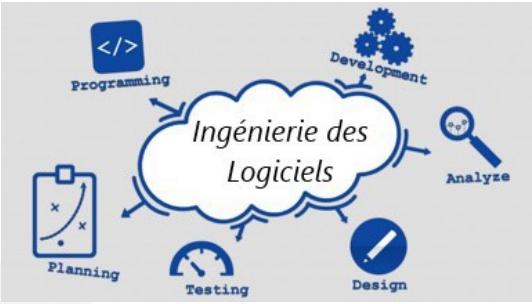
Ingénierie des Logiciels



Dr. Said MEGHZILI

Centre Universitaire de Mila

Département de Mathématique et Informatique

Email: s.meghzili@univ-mila.dz

1.0

Février 2022

Table des matières

Objectifs

A l'issu de ce module, l'étudiant sera capable :

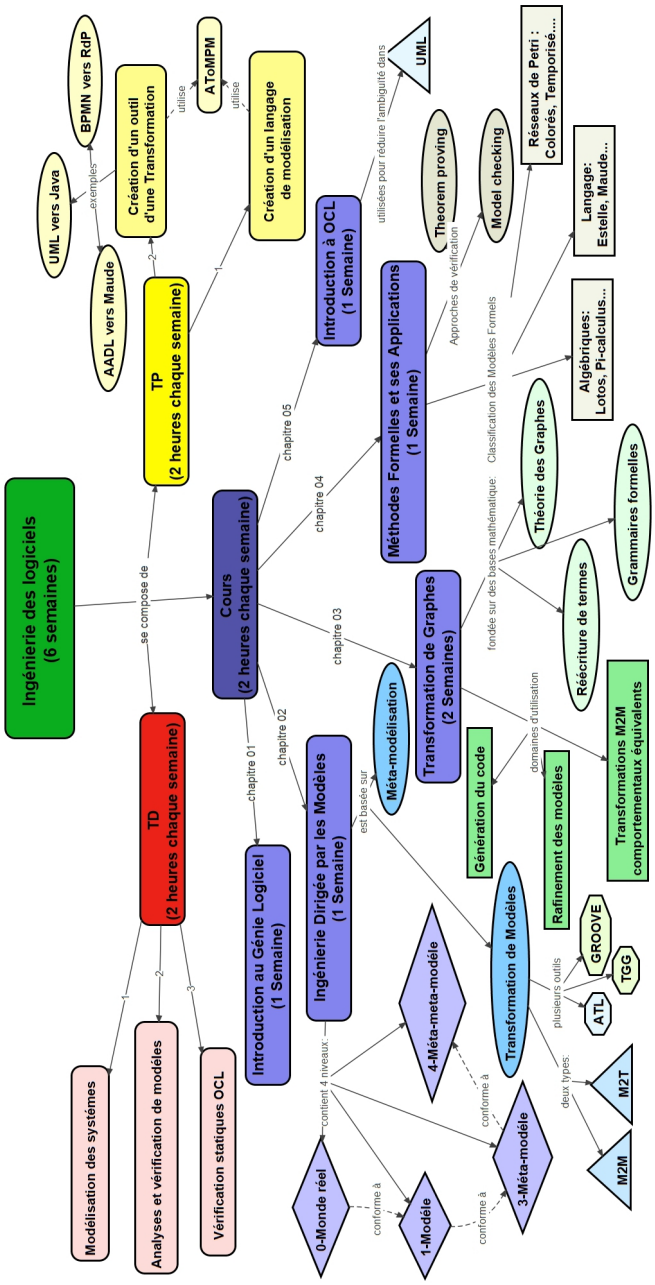
- En termes de savoir
  - Connaître les cycles du développement de logiciels.
  - Comprendre les principes de l'approche MDA pour le développement d'applications.
  - Maîtriser la conception formelle des systèmes complexes.
- En termes de savoir-faire
  - Réaliser des transformations de modèles basées sur la transformation de graphes.
  - Valider et vérifier formellement des systèmes complexes.
- En termes de savoir-être
  - Combiner des méthodes formelles avec semi-formelles pour le développement de logiciels sûr.

Pré-requis

Il est recommandé aux étudiants de connaître :

- Notions sur le génie logiciel
- Concepts de base de la théorie des langages

I Carte Conceptuelle



.

II Introduction

Au cours des dernières années, la transformation numérique de l'état peut être considérée comme une véritable révolution dont l'impact est direct sur le développement de logiciels en général et la sécurité des systèmes en particulier. Le processus de développement d'un logiciel contient un ensemble d'activités successives, organisées et suit un modèle de cycle de vie. Cependant, le processus de développement d'un logiciel est une tâche très difficile qui nécessite la prise en compte de plusieurs paramètres à savoir les attentes des clients, la fiabilité ainsi que le respect des délais et des coûts de construction. En plus, la tâche la plus complexe est l'évolution d'un logiciel en cas de changement de l'infrastructure d'exécution ou s'il y a de nouvelles exigences, en particulier les gros logiciels.

L'ingénierie dirigée par les modèles (IDM) est une approche qui offre au développeur des concepts, des outils et des techniques afin de diminuer la complexité du développement d'un logiciel. Cependant, l'IDM souffre d'un manque de techniques d'analyse et d'outils de vérification, ce qui rend cette approche insuffisante toute seule. Dans ce cas, l'intégration de l'IDM avec les méthodes formelles apparaît comme une solution prometteuse qui permet de décrire sans ambiguïté le comportement de systèmes. En plus, les méthodes formelles permettent de vérifier certaines propriétés qualitatives que doivent avoir les systèmes tels que l'absence de l'interblocage et de la famine.

Organisation de cours :

Dans le premier chapitre, nous introduisons les qualités attendues d'un logiciel ainsi que les principes d'ingénierie du logiciel. Ensuite, Nous abordons les composants du cycle de vie d'un logiciel. Enfin, Nous présentons les modèles de cycle de vie d'un logiciel.

Au deuxième chapitre, nous présentons les principes clés de l'ingénierie dirigée par les modèles (IDM) tel que les notions du modèle, méta-modelé et transformation de modèles. Ensuite, Nous abordons l'approche MDA (Architecture Dirigée par les Modèles). Enfin, nous présentons le domaine de la modélisation multi-paradigmes.

Le troisième chapitre est consacré à la présentation des concepts de base de transformations de graphes. Nous abordons les concepts suivants : graphe, grammaire de graphes et système de transformation de graphes. Enfin, nous présentons une transformation des modèles BPMN vers les Réseaux de Petri à l'aide de l'outil AToMPM. Cette approche est basée sur la Méta-modélisation et les Grammaires de Graphes.

Dans Le quatrième chapitre, nous introduisons des définitions générales des méthodes formelles ainsi que leur classification. Nous présentons les concepts de base des réseaux de Petri. Ensuite, nous abordons l'intégration des méthodes formelles avec l'IDM afin d'augmenter la fiabilité et de garantir l'absence d'erreurs de conception. Enfin, nous présentons un aperçu sur le sujet de la vérification formelle des transformations de modèles en utilisant le Model checker et les démonstrateurs de théorèmes.

Enfin, le dernier chapitre est consacré à la présentation du langage OCL qui permet de préciser les diagrammes UML. Nous abordons les concepts du langage OCL et nous montrons son efficacité à travers plusieurs exemples.

III Chapitre 01 : Introduction au génie logiciel

1. Introduction

Attention

- Les Défaillances des Systèmes informatiques à peu prés: 80 % de logiciel et 20 % de matériel.
    - Les problèmes liés à l'informatique sont essentiellement des problèmes de Logiciel.
- La « Crise du logiciel » : Étude sur 8 380 projets[2]* (Standish Group, 1995)
    - ***Succès*** : 16 %
    - ***Problématique*** : 53 %
    - ***Echec*** : 31 % (abandonné)
- Introduction de l'expression: « ***Génie Logiciel*** » (Software Engineering)
    - Comment faire des logiciels de qualité ?
    - Quels sont les critères de qualité pour un logiciel ?

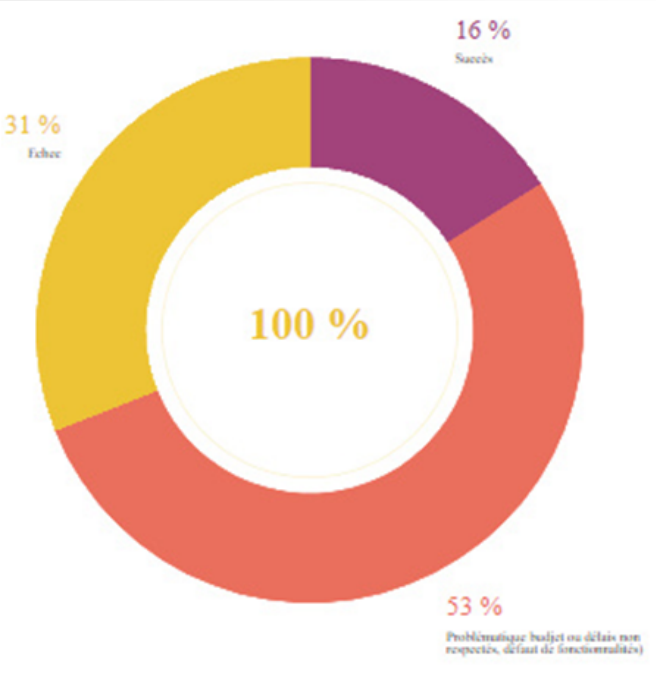


Figure 1.1 : Étude sur des projets informatique

2. Qualités attendues d'un logiciel

Fondamental

Il existe plusieurs qualités attendues d'un logiciel[2]*:

- Fiabilité: le logiciel fonctionne raisonnablement en toutes circonstances, rien de catastrophique ne peut survenir, même en dehors des conditions d'utilisation prévues.
- Interopérabilité: un logiciel doit pouvoir interagir avec d'autres logiciels
- Performance: un logiciel doit satisfaire aux contraintes de temps d'exécution
- Portabilité: Un même logiciel doit pouvoir fonctionner sur plusieurs machines
- Réutilisablité: 80% du code on retrouve partout 20% du code est spécifique
- Utilisabilité: Facilité d'apprentissage et Facilité d'utilisation
- Facilité de maintenance
- Maintenance corrective: Corriger les erreurs.
- Maintenance adaptative: Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu du l'évolution des Environnements d'exécution.
- Maintenance d'extension

3. Principes d'ingénierie pour le logiciel

Fondamental

Un certain nombre de grands principes se retrouvent dans toutes ces méthodes:

- Généralisation: regroupement d'un ensemble de fonctionnalités semblables en une fonctionnalité paramétrable (héritage...)
- Structuration : façon de décomposer un logiciel
- Abstraction :mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et en omettant ceux qui ne le sont pas
- Modularité :décomposition d'un logiciel en composants discrets
- Documentation : gestion des documents incluant leur identification, acquisition, production, stockage et distribution
- Vérification : détermination du respect des spécifications établies sur la base des besoins identifiés dans la phase précédente du cycle de vie

4. Composantes du cycle de vie d'un logiciel

Méthode

Ensemble d'activités successives, organisées en vue de la production d'un logiciel[2]*:

1. Analyse de besoins
    - Comprendre les besoins du client.
    - L'environnement et le contexte du futur système.
    - Ressources disponibles.

- Contraintes de performance.

2. Spécification
    - Établir une description claire de ce que doit faire le logiciel.
    - Clarifier le cahier des charges.
3. Conception
    - Conception générale
        - Conception architecturale : déterminer la structure du système.
        - Conception des interfaces : déterminer la façon dont les différentes parties du système agissent entre elles.
    - Conception détaillée : déterminer les algorithmes pour les différentes parties du système
4. Implémentation (codage)
    - Écrire le code du logiciel.
5. Tests
    - Validation
        - Assurer que les besoins du client sont satisfaits.
        - Concevoir le bon logiciel.
    - Vérification
        - Assurer que le logiciel satisfait sa spécification.
        - Concevoir le logiciel correctement.
6. Livraison
   Fournir au client une solution logicielle qui fonctionne correctement :
    - Installation : rendre le logiciel opérationnel chez le client.
    - Formation : apprendre aux utilisateurs à utiliser le logiciel.
    - Assistance : répondre aux questions des utilisateurs.
7. Maintenance
    - Mettre à jour et améliorer le logiciel pour assurer sa pérénité.
    - Pour limiter le temps et les coûts de maintenance, des efforts doivent être faits sur les étapes précédentes.

