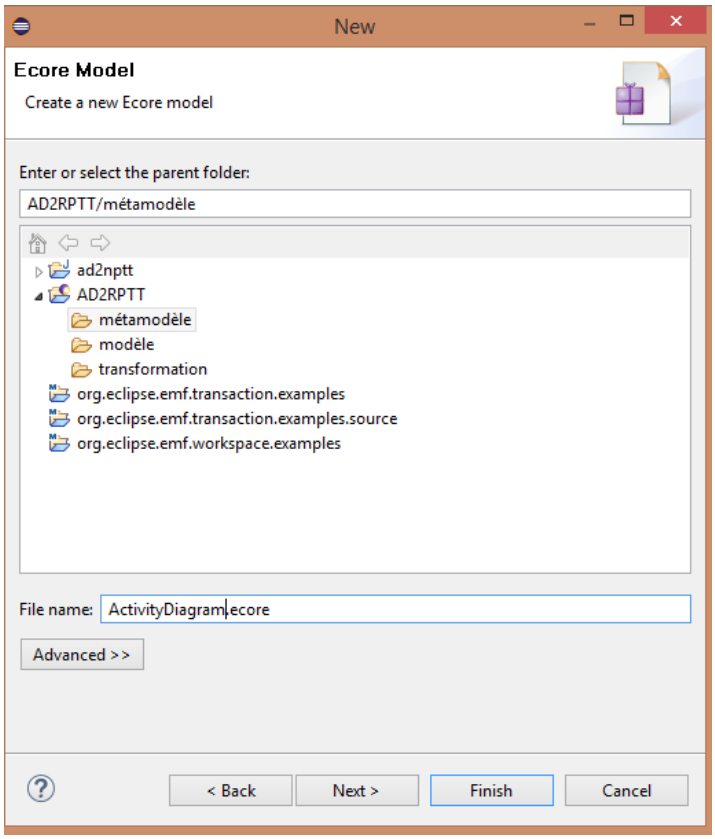


Figure 4.8: Création fichier de type ecore

- **Définir les éléments de metamodèles**

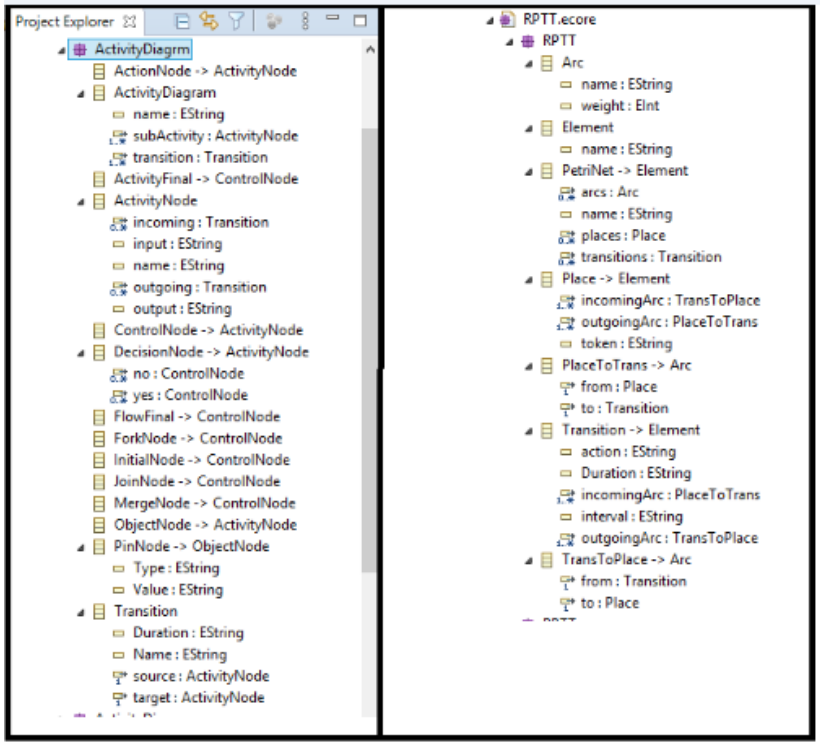


Figure 4.9: Les Éléments de MetaModele AD et RPTT

- **Génération de modèle conforme à méta-modèles de AD**
  On va générer le modèle conformément au méta-modèle des digramme d'acttivité. Et créer les instances dynamiques et choisir le nom de modèle et le dossier de l'emplacement comme la figure suivantes :

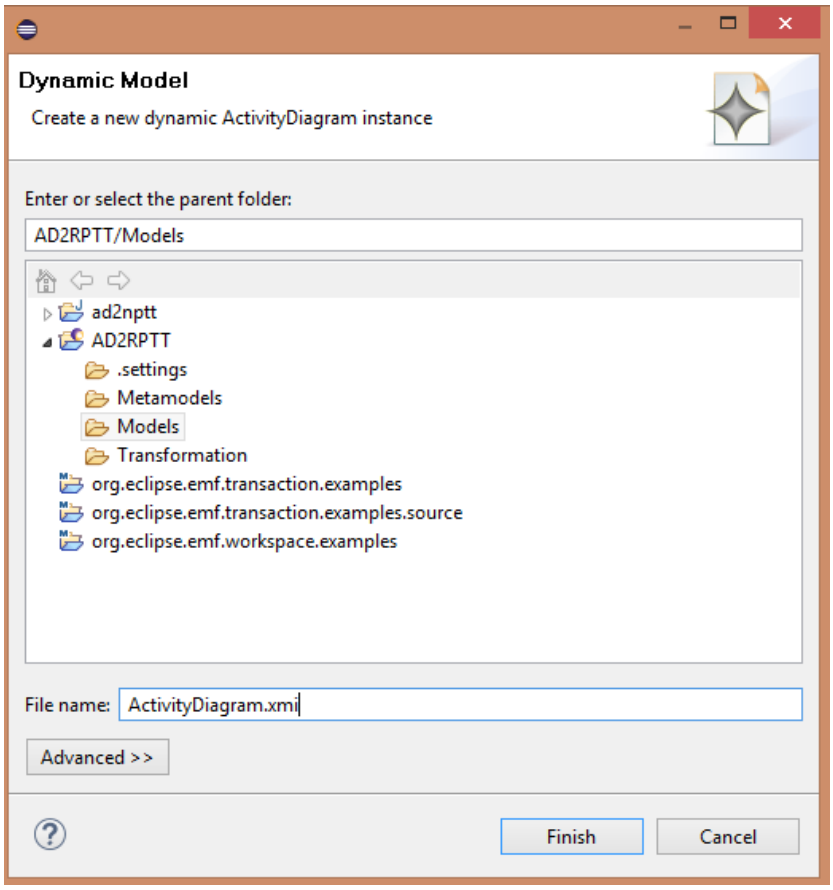


Figure 4.10: Choisir le nom de modèle et le dossier de l'emplacement

- La figure de modèle digramme d'activité généré :

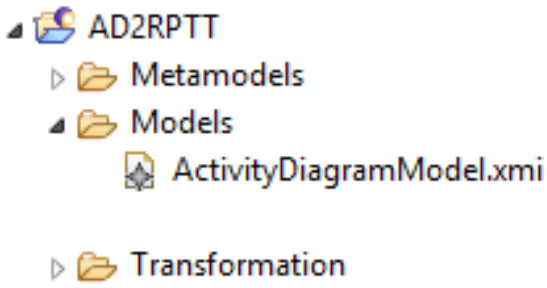


Figure 4.11: Modèles généré

- **La création des éléments de modèle à transformer**
  La figure Figure 4.12 : décrit un exemple, au format XMI instancié à partir de notre outil basant sur le métamodèl proposé pour les ADs,

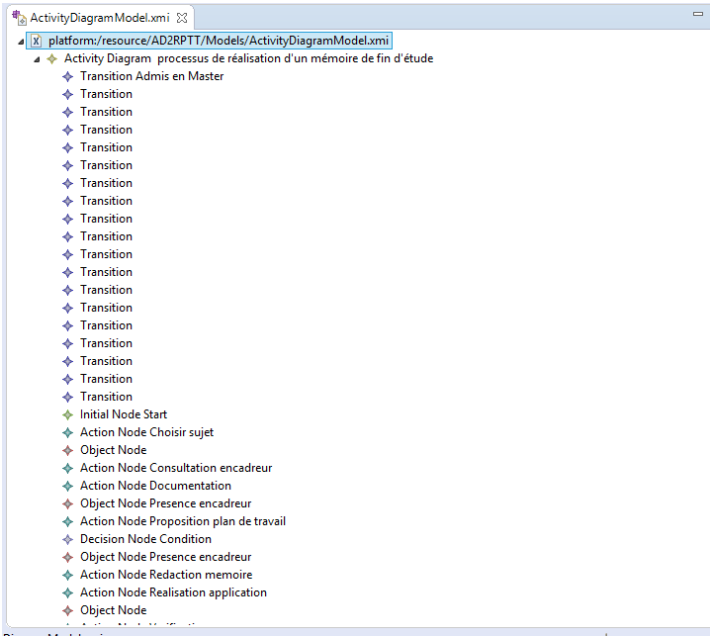


Figure 4.12: La création des éléments de modèle à transformer

- **Transformation « Model To Model »**
  Il existe deux type de transformation ATL : « Model To Model » et « Model To Text », dans notre cas, nous utilisons le permier.
  Maintenant que nous avons représenté ce que nous avons (AD, la source) et ce que nous voulons obtenir (RPTT, la cible), nous pouvons nous concentrer sur le cœur de la transformation : le code ATL. Ce code va faire correspondre une partie de la source avec une partie de la cible.
  Ce que nous voulons dans notre exemple, nous avons d'abord besoin d'un fichier dans lequel insérer ce code. Créez donc un nouveau fichier ATL, en allant dans **File > New >Other..., ATL > ATL File.**
  - Nommer le fichier et cliquer sur Finish.