5. Modèles de cycle de vie d'un logiciel

5.1. Processus de Développement Séquentiels

Modèle en cascade

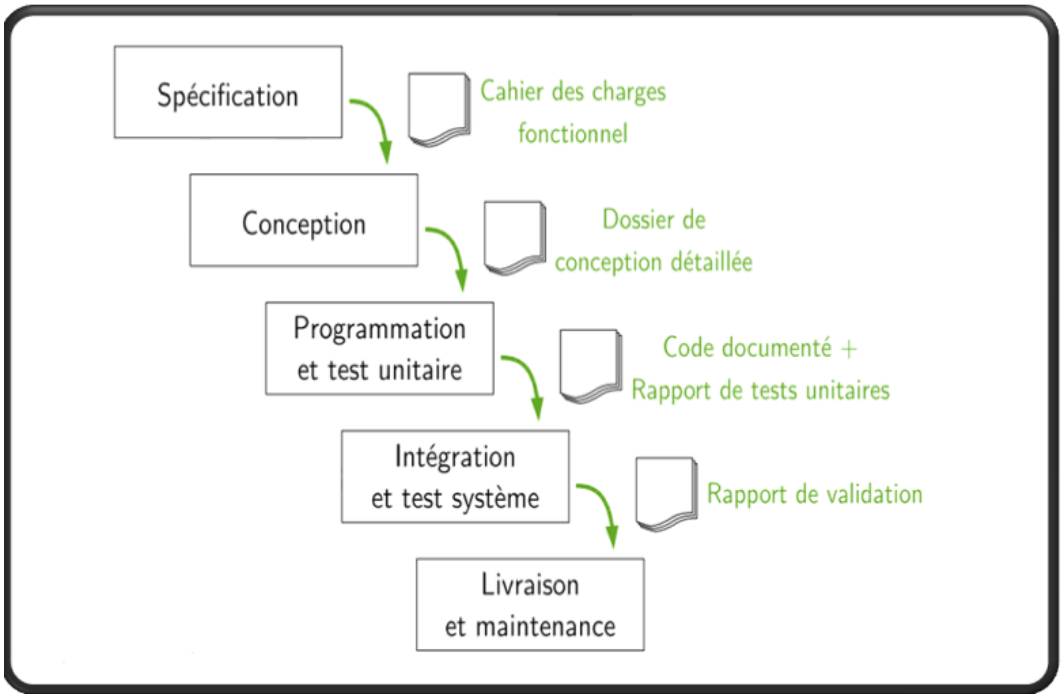


Figure 1.2 : Modèle en cascade

Caractéristiques :

- Hérite des méthodes classiques d'ingénierie.
- La découverte d'une erreur entraîne retour à la phase à l'origine de l'erreur et nouvelle cascade, avec de nouveaux documents..
- Coût de modification d'une erreur important.
- Mais pas toujours adapté à une production logicielle, en particulier si les besoins du client changeants ou difficiles à spécifier.

Modèle en V

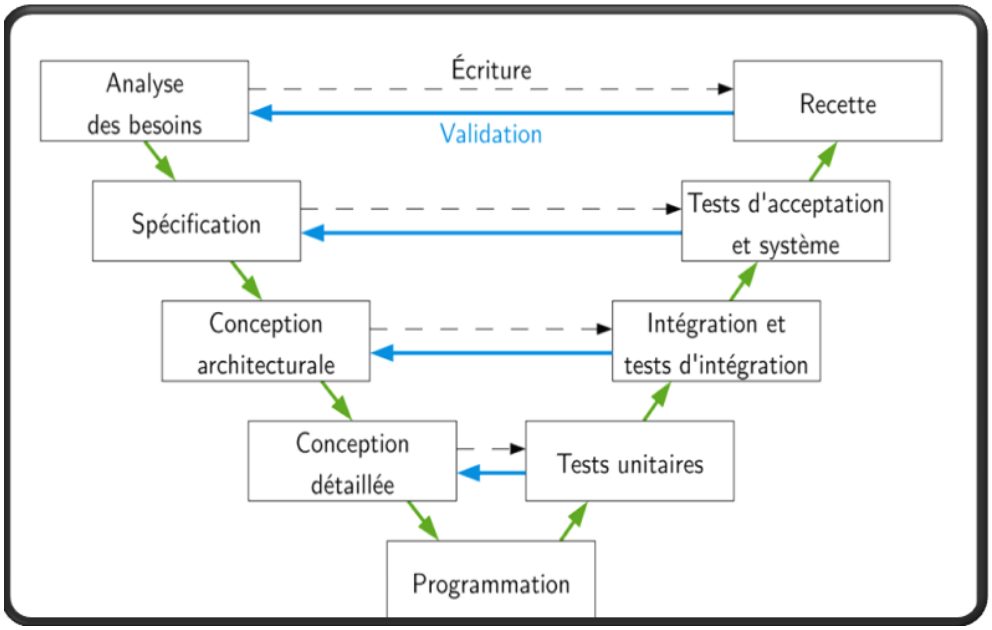


Figure 1.3 : Modèle en V

Caractéristiques:

- Variante du modèle en cascade.
- Mise en évidence de la complémentarité des phases menant à la réalisation et des phases de test permettant de les valider.

Inconvénients :

- Les mêmes que le cycle en cascade
- Processus lourd, difficile de revenir en arrière
- Nécessite des spécifications précises et stables
- Beaucoup de documentation.

5.2. Processus de Développement Itératifs

Modèle non linéaire

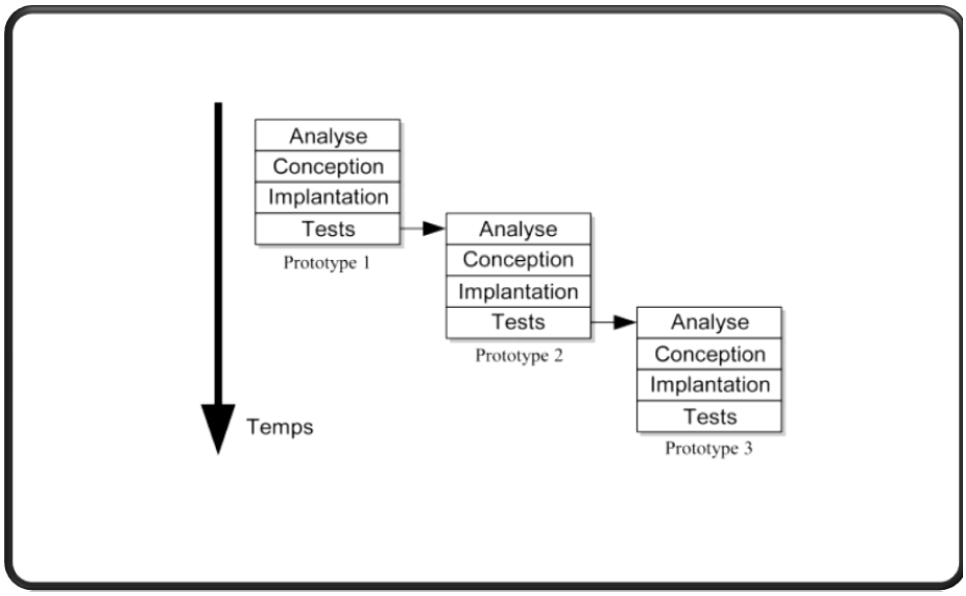


Figure 1.4 : Processus de développement itératifs

Avantages :

- Capture des besoins continue et évolutive.
- Détection précoce des erreurs.
- État d'avancement connecté à la réalité.
- Implication des clients/utilisateurs.

Inconvénients :

- Explosion des besoins.
- Difficile de définir la dimension d'un incrément.
- Nécessite une direction rigoureuse.

5.3. Processus Unifié

Caractéristiques

- Méthode itérative et incrémentale.
- Piloté par les cas d'utilisation.
- Centré sur l'architecture : Modélisation orientée objets, utilise la modélisation visuelle (UML) et Fondé sur la production et l'utilisation de composants.

Méthode : Les phases de développement d'un processus unifié

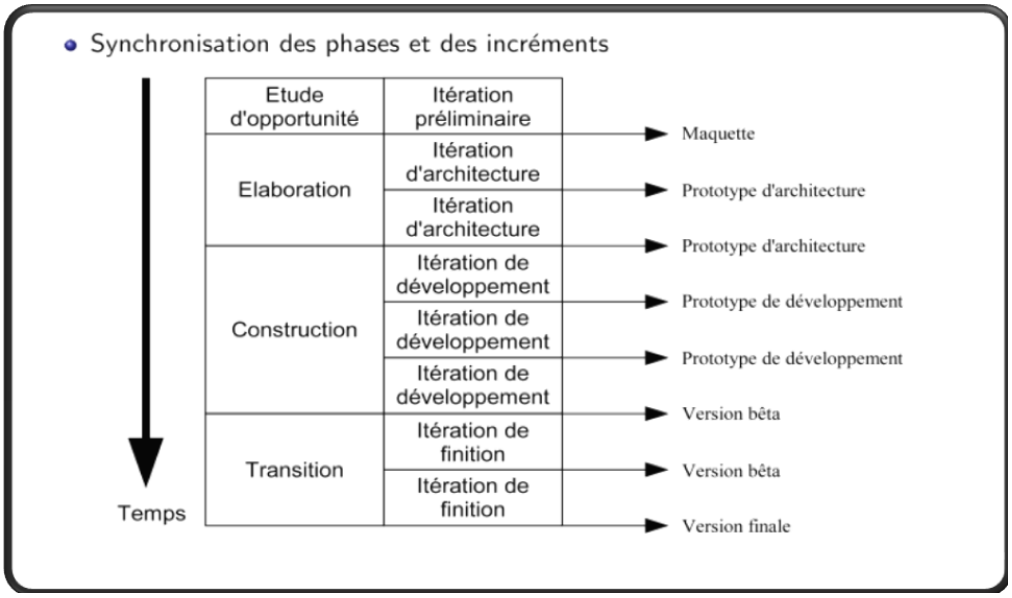


Figure 1.5 : Processus unifié

- Étude d'opportunité (préparation)
    - Étude de marché, estimation du coût.
    - Capture des besoins majeurs et analyse préliminaire.
- Élaboration
    - Capture et analyse des besoins.
    - Choix de l'architecture.
- Construction
    - Répartition des tâches sur plusieurs équipes.
    - Rédaction de la documentation finale.
- Transition
    - Fabrication, livraison.
    - Installation, formation.
    - Support technique.

5.4. Méthodes Agiles

Définition

Une méthode agile est une méthode de développement informatique permettant de concevoir des logiciels en impliquant au maximum le demandeur (client) :

- Plus grande réactivité à ses demandes.
- Plus pragmatiques que les méthodes traditionnelles.
- Recherche de la satisfaction réelle du besoin du client.

Exemple : Méthodes agiles

- XP (EXtreme Programming),
- DSDM (Dynamic Software Development Method),
- ASD (Adaptative Software Development),
- CCM (Crystal Clear Methodologies),
- SCRUM,
- FDD (Feature Driven development)

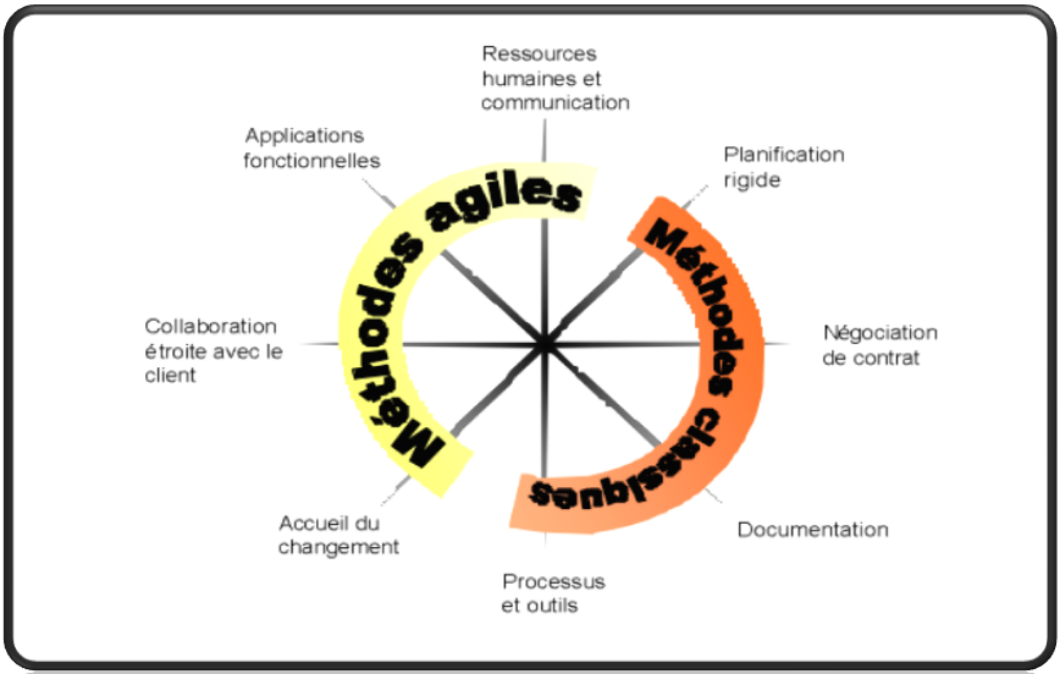


Figure 1.6 : Méthodes agiles

Caractéristiques :

- itératives à planification souple.
- Itérations très courtes.
- S'opposent à la procédure et à la spécification à outrance.

Les priorités d'une méthode agile :

1. Priorité aux personnes et aux interactions sur les procédures et les outils.
2. Priorité aux applications fonctionnelles sur une documentation pléthorique.
3. Priorité de la collaboration avec le client sur la négociation de contrat.
4. Priorité de l'acceptation du changement sur la planification.

Avantages/ Inconvénients :

- Développement rapide en adéquation avec les besoins.
- Mais pas de spécification, documentation = tests et maintenance.

6. Conclusion

Dans ce chapitre, nous avons présenté la notion générale de génie logiciel. Dans un premier temps, nous avons introduit les qualités attendues d'un logiciel. Ensuite, nous avons présenté les principes d'ingénierie du logiciel. Par la suite, nous avons abordé les composants du cycle de vie d'un logiciel. Enfin, nous avons présenté les modèles de cycle de vie d'un logiciel.

IV Chapitre 02 : Ingénierie Dirigée par les Modèles

1. Introduction

Exemple : Évolution des technologies

- Les technologies logicielles sont en Évolution permanente.
- Exemple dans les systèmes distribués.
    - Évolution pour faire communiquer et interagir des éléments distants: 1-C et sockets TCP/UDP →2-C et RPC→3-C++ CORBA → 4-Java et RMI→5-C# et Web Service→6-Java et EJB→7- A suivre .....

  Avantages: développement plus rapide et plus efficace.
- Si on veut profiter des nouvelles technologies et de leurs avantages :
    - Nécessite d'adapter une application à ces technologies.
    - Le coût de cette adaptation est très élevé donc on doit réécrire le code d'application.
- Exemple :Application de calculs scientifiques distribués sur un réseau de machines Passage de C/RPC (Paradigme procédural)à Java/EJB (objet/composant):
    - Impossibilité de reprendre le code existant (différence des Paradigmes)
- Nécessité de découpler ***la logique métier*** et de ***la mise en œuvre*** technologique
    - C'est l'un des principes fondamentaux de l'ingénierie dirigée par les modèles : ***Séparation des préoccupations***

Définition : Ingénierie Dirigée par les Modèles

L'Ingénierie IDM est une approche de développement qui met à la disposition de l'utilisateur des outils, des concepts et des langages afin de simplifier et de mieux maîtriser le processus de développement de systèmes. En plus, elle permet aussi d'augmenter la productivité, la qualité, la réutilisabilité et l'évolution de ces systèmes [5]*.

Les modèles sont considérés comme des éléments de base. Les outils permettant de créer et d'exploiter ces modèles sont construits autour des concepts de:

- Méta-modélisation
- Transformation de modèles

Fondamental

Que propose l'IDM ?

- Besoin de modéliser/spécifier
  - A un niveau abstrait ***la partie métier***
  - La plate-forme de ***mise en oeuvre***
  - De projeter ce niveau abstrait sur une plateforme
- Modéliser les applications à un haut niveau d'abstraction
  - Place le ***modèle*** au cœur du processus de conception
  - ***Transformation*** de modèles pour passer d'un niveau à l'autre
  - ***Génère le code*** de l'application à partir des modèles
  - Possibilité de ***contrôler***, ***simuler*** et ***tester*** à différents niveaux

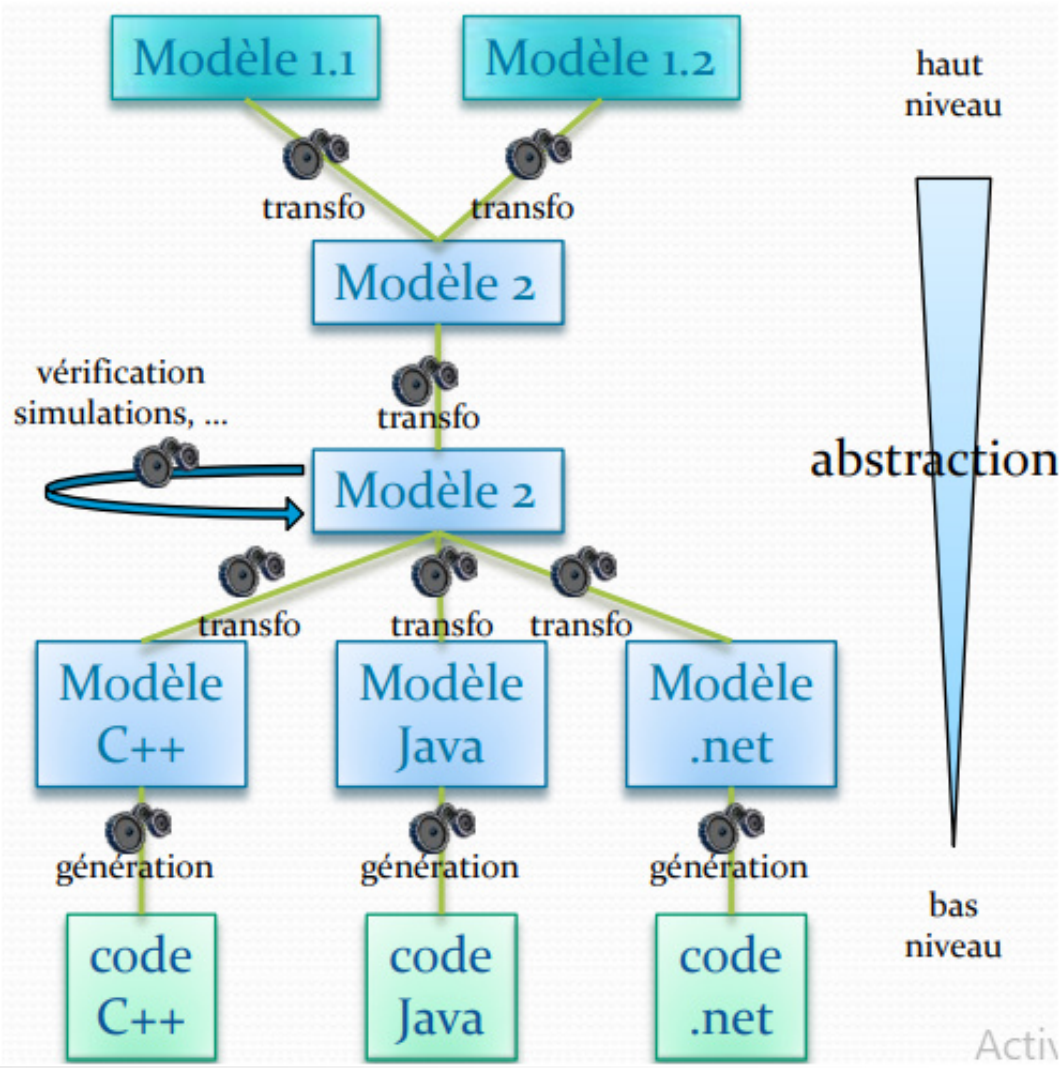


Figure 2.1 : Exemple de transformations de modèles

2. Modèles

Définition : Qu'est-ce qu'un modèle?

- Description / abstraction des objets du monde réel.
- Quelque chose avec une méta description de la façon dont il devrait être structuré.
- Objets et relations (un graphique).
- Vue simplifiée d'un système.

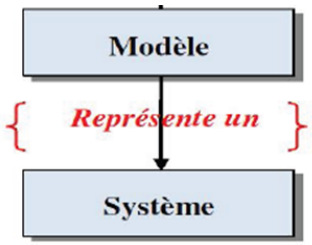


Figure 2.2 : Relation entre système et modèle

Fondamental : But d'un modèle

- Faciliter la compréhension d'un système complexe
- Simuler le fonctionnement d'un système complexe

Exemple : Modéles

- Réseaux de Petri (Rdp)
- Diagrammes UML
- Modèle BPMN
- Modèle Entité/Association
- Modèle météorologique

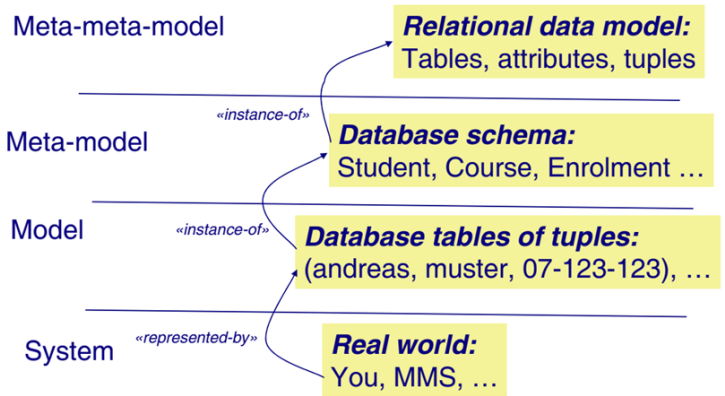


Figure 2.3 : Exemple de l'architecture à quatre niveaux dans les Bases de Données

3. Méta-modèles

3.1. Qu'est-ce qu'un méta-modèle?

- Pour passer à une vision productive, il faut que les modèles soient bien définis.
- La définition d'un langage de modélisation prend la forme d'un modèle, appelé Méta-modèle.
- Un méta-modèle est un modèle qui définit précisément les concepts manipulés dans les modèles ainsi que les relations entre ces concepts.
- Base de l'IDM permettant de développer des outils capables de manipuler les modèles.
- Assurer la correction syntaxiques des modèles.

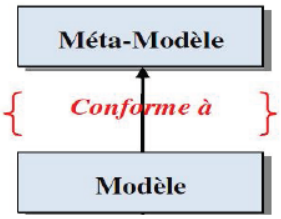


Figure 2.4 : Relation entre modèle et méta-modèle

Fondamental : But de la méta-modélisation

Définir des langages de modélisation ou des langages de manière générale.