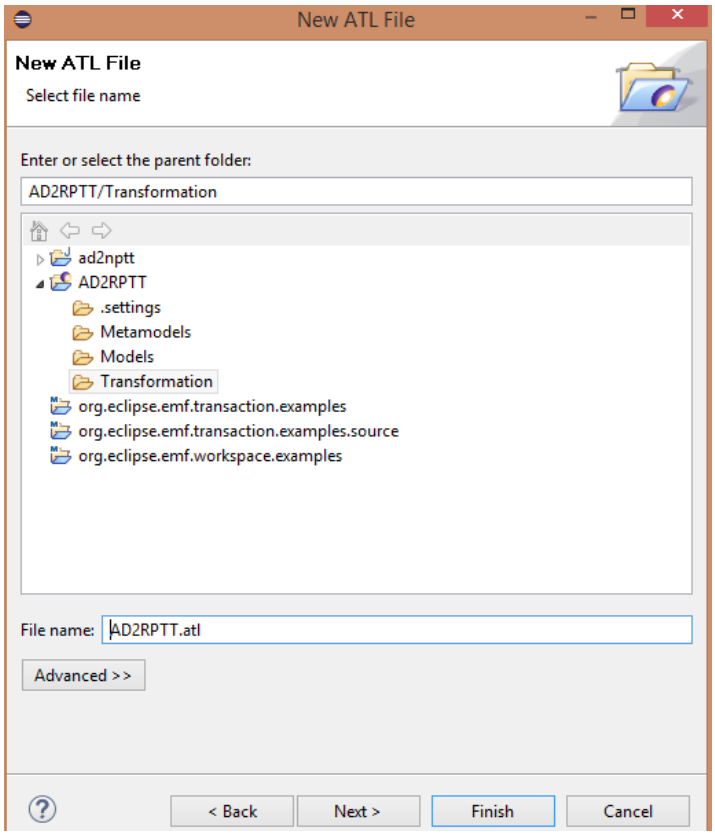


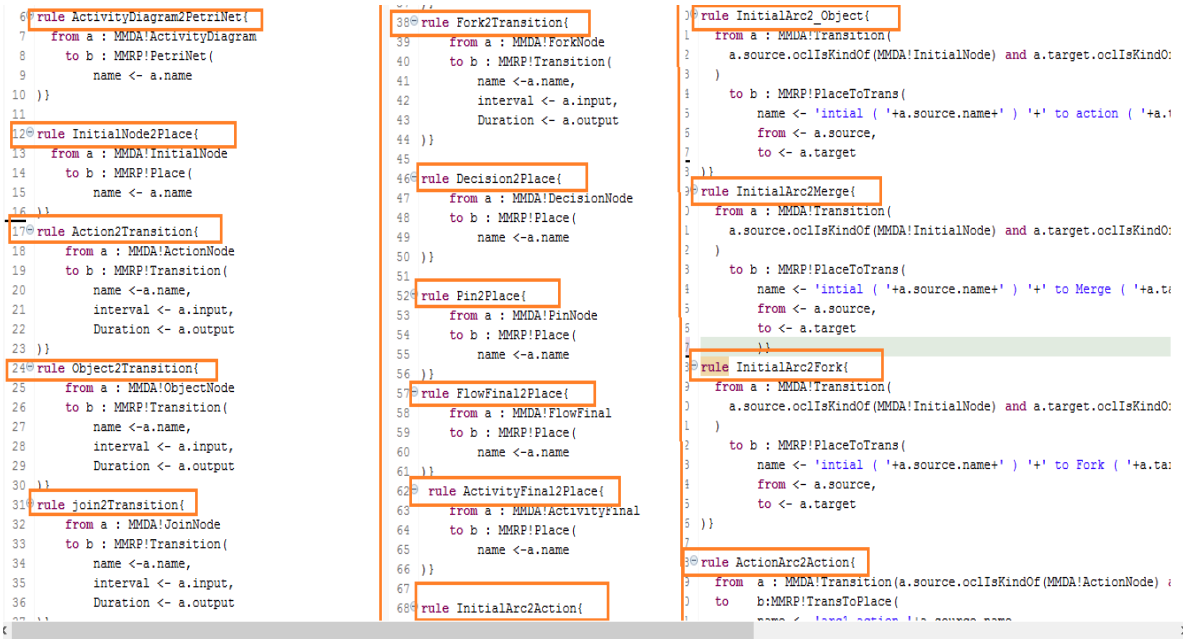
Figure 4.13: Création fichier ATL

- **Transformation des diagramme d'activité vers le réseau de Pétri temporellement temporisé :**
- **Implémentation des règle ATL :**
  Tout d'abord, nous ajoutons deux lignes en haut du fichier, une pour chaque métamodèle, afin que l'éditeur puisse utiliser l'auto-complétion et la documentation lorsque nous tapons du code concernant les deux métamodèles :
  Ensuite, nous disons à ATL que nous avons des diagrammes d'activité et que nous voulons que des Rptt sortent

```
1 -- @path MMDA=/AD2RPTT/Metamodels/ActivityDiagrm.ecore
2 -- @path MMRP=/AD2RPTT/Metamodels/RPTT.ecore
3 module MyRules;
4 create OUT : MMRP from IN : MMDA;
5
```

Figure 4.14: Éditeur de fichier ATL

– L'ensemble de ces transformations établissent des règles comme cela a été vu dans la figure suivantes pour établir une règle de base en ATL
– Regardons maintenant l'implémentation de la règle de transformation les classes InitialNode, Action, Object et join. (voir les figures) :



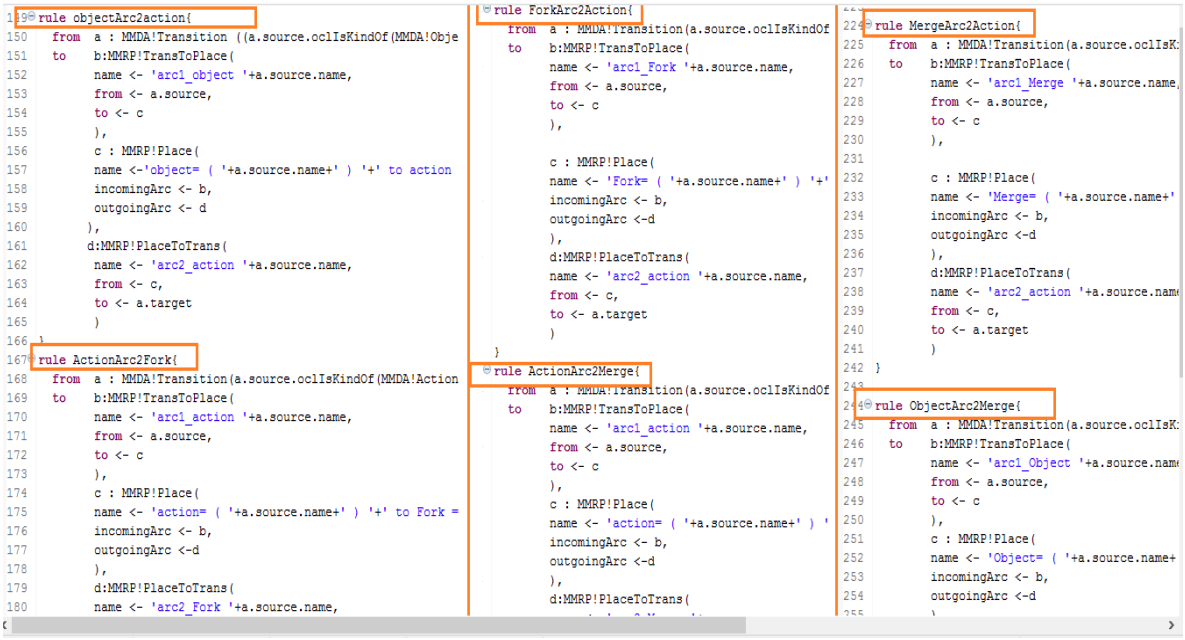
```
rule ActivityDiagram2PetriNet{
    from a : MMDA!ActivityDiagram
    to b : MMRP!PetriNet(
        name <- a.name
)}

rule InitialNode2Place{
    from a : MMDA!InitialNode
    to b : MMRP!Place(
        name <- a.name
)}
rule Action2Transition{
    from a : MMDA!ActionNode
    to b : MMRP!Transition(
        name <-a.name,
        interval <- a.input,
        Duration <- a.output
)}
rule Object2Transition{
    from a : MMDA!ObjectNode
    to b : MMRP!Transition(
        name <-a.name,
        interval <- a.input,
        Duration <- a.output
)}
rule join2Transition{
    from a : MMDA!JoinNode
    to b : MMRP!Transition(
        name <-a.name,
        interval <- a.input,
        Duration <- a.output
```

```
rule Fork2Transition{
    from a : MMDA!ForkNode
    to b : MMRP!Transition(
        name <-a.name,
        interval <- a.input,
        Duration <- a.output
)}

rule Decision2Place{
    from a : MMDA!DecisionNode
    to b : MMRP!Place(
        name <-a.name
)}

rule Pin2Place{
    from a : MMDA!PinNode
    to b : MMRP!Place(
        name <-a.name
)}
rule FlowFinal2Place{
    from a : MMDA!FlowFinal
    to b : MMRP!Place(
        name <-a.name
)}
rule ActivityFinal2Place{
    from a : MMDA!ActivityFinal
    to b : MMRP!Place(
        name <-a.name
)}

rule InitialArc2Action{
```

```
rule InitialArc2_Object{
    from a : MMDA!Transition(
    a.source.oclIsKindOf(MMDA!InitialNode) and a.target.oclIsKindO
    )
    to b : MMRP!PlaceToTrans(
        name <- 'intial ( '+a.source.name+' ) '+' to action ( '+a.
        from <- a.source,
        to <- a.target
)}
rule InitialArc2Merge{
    from a : MMDA!Transition(
    a.source.oclIsKindOf(MMDA!InitialNode) and a.target.oclIsKindO
    )
    to b : MMRP!PlaceToTrans(
        name <- 'intial ( '+a.source.name+' ) '+' to Merge ( '+a.t
        from <- a.source,
        to <- a.target
        )}
rule InitialArc2Fork{
    from a : MMDA!Transition(
    a.source.oclIsKindOf(MMDA!InitialNode) and a.target.oclIsKindO
    )
    to b : MMRP!PlaceToTrans(
        name <- 'intial ( '+a.source.name+' ) '+' to Fork ( '+a.ta
        from <- a.source,
        to <- a.target
)}

rule ActionArc2Action{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!ActionNode)
    to    b:MMRP!TransToPlace(
```

Figure 4.15: Les règles de transformation (1/3)

La suite des règles de transformation

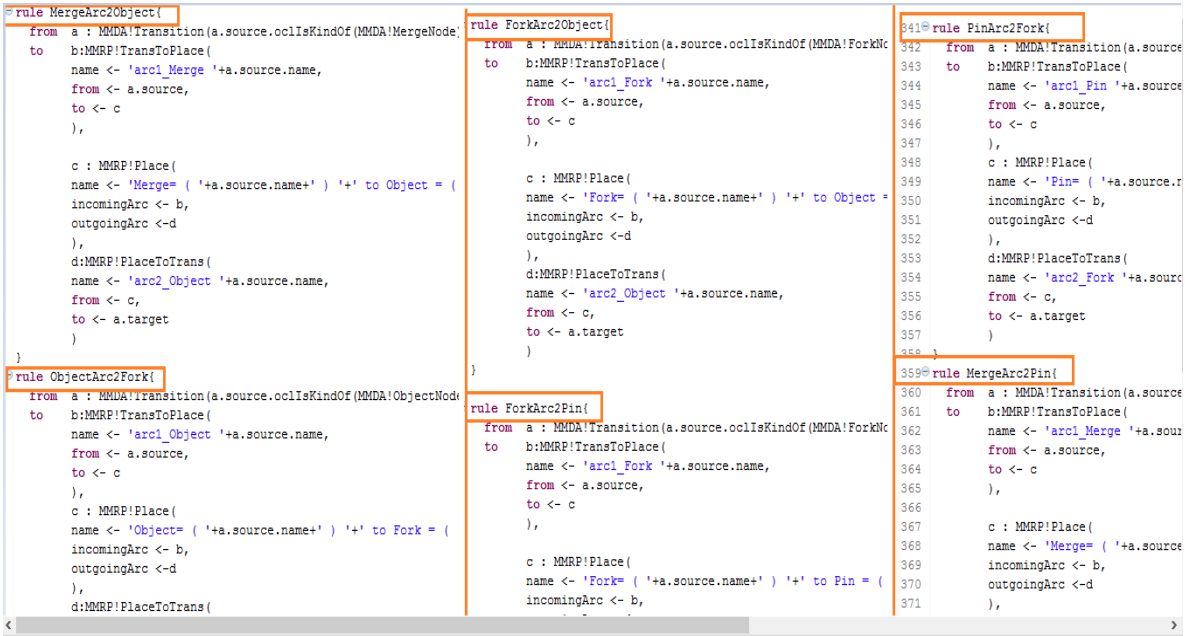


```
rule objectArc2action{
    from  a : MMDA!Transition ((a.source.oclIsKindOf(MMDA!Obje
    to    b:MMRP!TransToPlace(
          name <- 'arc1_object '+a.source.name,
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <-'object= ( '+a.source.name+' ) '+' to action
          incomingArc <- b,
          outgoingArc <- d
         ),
         d:MMRP!PlaceToTrans(
          name <- 'arc2_action '+a.source.name,
          from <- c,
          to <- a.target
          )
}
rule ActionArc2Fork{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!Action
    to    b:MMRP!TransToPlace(
          name <- 'arc1_action '+a.source.name,
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <- 'action= ( '+a.source.name+' ) '+' to Fork =
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_Fork '+a.source.name,
```

```
rule ForkArc2Action{
    from  a : MMDA!Transition(a.source.oclIsKindOf
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Fork '+a.source.name,
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Fork= ( '+a.source.name+' ) '+'
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_action '+a.source.name,
          from <- c,
          to <- a.target
          )
}
rule ActionArc2Merge{
    from  a : MMDA!Transition(a.source.oclIsKindOf
    to    b:MMRP!TransToPlace(
          name <- 'arc1_action '+a.source.name,
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <- 'action= ( '+a.source.name+' ) '
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
```

```
rule MergeArc2Action{
    from  a : MMDA!Transition(a.source.oclIsK
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Merge '+a.source.name,
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Merge= ( '+a.source.name+'
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_action '+a.source.nam
          from <- c,
          to <- a.target
          )
}
rule ObjectArc2Merge{
    from  a : MMDA!Transition(a.source.oclIsK
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Object '+a.source.nam
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <- 'Object= ( '+a.source.name+
          incomingArc <- b,
          outgoingArc <-d
```