3.2. Principales normes de modélisation OMG

- ***MOF*** : Meta-Object Facilities
  Langage de définition de méta-modèles
- ***UML*** : Unified Modelling Language
  Langage de modélisation
- ***CWM*** : Common Warehouse Metamodel
  Modélisation ressources, données, gestion d'une entreprise
- ***OCL*** : Object Constraint Language
  Langage de contraintes sur modèles
- ***XMI*** : XML Metadata Interchange
  Standard pour échanges de modèles et méta-modèles entre outils

3.3. Hiérarchie de Modélisation à 4 niveaux

Définition

L'OMG définit les quatre niveaux de modélisation suivants [4]*:

- M0 : système réel, système modélisé
- M1 : modèle du système réel défini dans un certain langage
- M2 : méta-modèle définissant ce langage
- M3 : méta-méta-modèle définissant le méta-modèle (M3 est le MOF)

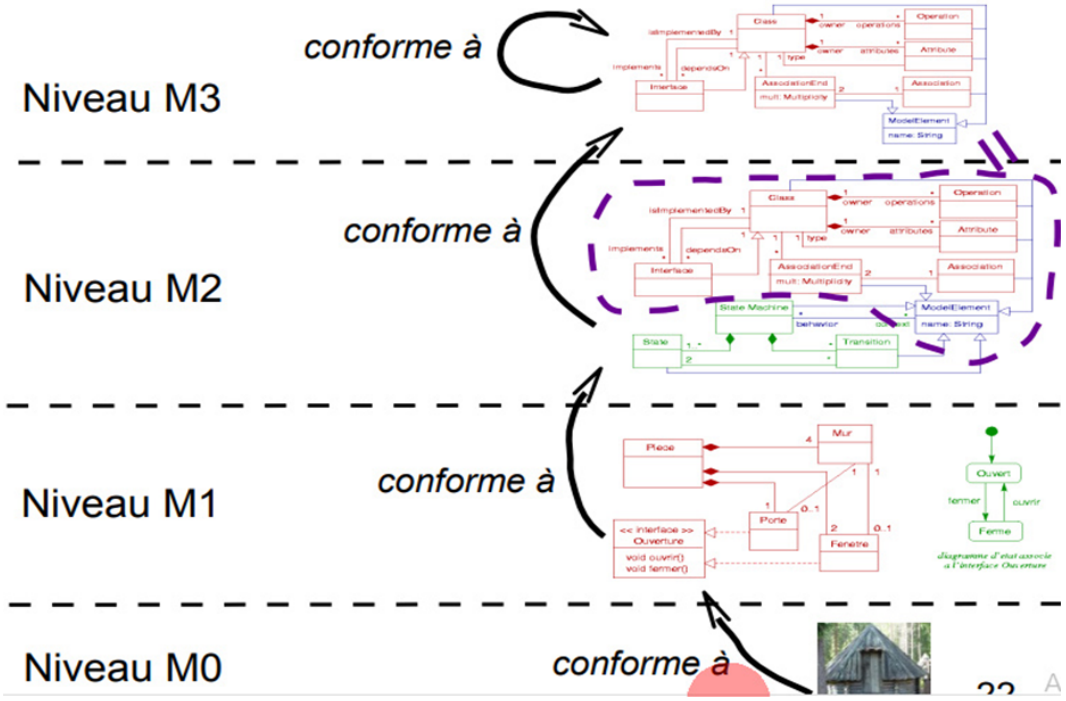


Figure 2.5 : Exemple Hiérarchie de Modélisation

3.4. Syntaxe abstraite et concrète d'un modèle

Définition

Un modèle peut être défini via la syntaxe abstraite ou concrète :

- Syntaxe Abstraite :
  - Les éléments et leurs relations sans une notation spécialisée.
  - Correspond à ce qui est défini au niveau du méta-modèle, à des instances de méta-éléments.
- Syntaxe Concrète :
  - Syntaxe graphique ou textuelle définie pour un type de modèle.
  - Plusieurs syntaxes concrètes possibles pour une même syntaxe abstraite.

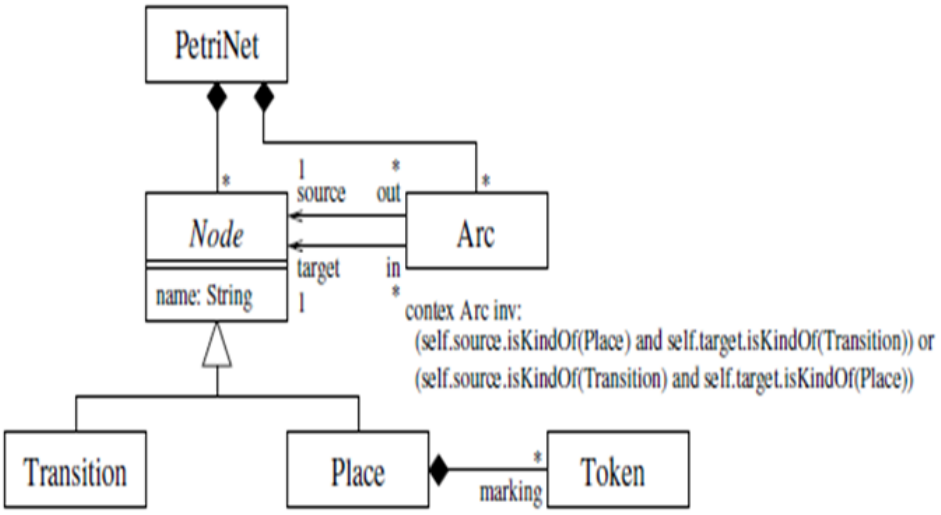


Figure 2.7 : Méta-modèle des Réseaux de Petri

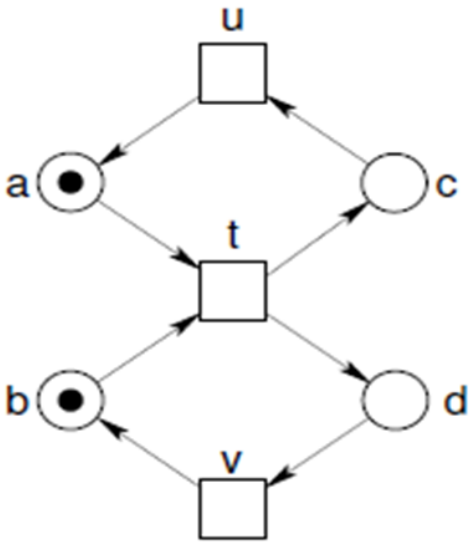


Figure 2.6 : Syntaxe Concrète d'un modèle de réseau de Petri

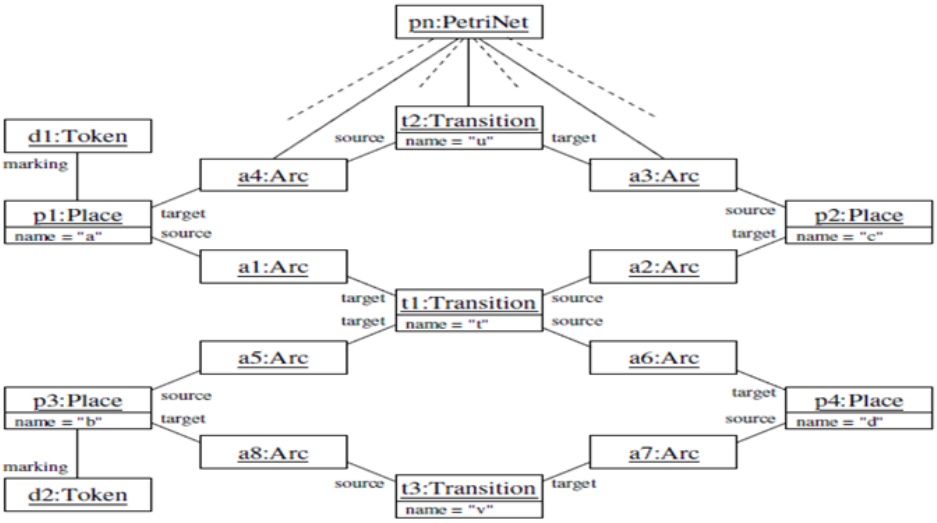


Figure 2.8 : Syntaxe Abstraite d'un modèle de réseau de Petri

4. Transformation de Modèles

Définition : Principe de transformation de modèles

- Une transformation de modèles consiste à passer d'un modèle source à un modèle cible.
- Différentes sémantiques :
    - Optimisation
    - Génération de code
    - Raffinement
- Deux grandes classes de transformation de modèles [24]*:
    - Transformations de type: Modèle vers Code (M2T).
    - Transformations de type: Modèle vers Modèle (M2M).

- Le passage de l'un à l'autre est décrit par des règles de transformation.
- Ces règles sont exécutées sur les modèles sources afin de produire les modèles cibles.
- Une transformation endogène : modèles source et cible conformes au même méta-modèle.
- Une transformation exogène : modèles source et cible conformes à des méta-modèles différents.

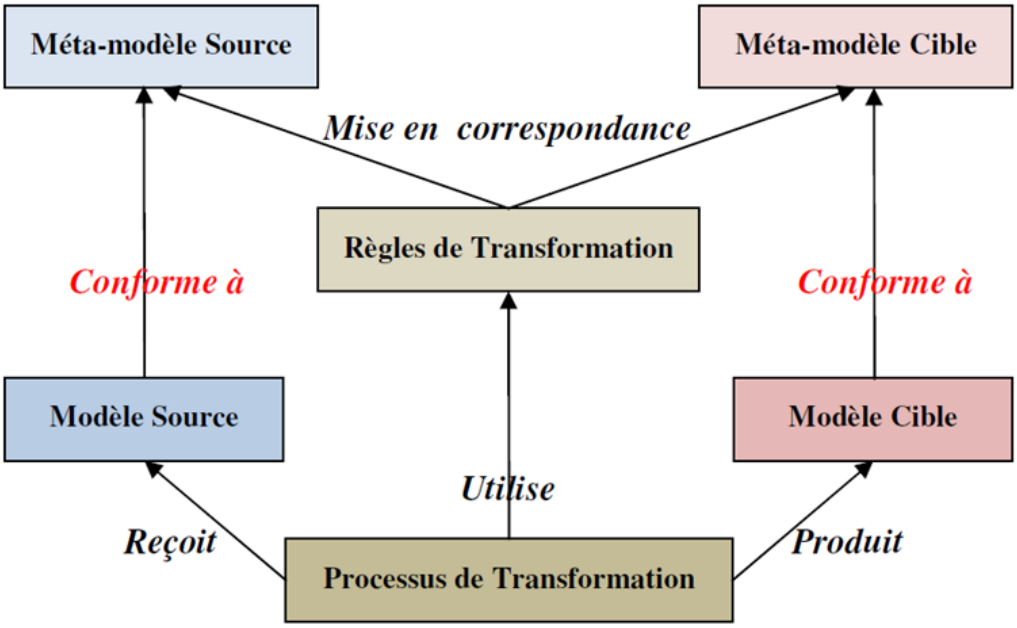


Figure 2.9 : Concepts de base de la transformation de modèles

Exemple : Transformations de modèles

- M2M exogène : Transformation des diagrammes UML vers les Réseaux de Petri.
- M2T :Génération de code Java à partir des diagrammes UML.
- M2M exogène : Transformation d'un diagrammes UML vers un schéma de BDD.
- M2M endogène Réduction des Réseaux de Petri.

5. Architecture MDA

Définition : Principe de la vision MDA

- MDA [3]*, [6]*, [7]* vise à modéliser des applications et des systèmes indépendamment de l'implémentation cible (niveau matériel ou logiciel), ce qui permet la réutilisation des modèles développés.
- Les modèles ainsi créés (PIM :Platform Independant Model) seront transformés pour obtenir des modèles d'applications spécifiques à la pateforme cible (PSM : Platform Specific Model).
- Des outils de génération automatique du code, permettent par la suite de générer les programmes directement à partir des modèles.
- Cette approche, permet en plus de faire évoluer aisément les applications et leurs architectures à partir des modèles.
- Par conséquent, Le MDA s'investit dans un grand atelier dont le défit relève techniquement de la manipulation des modèles. Dans ce cadre spécifique, la transformation de modèle basé sur les techniques de Méta-modélisation et l'ingénierie de modèles ouvre un champ d'investigation prometteur pour la recherche.

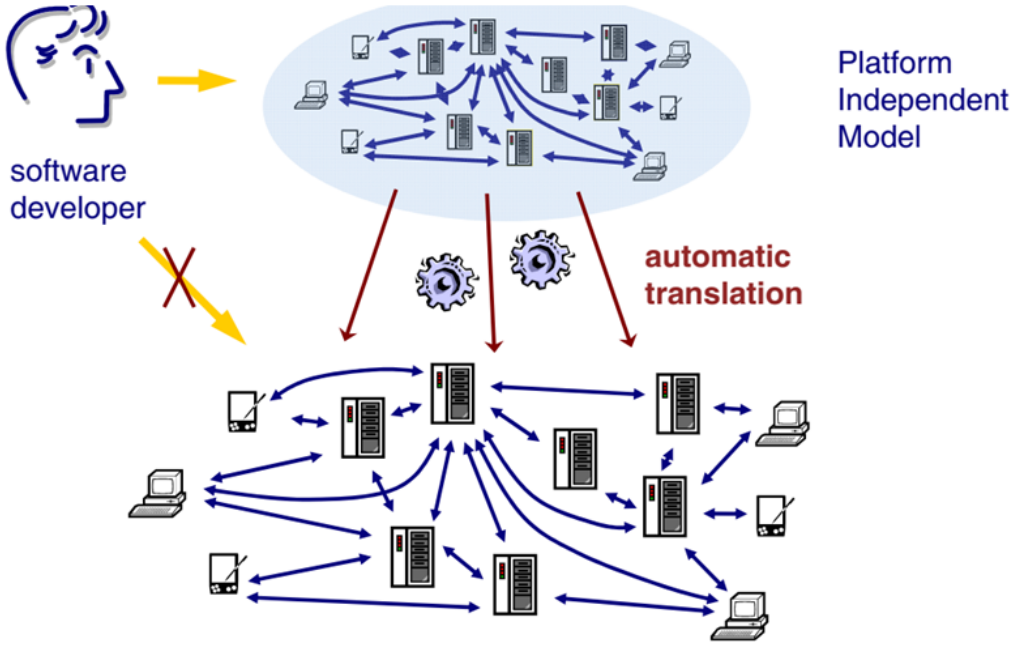


Figure 2.10 : La vision MDA

Complément : Principaux niveaux de modèles MDA

Le MDA définit deux principaux niveaux de modèles [4]*:

- PIM : Platform Independent Model
  - Modèle spécifiant une application indépendamment de la technologie de mise en œuvre.
  - Uniquement spécification de la partie métier d'une application.
- PSM : Platform Specific Model
  - Modèle spécifiant une application après projection sur une plate-forme technologique donnée.
  - Relation entre les niveaux de modèles.

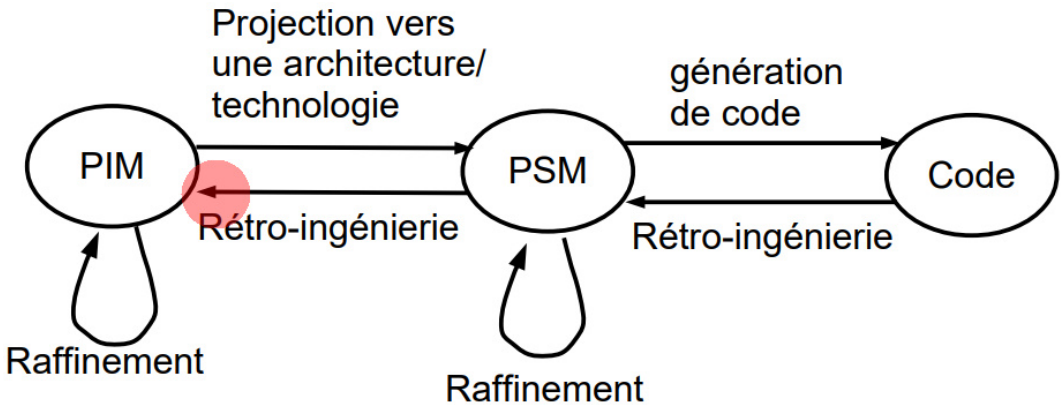


Figure 2.11 : Les modèles et les transformations dans l'approche MDA

6. Modélisation Multi-Paradigme

Définition : Les trois directions

La Modélisation Multi-Paradigme contient trois directions de recherche orthogonales [1]*:

- Modélisation Multi-Abstraction: la relation entre des modèles à des niveaux d'abstraction différents.
- Modélisation Multi-Formalisme: le couplage et la transformation entre des modèles décrits dans plusieurs formalismes.
- Méta-Modélisation: la description des formalismes et des langages de modélisation.

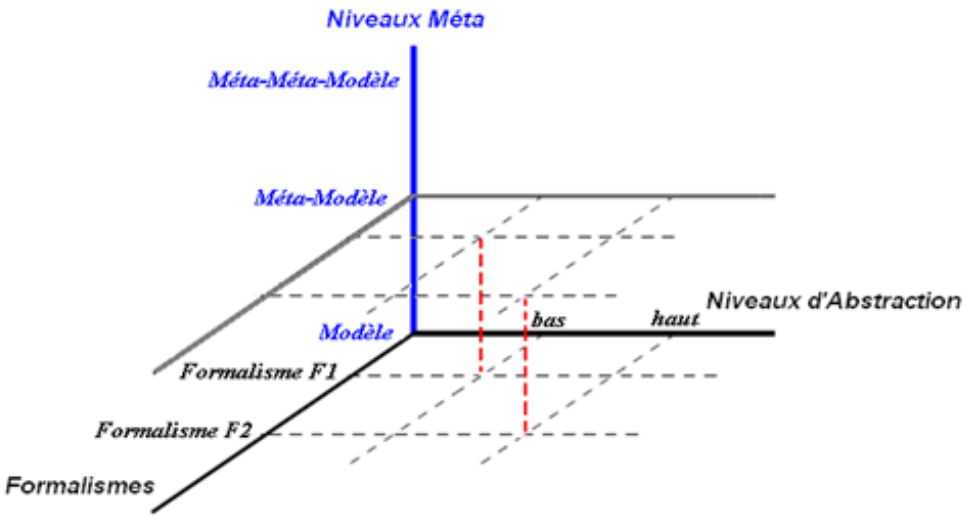


Figure 2.12 : Modélisation Multi-paradigme: les trois directions

Complément : Modélisation Multi-Abstraction

- Un système peut être décrit par plusieurs modèles, et dans plusieurs niveaux d'abstraction ou de détail.
- Chaque niveau d'abstraction est mieux adapté à une tâche particulière.
- Le processus de changement d'abstraction est un type de transformation de modèle.
- La dérivation automatique de modèles à différent niveaux d'abstraction augmente la qualité des modèles pour mieux maîtriser et comprendre tous les détails du système.

Complément : Modélisation Multi-Formalisme

- La complexité croissante des systèmes exige l'introduction de plusieurs formalismes.
- Chaque composant/aspect du système est modélisé en utilisant le formalisme et l'outil d'analyse approprié.
- Le comportement global du système est évalué en transformant les modèles des composants/aspects vers un formalisme unique.
- L'automatisation de ces transformations par des supports outillés augmente la productivité de la modélisation.
    - L'ajout de nouveaux formalismes sans fournir beaucoup d'efforts.
    - L'utilisation de nouveaux environnements de simulation et d'analyse.

Complément : Méta-Modélisation

- Besoin de multiples éditeurs
- La réalisation de ces éditeurs est relativement complexe et coûteuse
  Solution: principe de la Méta-modélisation
    - Modéliser les formalismes
    - Générer automatiquement des éditeurs pour ces formalismes

- La seule information à fournir est le méta-modèle du langage sans se préoccuper des détails d'implémentation de l'éditeur
- Avantages: préservation des acquis et flexibilité
    - Adaptation aux nouveaux besoins de modélisation
    - Ajout de contraintes spécifiques à un domaine

Complément : Mettre en relation les trois directions

- Transformation de modèles permet de relier ces trois dimensions pour cumuler leurs avantages
    - Combiner et transformer les différents formalismes
    - Utiliser des formalismes et des outils spécifiques au domaine d'application
    - Vérifier la cohérence entre les différentes vues/aspects du système
- Différentes manipulations de modèles
    - La transformation de formalismes
    - L'optimisation de Modèle
    - La simulation
    - La génération de code
- Les modèles et les méta-modèles sont des graphes
    - les transformations entre modèles peuvent être décrites et réalisées par des Transformations de Graphes

7. Conclusion

L'ingénierie dirigée par les modèles (IDM) est une approche de développement de logiciels qui met à la disposition de l'utilisateur des concepts, des langages et des outils pour réduire la complexité du développement. Dans ce chapitre, nous avons présenté les concepts de base de l'ingénierie dirigée par les modèles: le modèle, le méta-modelé, la transformation de modèles, l'approche MDA et la modélisation multi-paradigmes.

V Chapitre 03 : Transformation de Graphes

1. Introduction

Dans ce chapitre, nous présentons les principes théoriques et pratiques de la transformation de graphes. Dans un premier temps, nous abordons les concepts de base suivants : la notion de graphe, le système de transformation de graphes, la grammaire de graphes et les outils de la transformation de graphes. Ensuite, nous présentons une transformation des modèles BPMN vers les réseaux de Petri en utilisant l'outil AToMPM. Cette transformation est basée sur la méta-modélisation et la transformation de graphes. Enfin, nous appliquons la transformation proposée sur un modèle BPMN.

2. Concepts de Transformations de Graphes

2.1. Concept de Graphe

Définition : Un graphe

Un graphe est constitué de sommets ou nœuds qui sont reliés par des arêtes. Plus formellement, on appelle graphe G = {S, A} tel que:

- S : ensemble fini non vide d'éléments appelés sommets ou nœuds.

- A : ensemble fini non vide de paires de sommets appelés arcs.

- A ⊆ X × X = {(s, t)|s, t ∈ S}, chaque arc de A relie deux sommets de S.

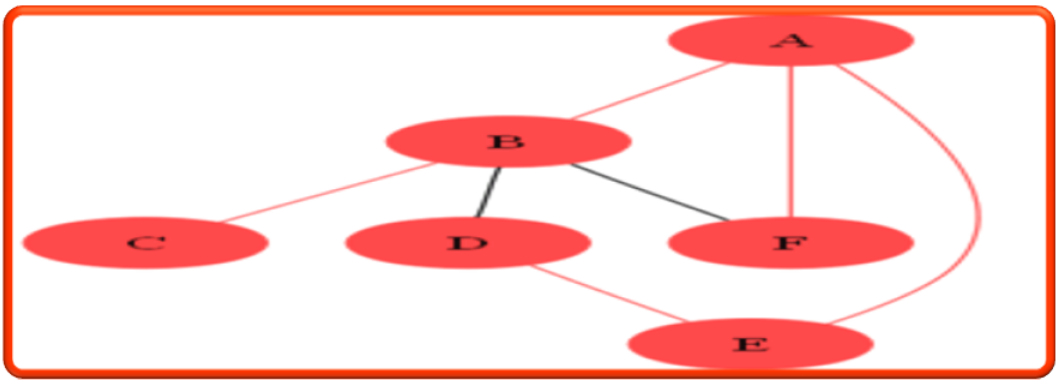


Figure 3.1 Graphe non orienté

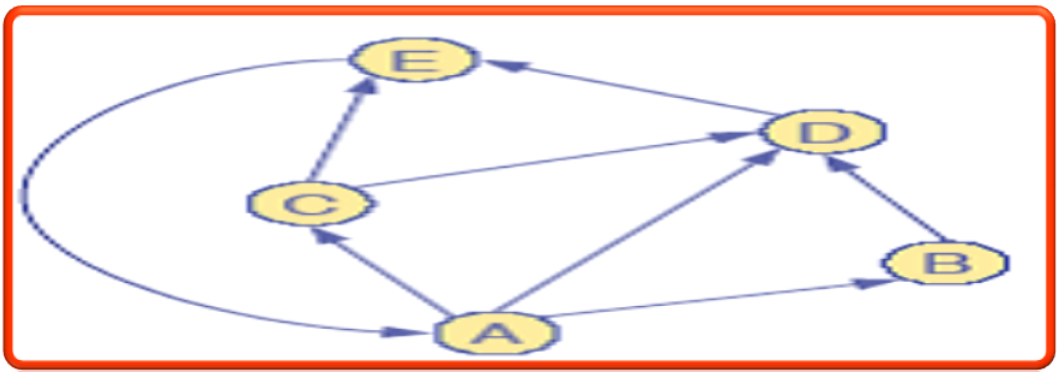


Figure 3.2 Graphe orienté

Définition : Graphe étiqueté

Un graphe étiqueté est un graphe orienté dans lequel les arcs possèdent un ensemble non vide d'étiquettes.