FIGURE 4.16: Les règles de transformation (2/3)



```
rule MergeArc2Object{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!MergeNode
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Merge '+a.source.name,
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Merge= ( '+a.source.name+' ) '+' to Object = (
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_Object '+a.source.name,
          from <- c,
          to <- a.target
          )
}
rule ObjectArc2Fork{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!ObjectNod
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Object '+a.source.name,
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <- 'Object= ( '+a.source.name+' ) '+' to Fork = (
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
```

```
rule ForkArc2Object{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!ForkNo
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Fork '+a.source.name,
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Fork= ( '+a.source.name+' ) '+' to Object =
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_Object '+a.source.name,
          from <- c,
          to <- a.target
          )
}

rule ForkArc2Pin{
    from  a : MMDA!Transition(a.source.oclIsKindOf(MMDA!ForkNo
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Fork '+a.source.name,
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Fork= ( '+a.source.name+' ) '+' to Pin = (
          incomingArc <- b,
```

```
rule PinArc2Fork{
    from  a : MMDA!Transition(a.source
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Pin '+a.source
          from <- a.source,
          to <- c
          ),
          c : MMRP!Place(
          name <- 'Pin= ( '+a.source.n
          incomingArc <- b,
          outgoingArc <-d
          ),
          d:MMRP!PlaceToTrans(
          name <- 'arc2_Fork '+a.sourc
          from <- c,
          to <- a.target
          )
}
rule MergeArc2Pin{
    from  a : MMDA!Transition(a.source
    to    b:MMRP!TransToPlace(
          name <- 'arc1_Merge '+a.sour
          from <- a.source,
          to <- c
          ),

          c : MMRP!Place(
          name <- 'Merge= ( '+a.source
          incomingArc <- b,
          outgoingArc <-d
          ),
```

FIGURE 4.17: Les règles de transformation (3/3)

- **Configuration de fichier ATL :**

Pour que la transformation de notre modèle s'exécute il faut d'abord créer une configuration pour notre module ATL et lui donné le méta modèle et le modèle que nous voulons transformer.

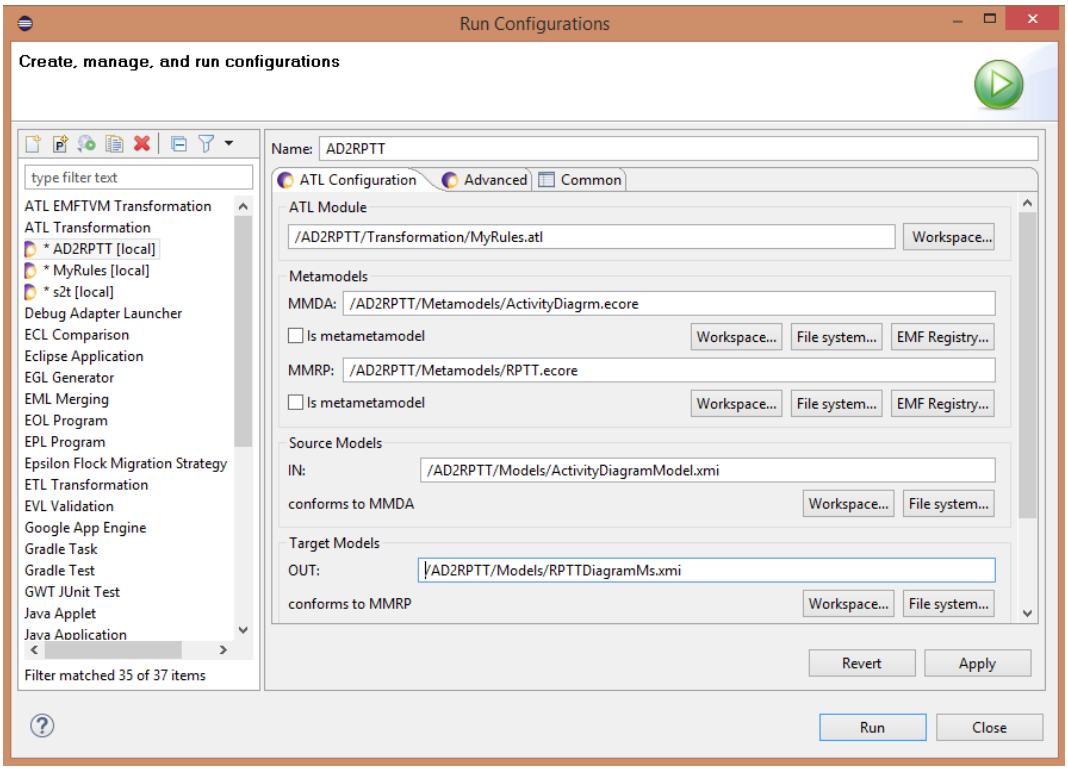


Figure 4.18: Configuration de fichier ATL

4.5 Partie Graphique (GMF)

Dans cette partie nous utilisons l'outil Graphical Modeling Framework :

4.5.1 Graphical Modeling Framework (GMF)

C'est un framework de l'environnement de travail Eclipse. Il fournit une infrastructure permettant l'exécution d'éditeurs graphiques basés sur les frameworks EMF et GEF.

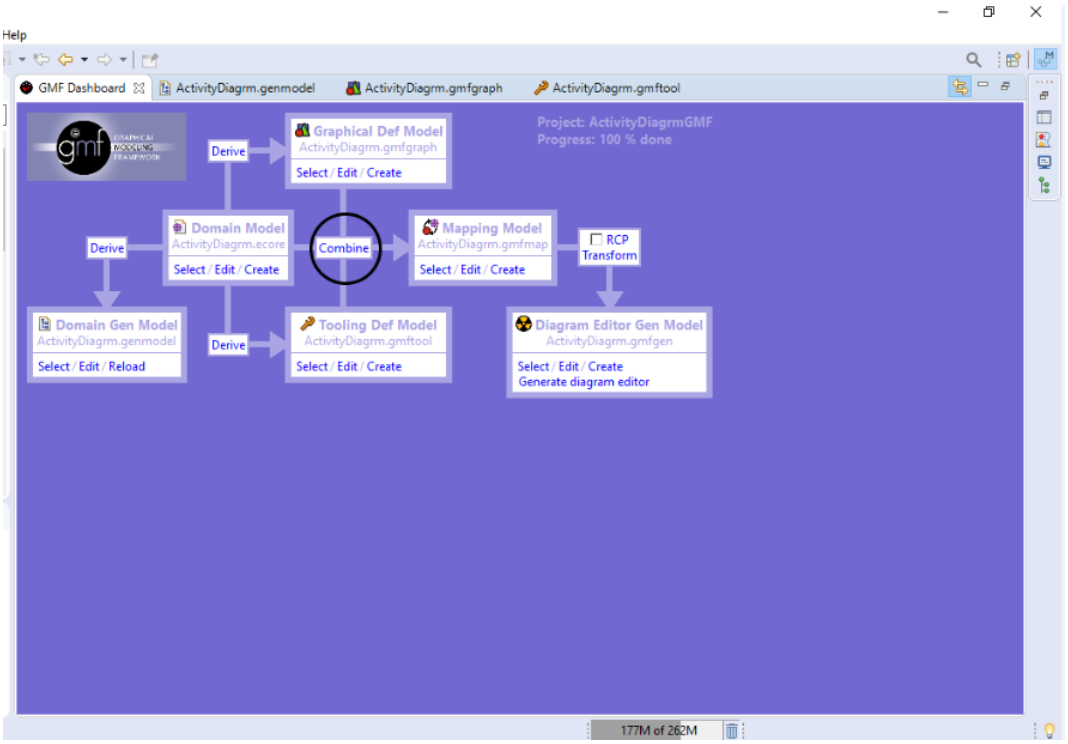


Figure 4.19: DashBoard GMF

Comme vous pouvez le voir, la génération d'un éditeur graphique GMF comprend six étapes :

- **Domain model :** le métamodèle que nous voulons utiliser pour créer l'éditeur graphique. Pour ce métamodèle, vous ont le choix entre plusieurs types de métamodèles : code Java annoté, modèle Ecore, classe Rose modèle, modèle UML ou schéma XML). Dans notre travail , nous utiliserons le métamodèle.
- **Domain Gen Model (.genmodel) :** ce fichier est utilisé pour générer le code du modèle de domaine avec EMF (c'est le Fichier EMF genModel)
- **Graphical Def Model (.gmfgraph) :** ce fichier permet de définir les éléments graphiques de votre domaine modèle
- **Tooling Def Model (.gmftool) :** ce fichier permet de définir la palette d'outils que vous pouvez utiliser dans le éditeur graphique.
- **Mapping Model (.gmfmap) :** ce fichier relie le modèle de domaine, le modèle graphique (.gmfgraph) et le modèle d'outillage (.gmftool).
- **Diagram Editor Gen Model (.gmfgen) :** ce fichier final nous a servi à générer l'éditeur graphique GMF dans ajout au code EMF généré par le fichier .genmodel.