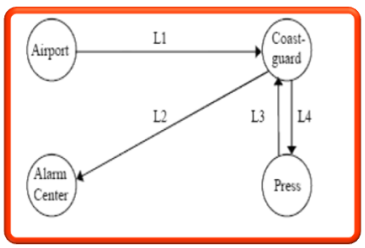


Figure 3.3: Graphe orienté étiqueté

Définition : Sous-graphe

sous graphe G'(S',A') d'un graphe G(S,A) est un graphe composé d'un sous ensemble de sommet S'∈ S et d'un sous ensemble d'arêtes A'∈A tel que A' représente les arêtes reliant les sommets S' dans le graphe G'.

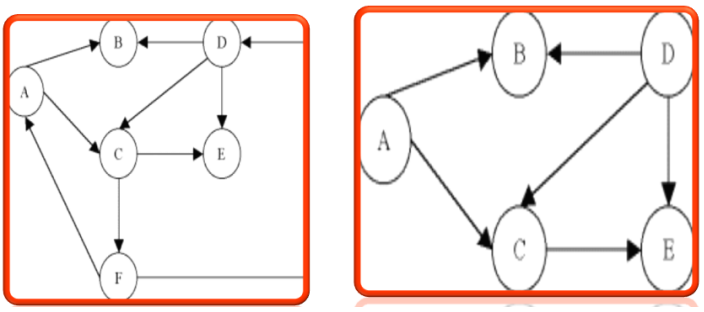


Figure 3.4: Graphe et sous-graphe

2.2. Grammaires de Graphes

Définition

Une grammaire de Graphe[9]* est généralement définie par un triplet: GG = (P, S, T)

Où : P est un ensemble de règles, S est un un graphe initial, T est un ensemble de symboles.

Nous utilisons des grammaires de graphes pour :

- Exprimer la sémantique opérationnelle (spécification du simulateur)
- Transformer des modèles en modèles comportementaux équivalents exprimés dans un autre formalisme.
- Optimiser les modèles.
- Générer du code pour un outil particulier (sémantique dénotationnelle).

Complément : Principe d'une règle

Une règle de transformation de graphe[9]* est définie par : r = (L, R, K, glue, emb, cond). Elle consiste en:

- Deux graphes ***L*** graphe de côté gauche et ***R*** graphe de côté droit.
- Un sous graphe ***K*** de ***L***.
- Une occurrence ***glue*** de ***K*** dans ***R*** qui relie le sous graphe avec le graphe de côté droit.
- Une relation d'enfoncement ***emb*** qui relie les sommets du graphe de côté gauche et ceux du graphe du côté droit.
- Un ensemble ***cond*** qui spécifie les conditions d'application de la règle.

Méthode : Application des règles

L'application d'une règle r= (L, R, K, glue, emb, cond) à un graphe G produit un graphe résultant H. Le graphe H fourni, peut être obtenu depuis le graphe d'origine G en passant par les cinq étapes suivantes :

1. Choisir une occurrence du graphe de côté gauche L dans G.
2. Vérifier les conditions d'application d'après cond.
3. Retirer l'occurrence de L (jusqu'à K) de G ainsi que les arcs pendillé, c.-à-d. tous les arcs qui ont perdu leurs sources et/ou leurs destinations. Ce qui fournit le graphe de contexte D de L qui a laissé une occurrence de K.

4. Coller le graphe de contexte D et le graphe de côté droit R suivant l'occurrence de K dans D et dans R. c'est la construction de l'union de disjonction de D et R et, pour chaque point dans K, identifier le point correspondant dans D avec le point correspondant dans R.
5. E. Enfoncer le graphe du côté droit dans le graphe de contexte de L suivant la relation d'enfoncement emb . L'application de r sur 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 ou simplement par G ⇒ H.

2.3. Système de transformation de graphes

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). 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 [18]*.

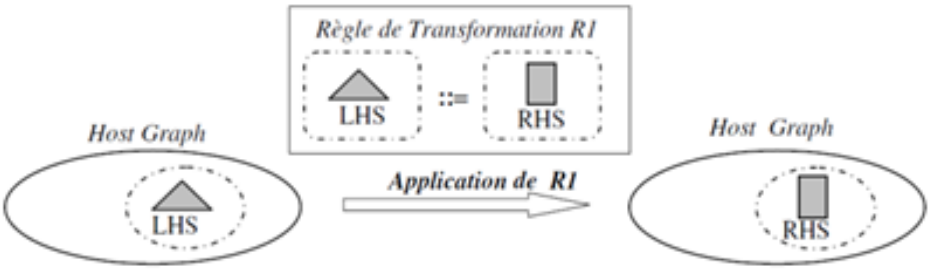


Figure 3.5 Principe de l'application d'une règle de transformation

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 [18]*.

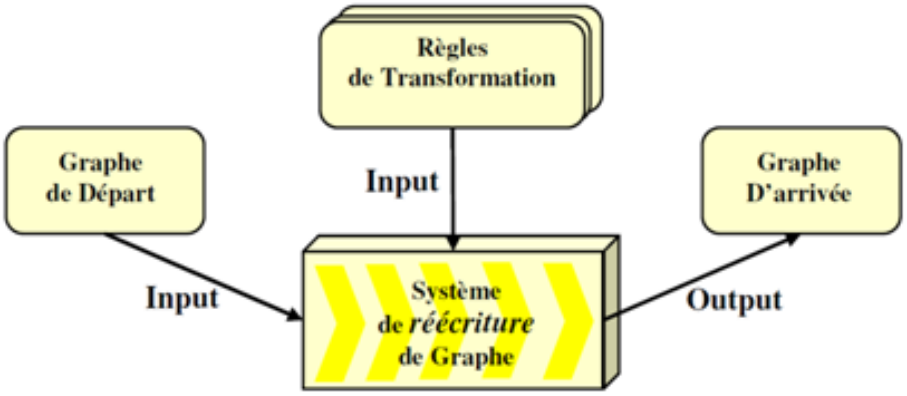


Figure 3.6: Système de réécriture de graphes

Fondamental : Avantage

Approche formelle fondée sur des ***bases mathématiques***:

- La théorie des graphes
- Les grammaires formelles
- La réécriture de termes

2.4. Outils de transformation de graphes

Plusieurs outils de transformation de graphes existent actuellement, parmi lesquels :

- AGG [AGG]:The Attributed Graph Grammar System.
- AToM3: A Tool for Multi-formalism and Meta-Modelling.
- AToMPM : A Tool for Multi-Paradigm Modeling.
- VIATRA :VIsual Automated model TRAnsformations.
- FUJABA: From UML to Java and back again.
- GreAT : The Graph Rewrite And Transformation tool suite.

L'outil AToMPM

- AToMPM[8]* signifie « A Tool for Multi-Paradigm Modeling ».
- Outil pour la modélisation multi-paradigme.
- Successeur de AToM3.
- Il basé complètement sur le Web.
- Il fonctionne sur le nuage.
- Effectuer des transformations de modèles, et manipuler et gérer des modèles.
- AToMPM possède :

    1. Une couche de Méta-modélisation qui permet La modélisation graphique d'un formalisme ainsi que la génération automatique d'un outil pour manipuler les différents modèles décrits dans le formalisme spécifié.
    2. Un système de réécriture de graphes qui permet Les manipulations de modèles par application itérative des règles d'une Grammaire de Graphes.

3. Transformation des modèles BPMN vers les Réseaux de Petri

3.1. Les formalismes source BPMN et cible Rdp

Définition : Notation BPMN

BPMN "Business Process Model and Notation" est un langage pour la modélisation des processus métier.

L'objectif principal du langage BPMN est de fournir une notation standard qui soit simple, facile et compréhensible par tous les utilisateurs métiers.

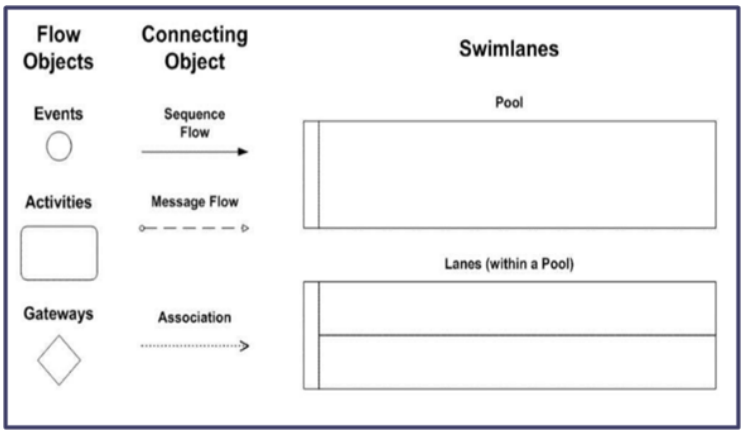


Figure 3.7: Les principaux éléments de la notation BPMN

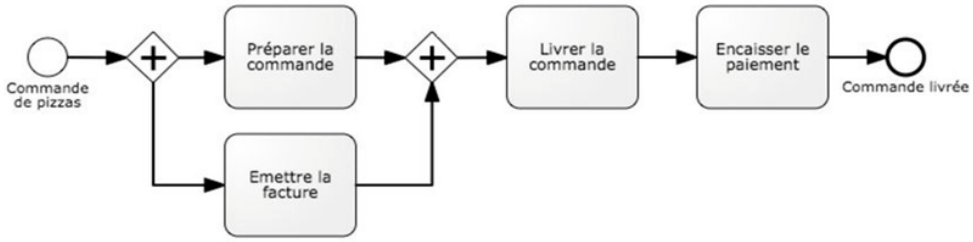


Figure 3.8 : Exemple d'un modèle de la pizzeria avec BPMN

Définition : Réseaux de Petri

Un réseau de Pétri (aussi connu comme un réseau de Place/Transition) est un modèle graphique et mathématique permettant de modéliser et de vérifier le comportement dynamique des systèmes [13]*.

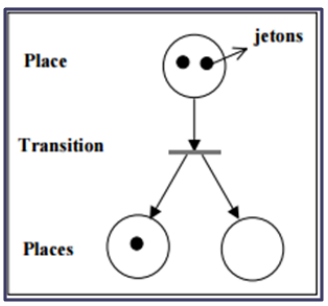


Figure 3.9 : Exemple d'un modèle RdP

Motivations de la transformation des modèles BPMN vers RdP

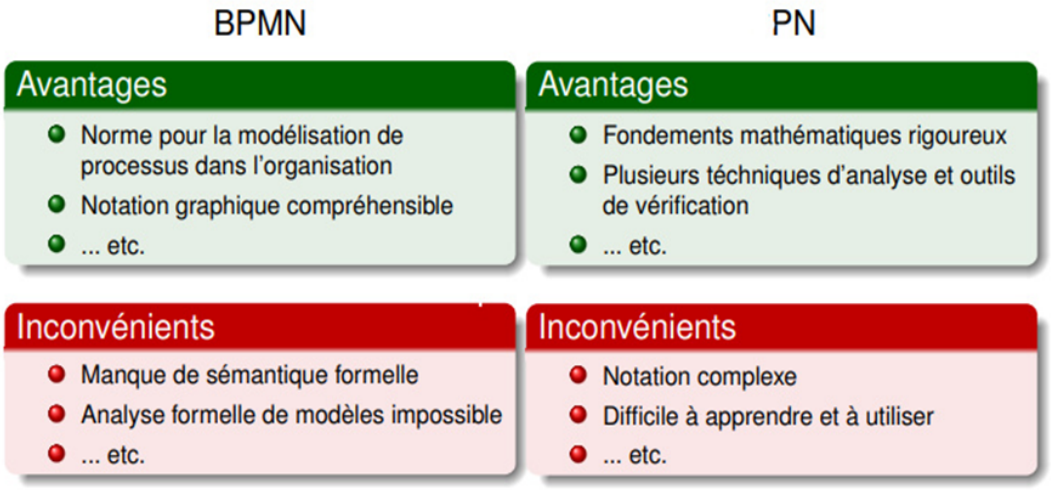


Figure 3.10 : Avantages et inconvénients des modèles source et cible

- Chacune des deux modèles (BPMN et Rdp) a des points forts et des points faibles.
- Les deux approches peuvent être combinées dans le sens où les inconvénients de l'un peuvent être surmontés grâce aux apports de l'autre.

Méthode

Afin de transformer les modèles BPMN vers les réseaux de Pétri, on suit les trois étapes suivantes:

1. Construction d'un méta-modèle pour le langage BPMN.
2. Construction d'un méta-modèle pour les réseaux de Pétri.
3. Définition des règles de la transformation (Grammaire de Graphes).

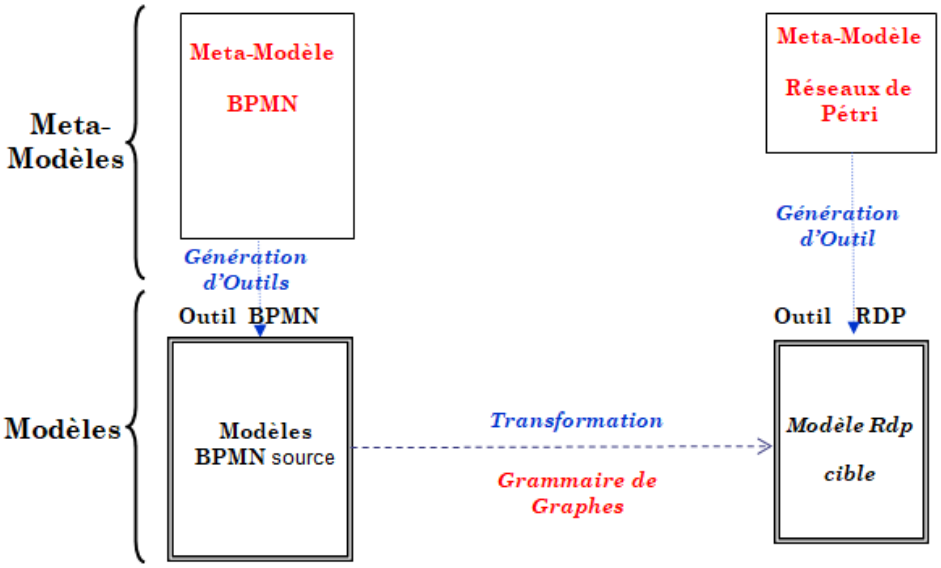


Figure 3.11 : Architecture de la transformation

3.2. Méta-Modélisation des BPMN et les RdP

Méta-Modélisation des BPMN

La figure 3.12 montre le méta-modèle (la syntaxe abstraite) des modèles BPMN. Ce méta-modèle regroupe les éléments de bases qui comprennent une activité (Activity), deux type de branchement (XOR et AND), et les flux de séquence (AND2Act, Act2XOR, XOR2Act, sequencearc).

La figure 3.13 montre la syntaxe concrète de BPMN.

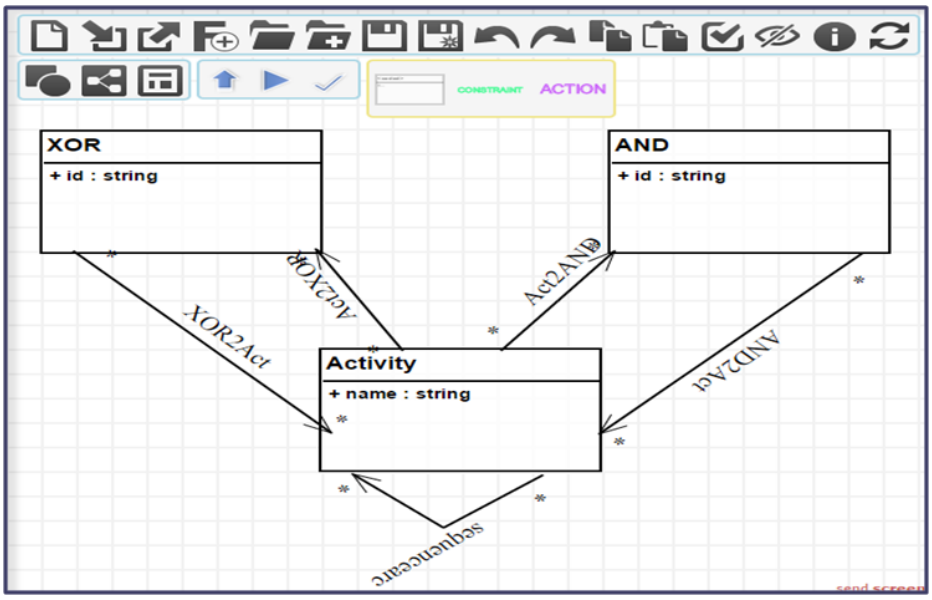


Figure 3.12 : Méta-Modèle de la notation BPMN

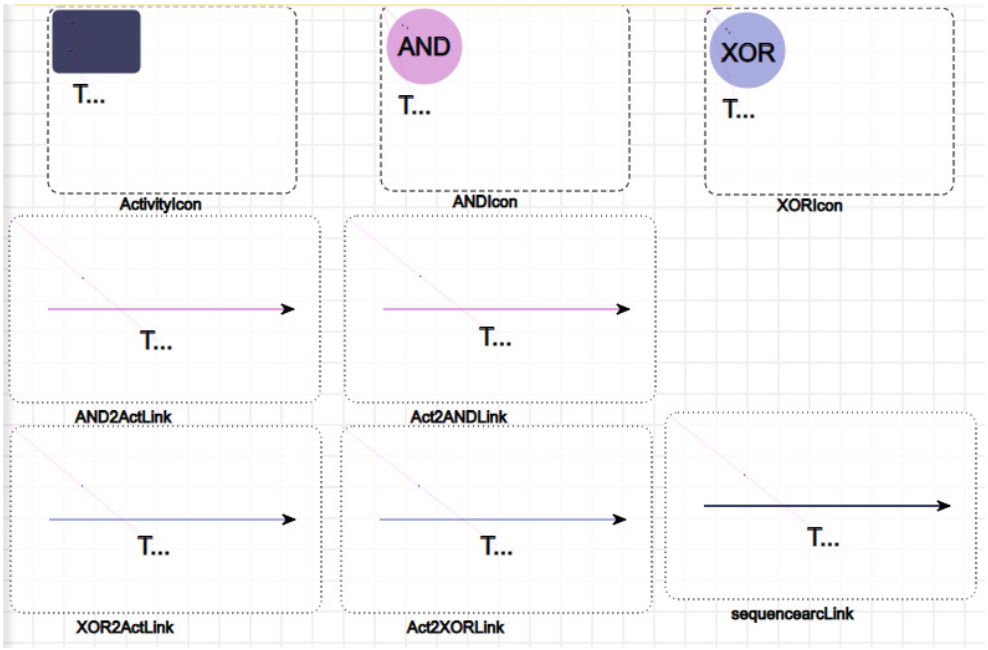


Figure 3.13 : Syntaxe concrète de la notation BPMN

Méta-Modélisation des Réseaux de Petri

La figure 3.14 montre le méta-modèle (la syntaxe abstraite) des Réseaux de Petri. Ce méta-modèle regroupe les éléments de bases qui comprennent une transition (Transition), et une place (Place), et deux types des arcs (T2P: outarc et P2T: inarc).

La figure 3.15 montre la syntaxe concrète des Réseaux de Petri.

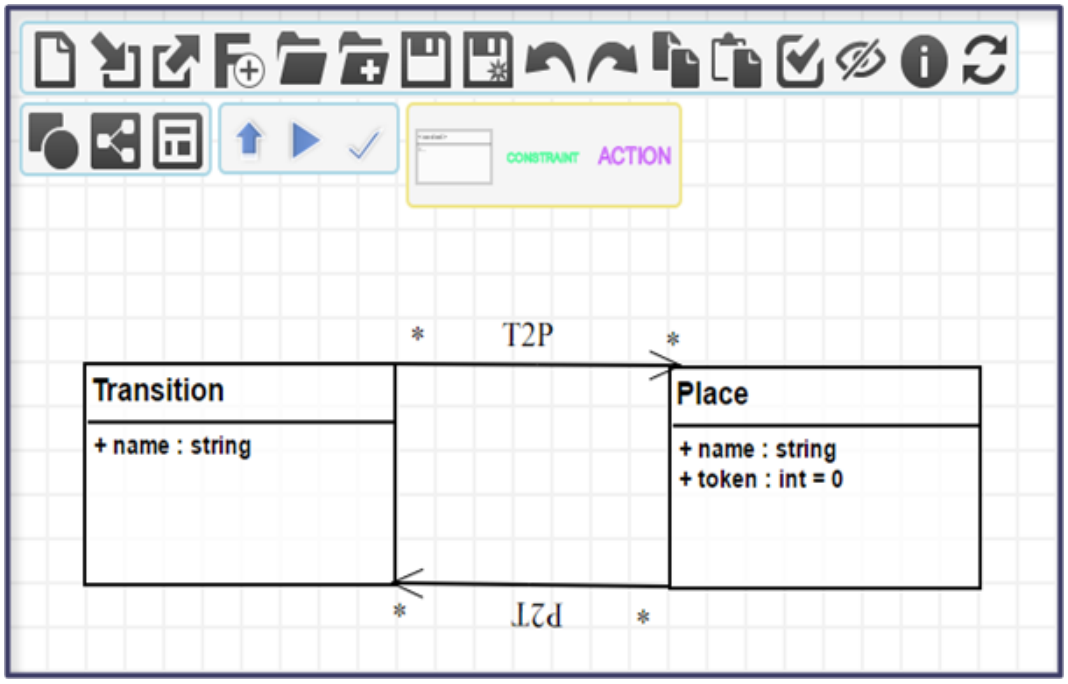


Figure 3.14 : Méta-Modèle des Réseaux de Petri

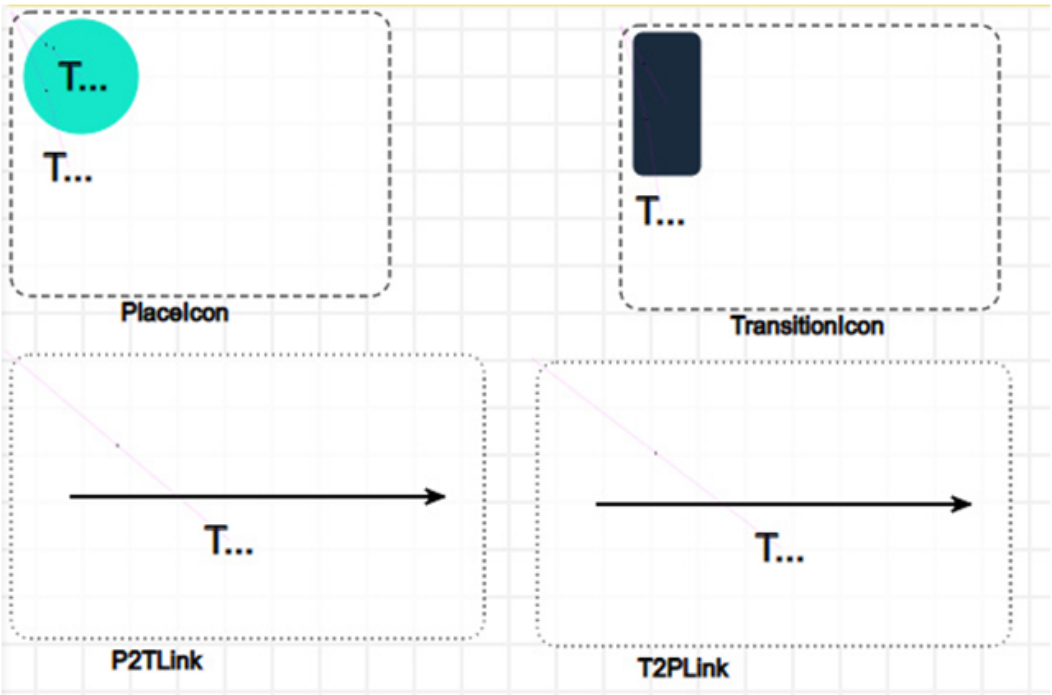


Figure 3.15 : Syntaxe concrète des Réseaux de Petri

3.3. Règles de la transformation

Définition : Idée de la transformation BPMN vers RDP

L'idée de la transformation des modèles BPMN vers les RdP est la suivante:

1. Les activités sont transformés en des transitions.
2. Les connecteurs XOR sont transformés en des places.
3. Les connecteurs AND sont transformés en des places.