4.5.2 Création projet GMF

- Depuis le menu « **File** », cliquer sur le sous-élément « **Project...** » :
- Sélectionner ensuite l'élément « New GMF Project »
- Cliquer sur le bouton « Next> », nommer ensuite le projet :
- Cliquer sur le bouton « Finish », nous obtenons finalement :

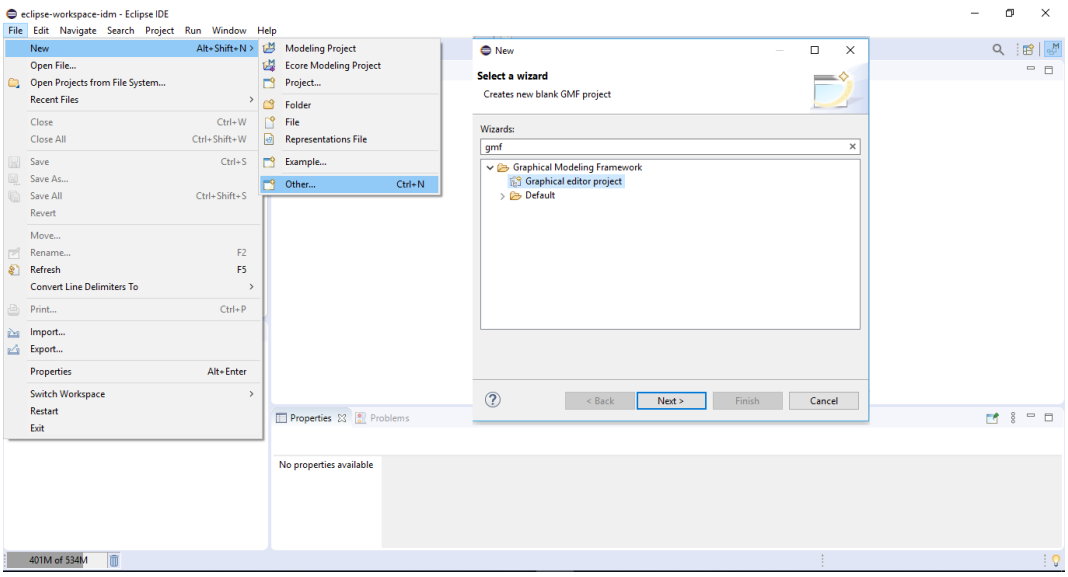


Figure 4.20: Création projet GMF

4.5.3 Génération de l'éditeur graphique

Pour cela nous allons utiliser le « Dashboard ».

- Sélectionner Domain Model (Fichier : *.ecore) :
- Sélectionner le projet «activityDiagram.ecore » :

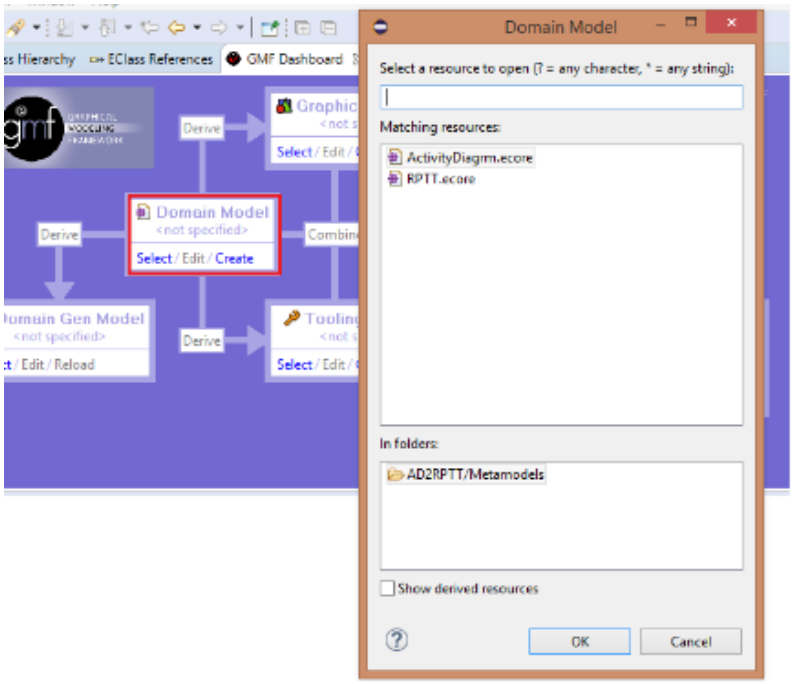


Figure 4.21: Sélection Ecore model

- **Domain Gen Model :**
  Nous devons générer le fichier .genmodel. Pour cela, il suffit de :
  - Cliquer sur **Derive** à gauche du Domain Model.
  - Saisir dans le champ « **File name** », la valeur « **ActivityDiagram.genmodel** », puis cliquer sur le bouton « **Next** » :

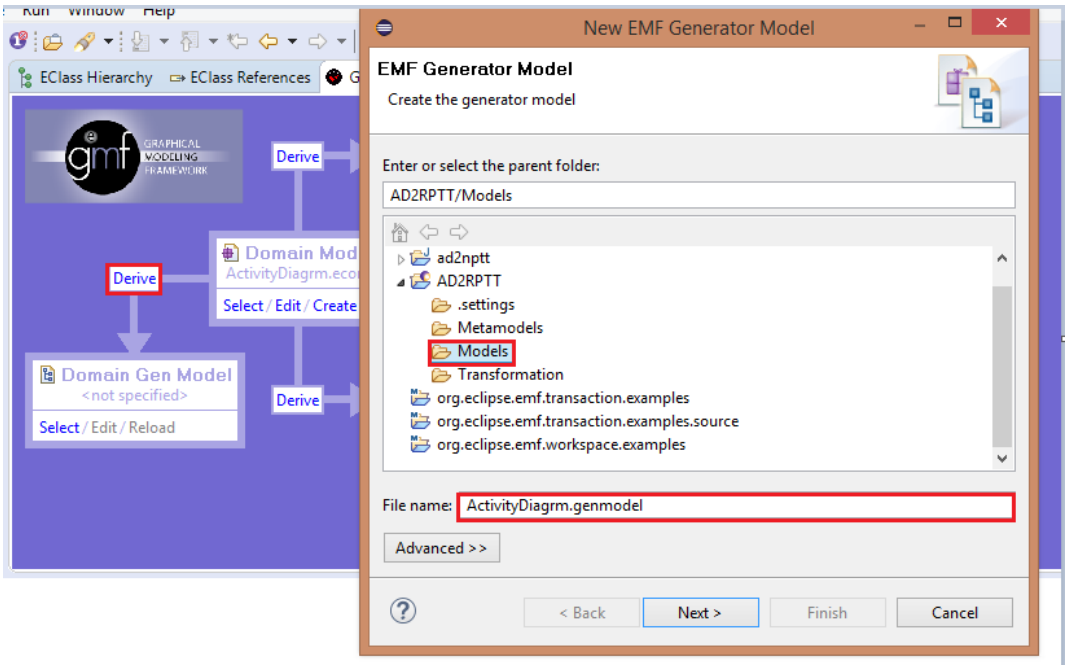


Figure 4.22: Création fichier.genmodel

– Sélectionner un modèle basé sur « **ecore** », puis cliquer sur le bouton « **Next** > ».
– Cliquer sur le bouton « **Load** » puis sur le bouton « **Next** > ».
– Ne pas modifier « **New EMF Generator Model - Package Selection** » , voir ces étapes dans la figure suivante.

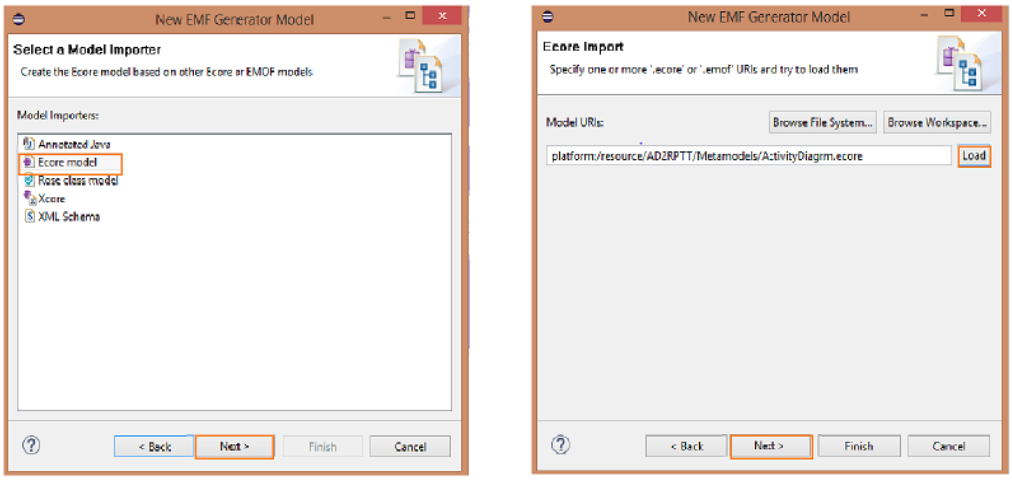


Figure 4.23: Sélection Ecore model et Load Ecore model

– Cliquer sur le bouton « **Finish** ».
Le fichier **ActivityDiagram.genmodel** apparaît sur le projet et le Dash-Board. Ouvrez et cliquez sur la flèche près de ActivityDiagram.

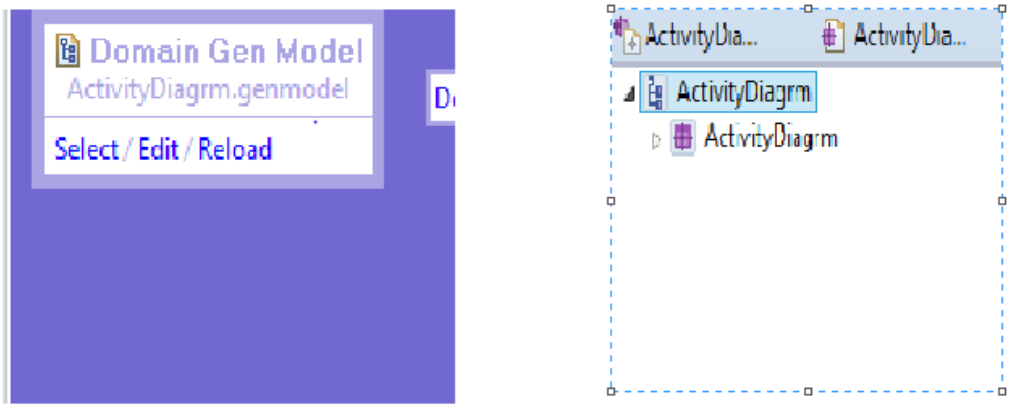


Figure 4.24: Création fichier ActivityDiagram.genmodel.

- **Générer le code :**
  – Le fichier «**ActivityDiagram.genmodel**» est un fichier de génération, vous pouvez l'ouvrir en édition si nécessaire.
  Utiliser la commande « **Generate All** » du menu contextuel de l'éditeur pour générer le code :
  – Cliquer sur Generate All. Cela aura pour effet de générer 3 nouveaux projets et Le code généré est placé dans «ActivityDiagram.gmf\src » et dans trois projets Eclipse

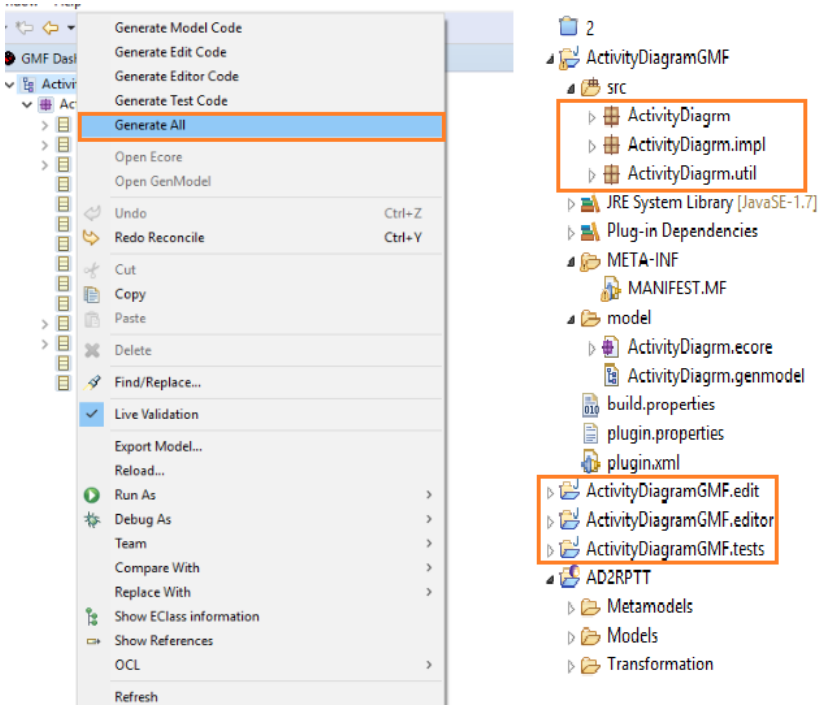


Figure 4.25: Génération code de java EMF

- **Générer « Tooling Def Model » (Fichier : *.gmftool) :**
  - Cliquer « **Derive** » :
  - Cliquer sur le bouton « **Next** » :
  - Dans « **New - Domain Model** », cliquer « **Load** » et sélectionner « **Activity diagram** », puis cliquer sur le bouton « **Next** » :

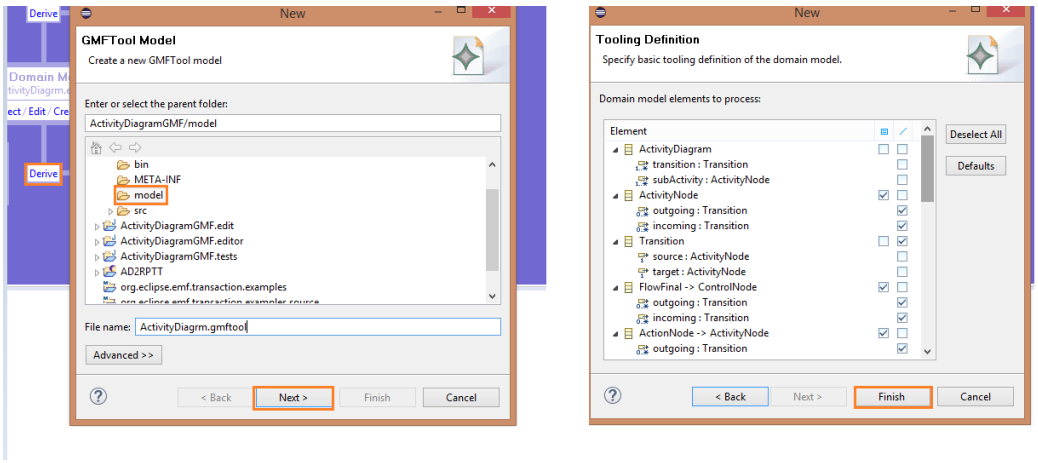


Figure 4.26: Création GMF model

Le contenu du fichier «activity diagram.gmftool » et sur le DashBoard est le suivant :

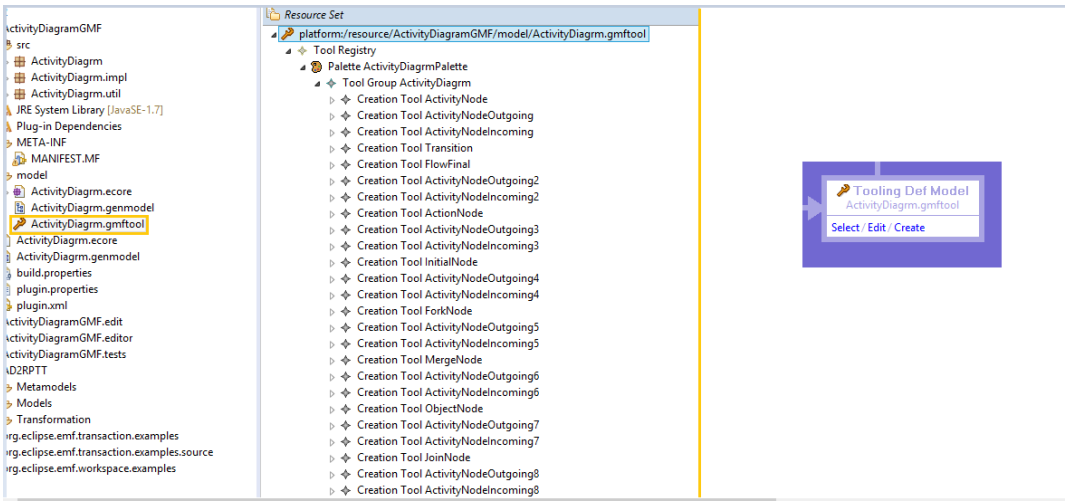


Figure 4.27: Fichier activitydigram.gmftool

- **Éditer « Graphical Def Model » :**
  Maintenant, il faut générer le fichier permettant de définir la forme et l'apparence des noeuds et Labels du futur diagramme.

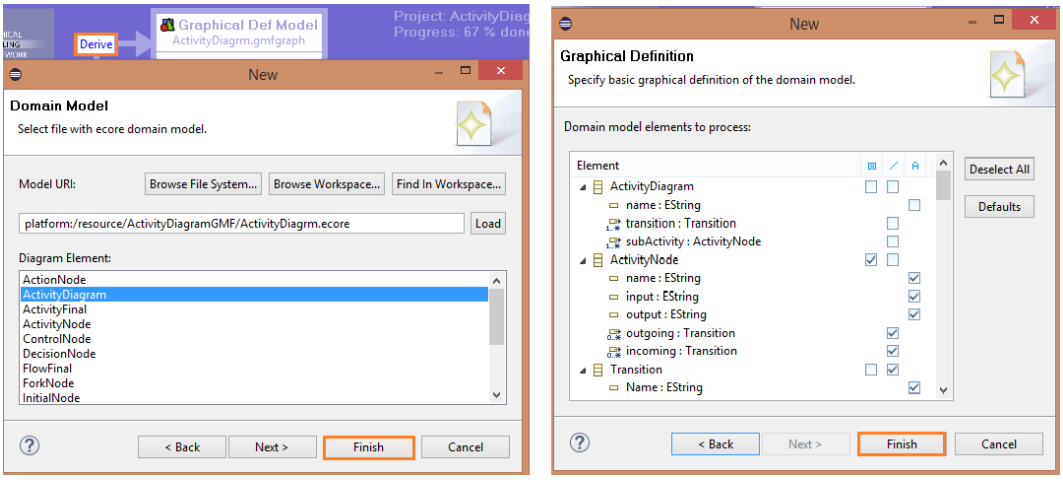


Figure 4.28: Sélection les éléments de Domain model

- Le resultat

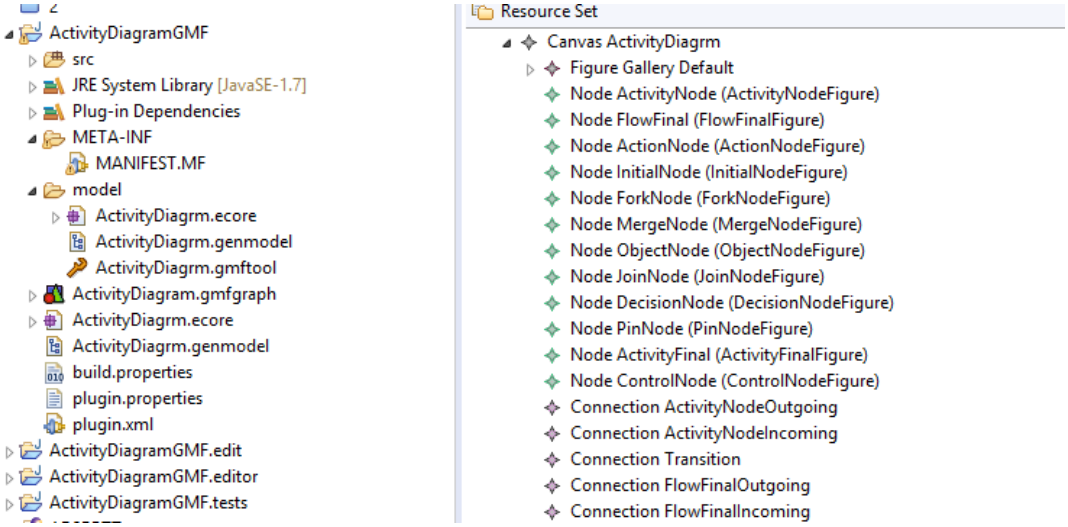


Figure 4.29: ActivityDiagram.gmfgraph en début

- **Générer « Mapping Model » (Fichier : *.gmfmap) :**
  Le modèle de mappage (.gmfmap) permet de lier les fichiers .genmodel, .gmfgraph et .gmftool. Dans le DashBoard.
  - Cliquer sur « Combine » :
  - Cliquer sur le bouton « Next » :
  - Cliquer « Load » puis sélectionner « ActivityDiagram».

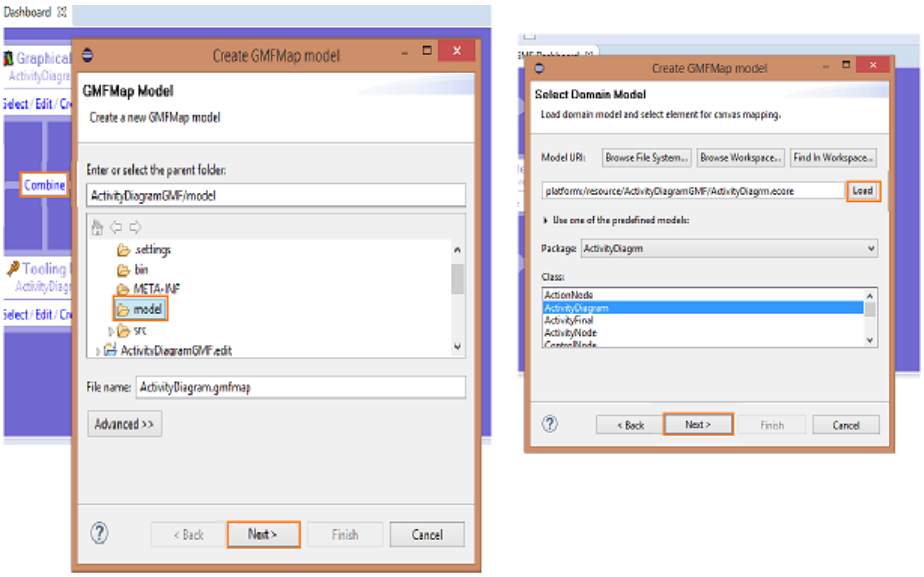


Figure 4.30: Création gmfmap

Le contenu du fichier « ActivityDiagram.gmfmap » est le suivant :

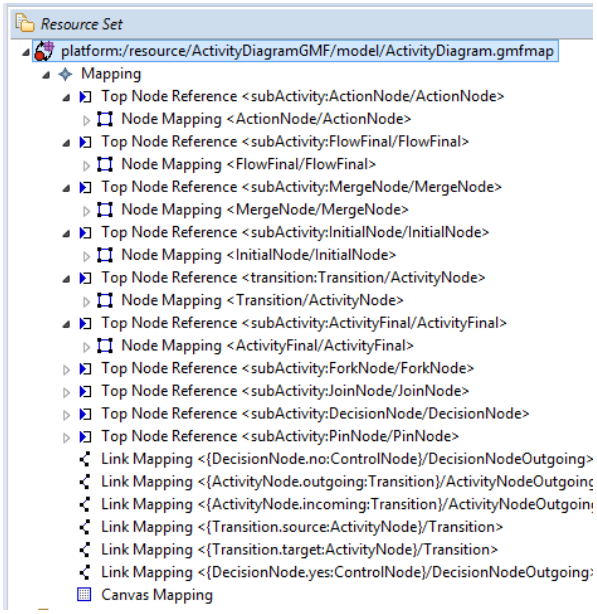


Figure 4.31: Fichier ActivityDiagram.gmfmap

- **Générer « Diagram Editor Gen Model » (Fichier : *.gmfgen) :**
  - Cliquer « Transform » :
  - Nous obtenons le résultat : Dans « model » et dans , l'affichage attendu doit être le suivant :

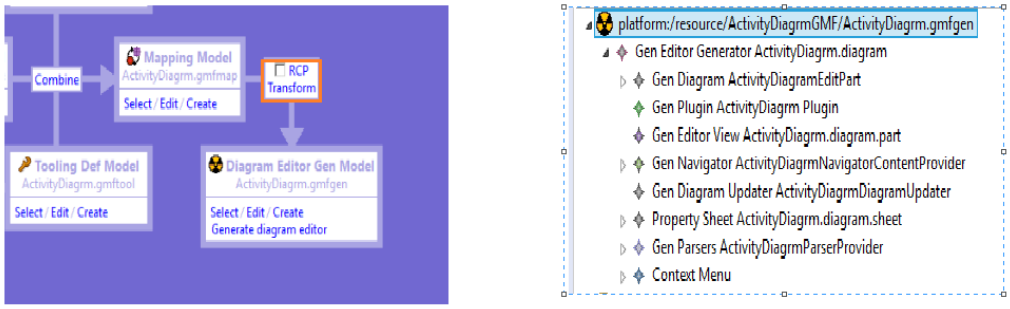


Figure 4.32: Fichier ActivityDiagram.gmfgen

- **Générer l'éditeur de diagramme :**
  - Cliquer « Generate diagram editor » :
  - Le projet «ActivityDiagram..diagram » est créé.

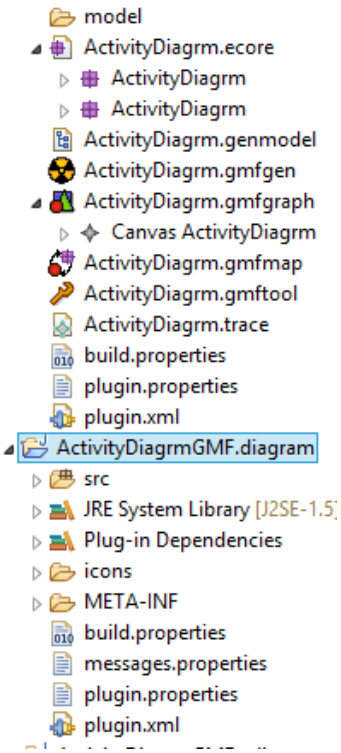


Figure 4.33: Création projet ActivityDiagram.diagram

4.5.4 Test de l'éditeur

- **Créer une configuration d'exécution :**
  - Sélectionner le projet « **ActivityDiagrmGMF.diagram** » et afficher la configuration d'exécution
  - Créer la nouvelle configuration d'exécution de type : « Eclipse Application » en double-cliquant « Eclipse Application ».
  - Nommer la configuration, cliquer sur le bouton « Apply » et exécuter la configuration d'exécution en cliquant sur le bouton « Run » :

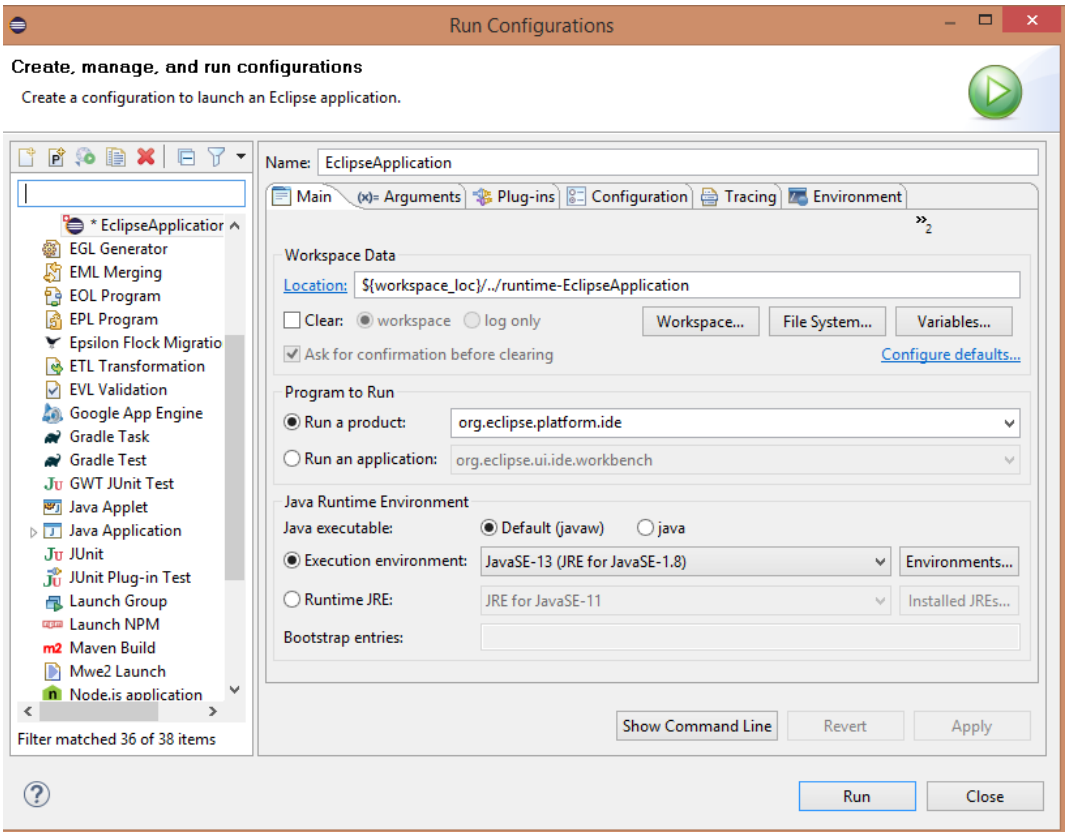


Figure 4.34: Configuration l'exécution

– Une nouvelle plateforme « Eclipse » est lancée avec pour « workspace » : « ...\runtime-GenerationEditeurGraphique ».
– Fermer ensuite la fenêtre « Welcome » et quelques vues afin d'obtenir le résultat suivant :

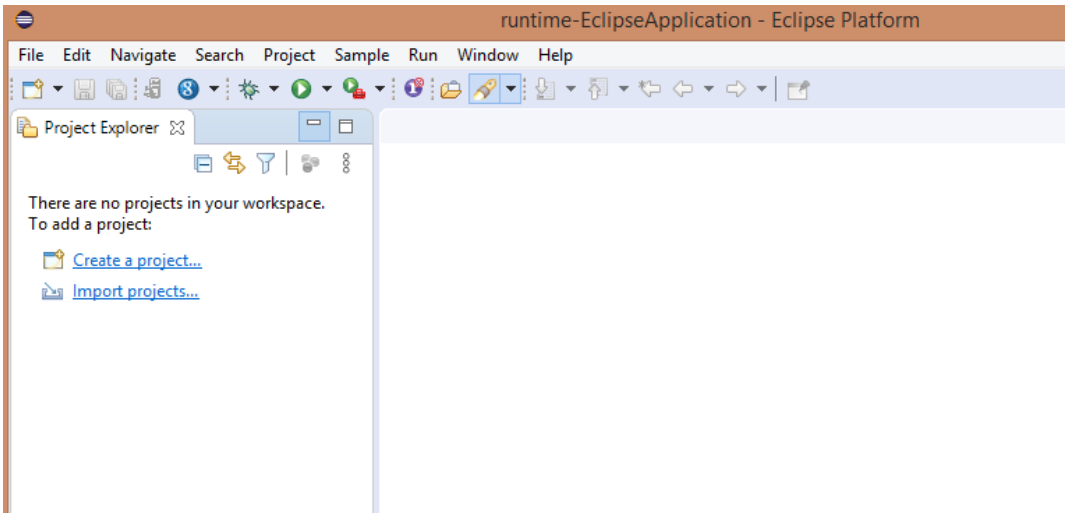


Figure 4.35: L'interface de Eclipse

- **Créer un projet :**
  - Créer un nouveau projet via le menu « File > New > Project... » :
  - Sélectionner l'élément « Project » :
  - Cliquer sur le bouton « Next », puis nommer le projet «ActivityDiagram» :
  - Cliquer sur le bouton « Finish »

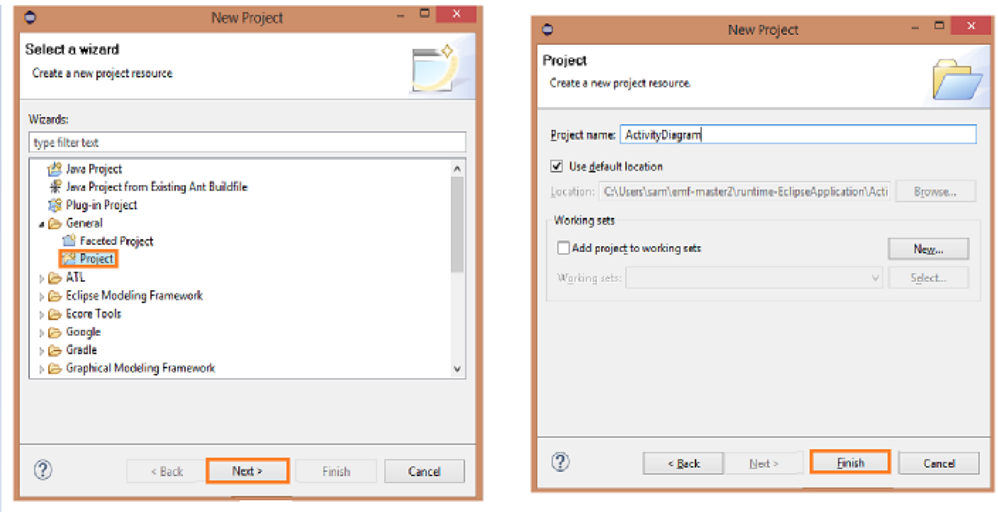


Figure 4.36: création du projet

- **Éditer un graphe :**
  - Ouvrir l'éditeur de construction de diagrammes « Graphe Diagram » via le menu « New > Other... » :
  - Cliquer sur le bouton « Next > », nommer le diagramme «ActivityDiagram» puis
  - Cliquer sur le bouton « Next > » :
  - Cliquer sur le bouton « Finish », éditer un graphe puis faire une sauvegarde

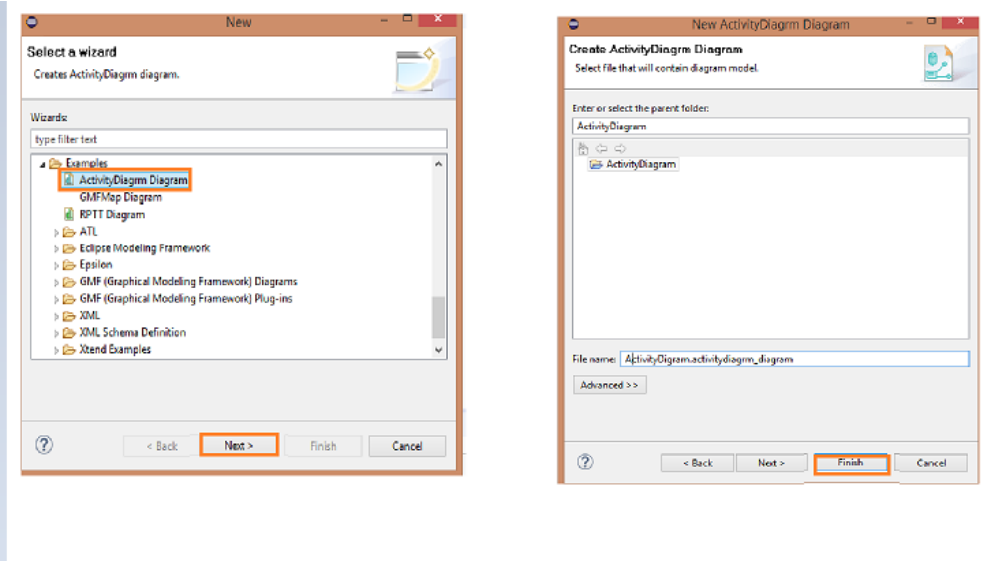


Figure 4.37: Éditeur graphe

- De la même manière sélectionner les élément de RPTT

4.5.5 Cas d'etude

- **Exemple1 : le processus de réalisation d'un mémoire de fin d'étude :**

Le principe de ce système est : un étudiant admis en master 2 choisis un sujet de mémoire, il consulte son encadreur en lui monteront sa documentation sur le sujet, un plan de travail sera proposé par l'étudiant¸ si le plan proposé n'est pas validé par l'encadreur, l'étudiant doit présenter un autre plan de travail. Sinon l'étudiant passe à la rédaction du mémoire et la réalisation de l'application en parallèle avant le dépôt de mémoire l'étudiant doit procéder à la vérification est cela avant le délai de dépôt. Le mémoire déposé sera examiné et présenté par la suite. Dans la Figure (4.38), nous présentons la modélisation du comportement de ce processus avec le diagramme d'activité

qui est notre modèle de base

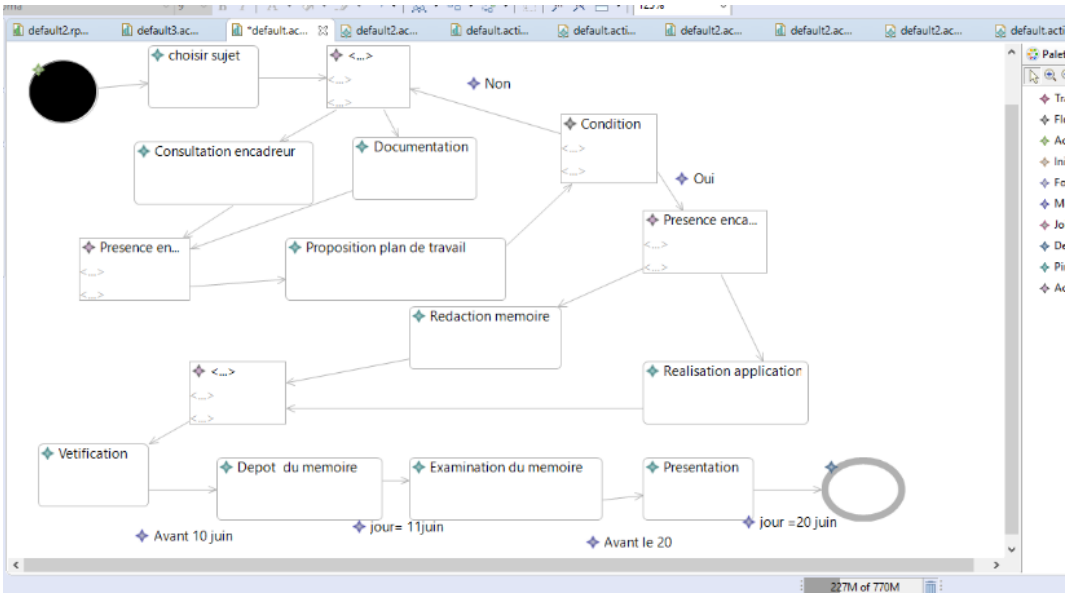


Figure 4.38: Modèle source de processus de réalisation d'un mémoire de fin d'étude

Le modèle cible présenté dans la figure suivant.

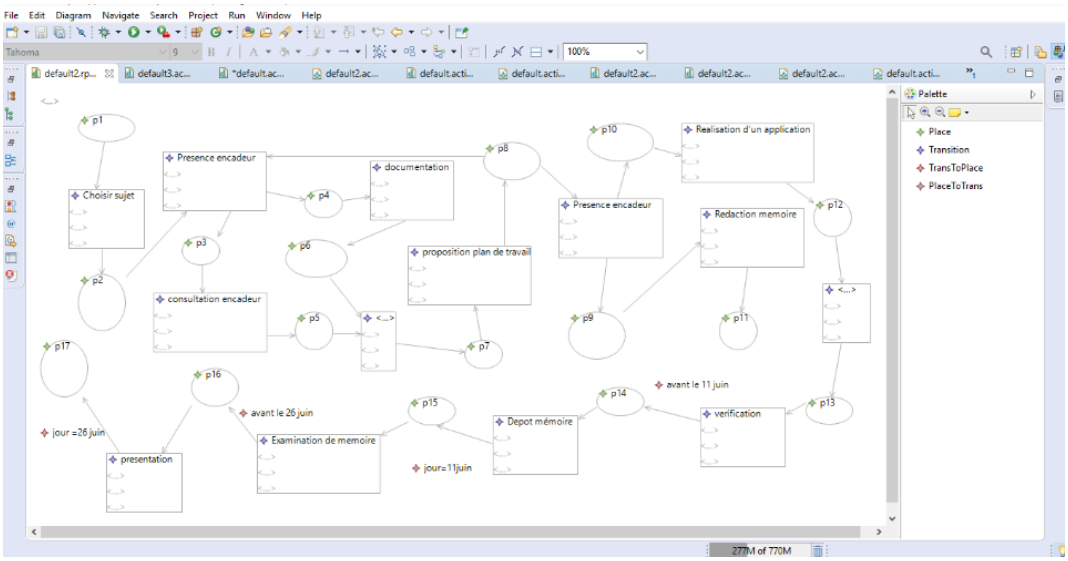


Figure 4.39: Modèle cible de processus de réalisation d'un mémoire de fin d'étude

4.5.6 Comparaison entre la transformation par ATOM3 et Eclipse Modeling Framework

Ce tableau illustre les différences entre les critères de ATOM3 et EMF.

Critère	**ATOM3**	**EMF**
Type de transformation	Endogène	Exogène
Approche utilisé	Grammaire degraphe	SPO
Intégrité de contrainte	Non supporté présenté par OCL ou Python	Non supporté présenté par NAC
Éditeur	Graphique	Graphique
Degré d'automatisation	Grammaire de graphe	Simulation automatique
Processus de transformation	Automatique	Automatique
Réutilisabilité		Transformation configurable
Vérification	Priorité de règles	Contrôles de fin de contrat et l'unicité des résultats de la transformation
Validation	Typage correct du modèle cible	Confluence, terminaison, analyse des paires critiques de règles
Composition	Priorité de règle	Niveaux de priorité procédural
Complexité	Des couches avec priorités, le séquençage par priorité, l'exécution en parallèle des matches non chevauchés	Respecter les contraintes induites par la sémantique d'agrégation
Standardisation	XMI	ECORE
Type d'utilisation	Transformation de modèle	Transformation de modèle
Typage	Requis	Requis
Contrôle	Priorité	Priorité
Exploration	Linéaire	Linéaire

Table 4.1: Les critère de ATOM3 et GMF

4.5.7 Les avantages de modélisation avec EMF

- La popularité d'EMF a donné naissance à de nombreuses initiatives réutilisant EMF au sein de la fondation Eclipse dans le projet Eclipse Modeling regroupant les projets dédiés au MDA.

- Le projet Eclipse Modeling héberge une implémentation du standard UML reposant sur EMF. Cette implémentation contient un méta-modèle du standard UML avec des éditeurs permettant de définir des diagrammes de classes, d'activité, de cas d'utilisation etc.
- Parmi les projets contenus dans Eclipse Modeling, on trouve le projet dédié à la transformation de modèles vers modèles.
- Projet Eclipse Modeling contient une partie dédiée aux projets de génération de code à partir de modèle EMF. Dans cette catégorie, on peut trouver le projet implémentant le langage standard de transformation de modèle vers texte de l'OMG (MOFM2T). Contrairement à la génération de code vu précédemment, permet de générer du code pour n'importe quelle cible technologique (java, scala, ruby, etc). La compatibilité avec les modèles définit par EMF permet ainsi de générer n'importe quel type de code depuis n'importe quel méta-modèle (UML, SysML, un méta-modèle dédié à un domaine .
- Le projet Eclipse Modeling contient aussi des projets dédiés à la manipulation graphique de modèle EMF tel que Graphical Modeling Framework (GMF) et Graphiti.
- L'objectif général d'EMF est de proposer un outillage qui permet de passer du modèle au code Java automatiquement.

4.6 Conclusion

Nous avons présenté dans ce chapitre les différentes étapes d'un processus de transformation M2M. Cette approche est basée sur la méta-modélisation (les méta-modèles). Nous avons proposé deux méta-modèles (le diagramme d'activité, réseau de pétri TT), Ensuite nous avons présenté un ensemble de règles permettant la réalisation de la transformation. pour illustrer notre approche de transformation nous avant créer deux model et fait la présentation graphique a travers gmf (graphique modeling framework)

Conclusion générale

Dans ce mémoire, on a essayé de présenter le concept de transformation de modèles dans le domaine de l'ingénierie dirigée par les modèles, par une proposition d'une grammaire de graphe qui permet de transformer un modèle d'un formalisme source graphique à un formalisme cible Graphique aussi , Il se base essentiellement sur l'utilisation combinée de méta-modélisation.

Le résultat de notre travail est une approche automatique pour transformer les diagrammes d'activité d'UML2.0 vers RPTT « Réseaux de pétri temporellement temporisé ».

Le but de notre travail est de réaliser une transformation des diagrammes d'Activité vers les RPTT en respectant les concepts MDA et donc les spécifications définies par l'OMG. Définir le concept de méta-modélisation appliqué au problème de transformation DA2RPTT. La deuxième partie nous avons utilisé GMF pour mettre en avant le mapping des concepts pour passer d'un modèle à l'autre.

Le langage utilisé pour implémenter les règles de transformation (ATL) s'est avéré être un langage rapidement lourd et complexe. L'élaboration de règles de base est triviale si les deux méta-modèles sont proche (une méta-classe source correspond à une ou deux méta-classe cible) mais devient lourd et complexe pour élaborer des transformations complexes.

ATL est apparu avec nos connaissances actuelles du langage peu efficace vis à vis des liens d'héritages entre méta-classe et donc de l'héritage de règles.

L'EMF présente un Framework complet et extensible pour le développement MDA. L'objectif de notre travail est de suivre cette démarche pour résoudre certains problèmes rencontrés lorsqu'on utilise le langage de modélisation le plus populaire (UML).

Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités association. On peut donc transformer automatiquement des diagrammes d'activités en RPTT.

Finalement, et afin d'arriver à développer une approche totalement automatique, incluant tous les diagrammes UML, nous proposerons de continuer la transformation des autres diagrammes UML (diagramme de séquence, diagramme de cas d'utilisation, diagramme d'état/transition, etc . . .) vers les RPTT.

Bibliographie

[1] A. Zahoui. “developpement d’une chaine d’outils en fonction du nouveau standard fondationnel uml (fuml)”. In : l’Universitéd’Annaba.

[2] J.Bézivin. “In Search of a Basic Principle for Model Driven Engineering”. In : *The European Journal for the Informatics Professional* 2 (2004), p. 21-24.

[3] F. FOURATI. “Une approche IDM de transformation exogène de Wright vers Ada”. In : L’École Nationale d’Ingénieurs de Sfaxtravaux.

[4] AMOUR Nasreddine. *Une approche MDA pour la transformation des diagrammes de classe UML vers une base de données Basant sur l’outil EMF.*

[5] J. Bézivin. “Sur les principes de base de l’ingénierie des modèles”. In : *RSTI-L’Objet* 10.4 (2004), p. 145-157.

[6] Benoît Combemale. “Ingénierie Dirigée par les Modèles (IDM) – État de l’art”. In : (29 Mar 2009). url : https://hal.archives-ouvertes.fr/hal-00371565.

[7] JeffRothenberg. *the Nature of Modeling.* 1989.

[8] Kh. Chemani F. Berabah. *Une Approche De Transformation Des Diagrammes de séquence UML Vers Les Réseaux De Pétri Temporellement Temporisés, Université Khemis Meliana.*

[9] Saidi Baya Bouanane Razika. *Une Approche De Transformation Des Diagrammes D’activités UML Vers Les Réseaux De Pétri Temporellement Temporisés.*

[10] M. SELLAM. *diplôme de Magister « Vérification et validation de transformation de modèles ».* 2015.

[11] S. Cook. “Domain-Specific Modeling and Model Driven Architecture”. In : *MDA Journal* (Jan 2004), p. 1-10.

[12] J. M. Favre. *Towards a Basic Theory to Model Driven Engineering.* 2004.

[13] X. Dolques. *Génération de Transformations de Modèles : une approche basée sur les treillis de Galois.*

[14] S. Czarnecki K. et Helsen. *Featurebased survey of model transformation approaches.* T. 45. 03. IBM SYSTEMS JOURNAL.

[15] H.HAMROUCHE. *Une Approche de transformation des diagrammes D’activité d’UML vers CSP basée sur la transformation de graphes.*

[16] Touil Amara. *Spécification d’une chaine de transformation.*

[17] M. AOUAG. *Des diagrammes UML 2.0 vers les diagrammes orientés aspect à l’aide de transformation de graphes.*

[18] A. Elmounadi. *Architecture dirigée par les modèles : Méthodes et outils de transformation de Modèles.* 2013.

[19] url : https://www.eclipse.org/viatra/. (22.03.2020).

[20] URL : https://wiki.eclipse.org/UMLX/. (22.03.2020).

[21] URL : http://atom3.cs.mcgill.ca/. (22.03.2020).

[22] K. Czarnecki et S. Helsen. *Classification des approches detransformation de modèles, Université of Waterloo.*

[23] I. Ragoubi. *Motifs de transformation de méta modèle.*

[24] D. Foures. "Transformation des diagrammes d'activités SysML1.2 vers les réseaux de Petri dans un cadre MDE". In : Université Paul Sabatier - Laboratoire d'Analyse et d'Architecture des Systèmes, 2011.

[25] B. Coulette S. Diaw R. Lbath. "Etat de l'art sur le développement logiciel dirigé par les modèles". In : Université de Toulouse, 2008.

[26] Jean Bézivin Xavier Blanc. "MDA : VERS UN IMPORTANT CHANGEMENT DE PARADIGME EN GENIE LOGICIEL". In : Université de Nantes.

[27] Jamal Abd-Ali. *métamodélisation et transformation automatique de psmDans une approche mda.* 2006.

[28] Sylvain ANDRE. "MDA (Model Driven Architecture) principes et états de l'art". In : Centre d'enseignement de lyon, France, 2004.

[29] GAOUAR Lamia. "Approche MDA pour la construction d'interfaces homme-machine : cas de la multimodalité et de la plasticité". In : universite abou-bekr belkaid – tlemcen, 2019.

[30] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation.* T. 1. World Scientific, 1999.

[31] A. Agrawal G. Karsai. *Graph Transformations in OMG's Model-Driven Architecture.* T. 3062. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, juillet 2004, p. 243-259.

[32] M. Andries G. Engels A. Habel B. Hoffmann h.-J. Kreowski s. Kuske D. Pump A. Schürr et G. Taentzer. *Graph transformation for specification and programming.* T. 34. 1. Science of Computer programming, Avril 1999, p. 1-54.

[33] RAIDA ELMANSOURI. "Thèse de Doctorat - Modélisation et Vérification des processus métiers dansles entreprises virtuelles : Une approche basée sur la transformation de graphes algerie". In : Université Mentouri Constantine.

[34] El-hillali Kerkouche. *Modélisation Multi-Paradigme :Une Approche Basée sur la Transformation de Graphes.*

[35] Mouna Bouarioua. "Une approche basée transformation de graphes pour la génération de modèles de réseaux de Petri analysables à partir de diagrammes UML". In : UNIVERSITÉ CONSTANTINE 2, 2013.

[36] *AToM3 (2006).* URL : http://atom3.cs.mcgill.ca/.

[37] A. Maria. "Introduction to modelling and simulation". In : 1997, p. 7-13.

[38] D. Hohan et P. Stooper. *Fundamentals of Softuiare Design and Verification*. 1994.

[39] J.M.Wing. *A Specifier's Introduction to Formal Methods*. T. 23. 9. Computer, 1990, p. 8-23.

[40] KAZI AOUEL w. et l. BELBACHIR. "mémoire l'obtention de la licence « gestion de la prise d'un rendez-vous »". In : université aboubakrbelkaid– Tlemcen, 2014.

[41] L. AUDIBERT. *UML 2*. 2007-2008.

[42] F. GUERROUF. "Une Approche de Transformation des Diagrammes d'Activités d'UML Mobile 2.0 vers les Réseaux de Pétri". In : Université El Hadj Lakhdar – BATNA(Algérie).

[43] *Diagramme d'activité*. URL : http://remy-manu.no-ip.biz/UML/Cours/coursUML11.pdf.

[44] G. MOTET J. Vida H. MALGOUYRES. "REGLES DE COHE´RENCE UML 2.0 (Version 1.1)". In : Institut National des Sciences Appliquées de Toulouse(France).

[45] A.ZITOUNI. "utilisation des design patterns et des méthodes formelles dans le développement des systèmes d'information », thèse doctorat". In : université Constantine(Algérie), 2008.

[46] URL : http://dictionnaire.sensagent.leparisien.fr/m%c3%a9thode%20formelle%20(informatique)/fr-fr/. (15.04.2020).

[47] H. DIAB. "« évaluation de méthodes formelles de spécification », mémoire du grade de maîtresses sciences". In : université de Sherbrooke (canada), mai 1999.

[48] I.BENBOUDINA. "« spécification et vérification d'un système d'ascenseur par la méthode b », mémoire du diplôme de master". In : université Mohamed Boudiaf (Algérie), 2018.

[49] Charles Antony Richard Hoare. *An axiomatic basis for computer programming communication of the ACM*. T. 12. 10. Computer, 1969, p. 576-580.

[50] BERRAMLA KARIMA. "vérification et validation des transformations des modèles". In : université d'Oran, 2014.

[51] HETTAB ABDELKAMEL. "Mémoire de Magistère « De M-UML vers les réseaux de Pétri « Nested Nets » : Une approche basée transformation de graphes". In : Université Mentouri Constantine, 2009.

[52] Elvinia Riccobene Angelo Gargantini et Patrizia Scandurra. *Integrating Formal Methods with Model-driven Engineering*. In Proceedings of the Fourth International Conference on Software Engineering Advances, 2009. ISBN : 978-0-7695-3777-1.

[53] URL : https://moves.rwth-aachen.de/teaching/ss-15/introduction-to-model-checking/. (30.04.2020).

[54] Monin J. F. *Introduction aux méthodes formelles*. Hermès, préface de G. Huet, 2000. ISBN : 2-7462-0140-2.

[55] A. P. Mathur. *Fondations of Software Testing*. April 17, 2008. ISBN : 8131716600.

[56] Redouane KARA. "THESE DE DOCTORAT « Contribution à l'Analyse Quantitative et à la Commande des Réseaux de Pétri Continus à Arcs Valués. Application aux Systèmes de Production Manufacturière »". In : Université Mouloud Mammeri de Tizi Ouzou, 2009.

[57] DRISSI Meriem. “l’Obtention du Diplôme de Master « Génération des réseaux de Pétri à partir Des modèles OSSAD »”. In : UNIVERSITE ABDELHAMID IBN BADIS MOSTAGANEM, 2016.

[58] R. BAHRI. “Une approche intégrée Mobile-UML/Réseaux de Pétri pour l’Analyse des systèmes distribués à base d’agents mobiles”. In : Université Mentor Constantine.

[59] T. Murata. “Pétri nets : Properties, analysis and applications”. In : *Proceedings of the IEEE* 77 (April 1989), p. 541-580.

[60] Scorletti Gérard Binet G. “Réseaux de Pétri”. In : université de France, Juin 2006.

[61] RAMCHANDANI C. *« Analysis of asynchronous concurrent systems by timed Petri nets », PhD thesis*. 1974.

[62] MERLIN P. M. *« A study of the recoverability of computing systems », PhD thesis*. 1974.

[63] Franck Cassez et Olivier H. Roux. *Traduction structurelle des Réseaux de Petri Temporels vers les Automates Temporisés*. 26 février 2003.

[64] Asma Chachoua et Radja Boukharrou. *Mise en œuvre distribuée de la sémantique opérationnelle des réseaux de Pétri temporellement temporisés : Approche Orientée Objet.*