Méthode : Grammaire de Graphes

Afin de produire les modèles BPMN équivalents au RdP, nous avons proposé les 9 règles suivantes [11]* :

- Règle 1 : R_Activity2Transition
- Règle 2 : R_sequencearc
- Règle 3 : R_xor2place
- Règle 4 : R_inXOR2inPL
- Règle 5 : R_outXOR2outPL
- Règle 6 : R_ANDrule
- Règle 7 : R_DeleteXOR
- Règle 8 : R_DeleteAND
- Règle 9 : R_DeleteActivity

Règle 1 R_Activity2Transition

Cette règle génère une nouvelle transition (dans RdP) pour chaque activité (dans BPMN).

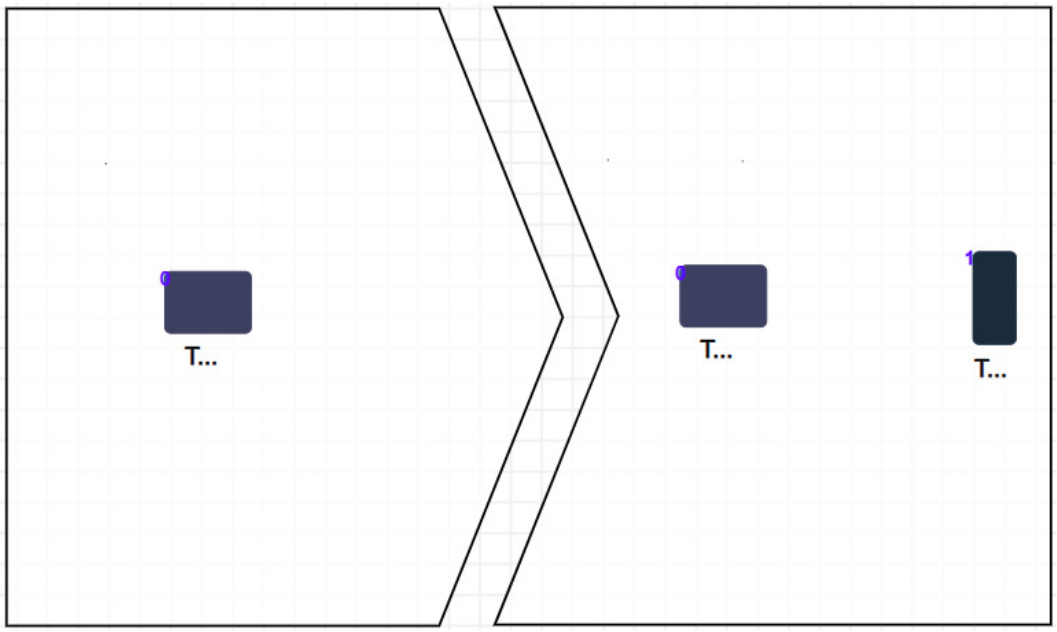


Figure 3.16 : Règle R_Activity2Transition

Règle 2 R_sequencearc

Cette règle est appliquée à chaque arc de séquence:

- Elle génère une nouvelle place avec un arc entrant et un arc sortant aux transitions.
- Elle supprime tous les arcs de séquence.

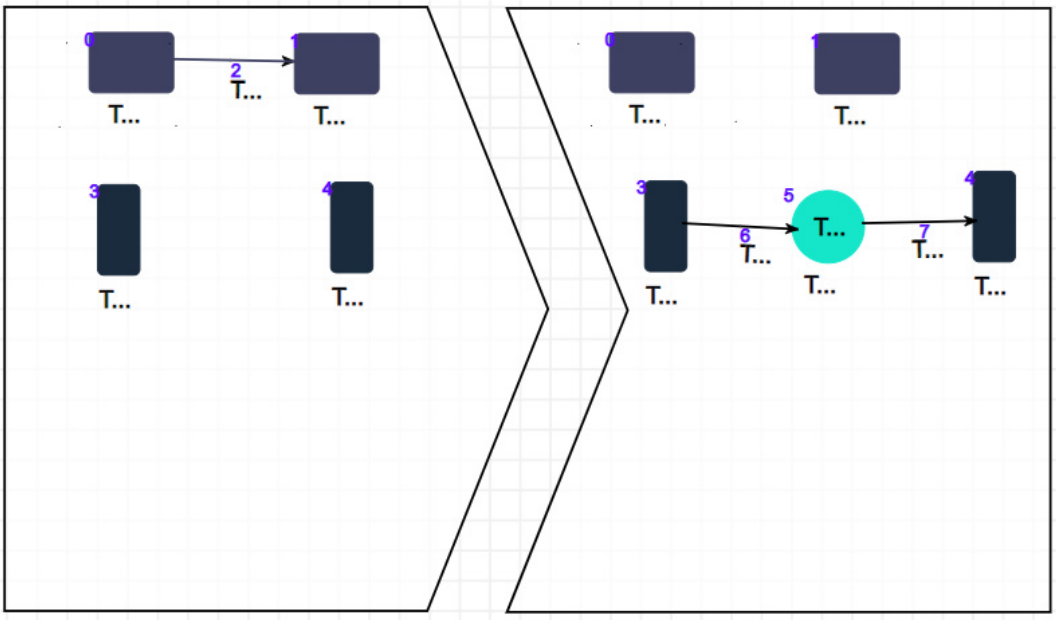


Figure 3.17 : Règle R_sequencearc

Règle 3 R_xor2place

Cette règle génère une nouvelle place pour chaque connecteur XOR.

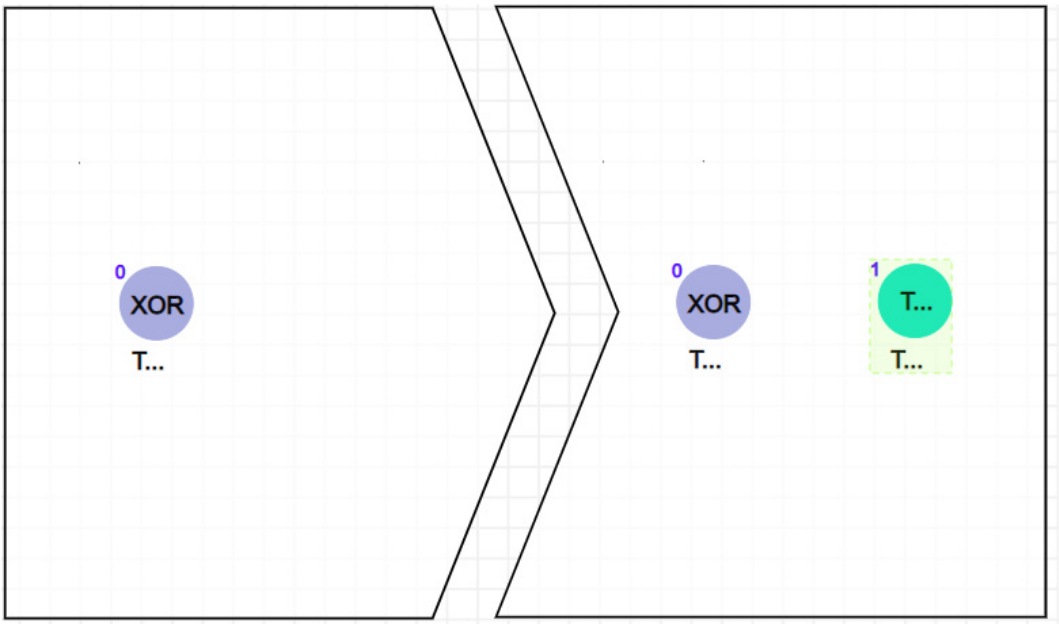


Figure 3.18 : Règle R_xor2place

Règle 4 R_inXOR2inPL

Cette règle génère un arc sortant de la transition (attachée à l'activité) vers la place et elle supprime tous les arcs (Act2XOR).

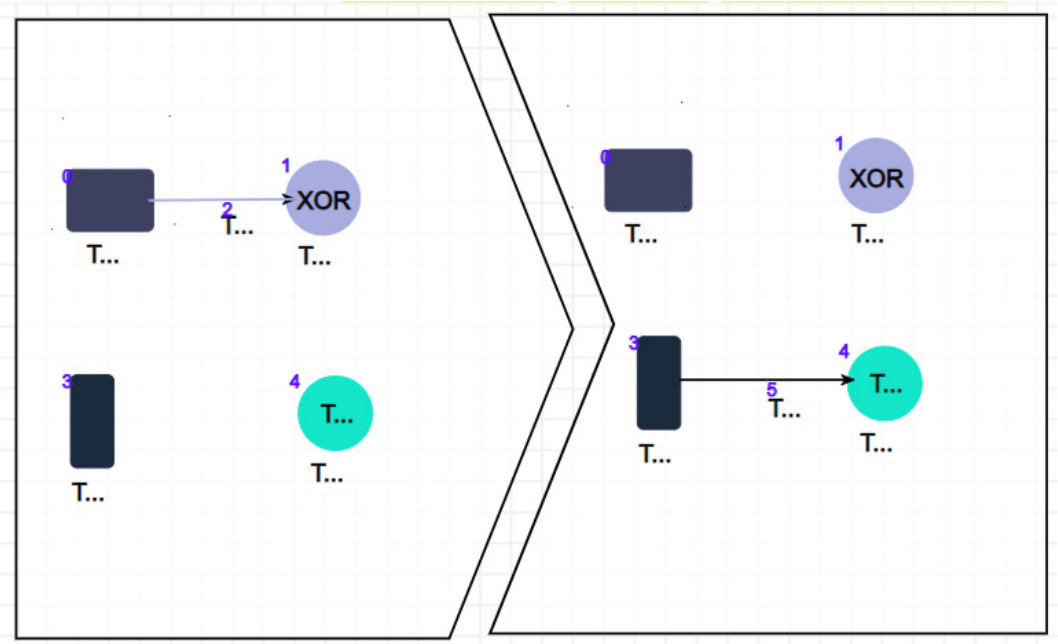


Figure 3.19 : Règle R_inXOR2inPL

Règle 5 R_outXOR2outPL

Cette règle génère un arc sortant de la place (attachée au lien XOR) vers la transition (attachée à l'activité) et elle supprime tous les arcs (XOR2Act).

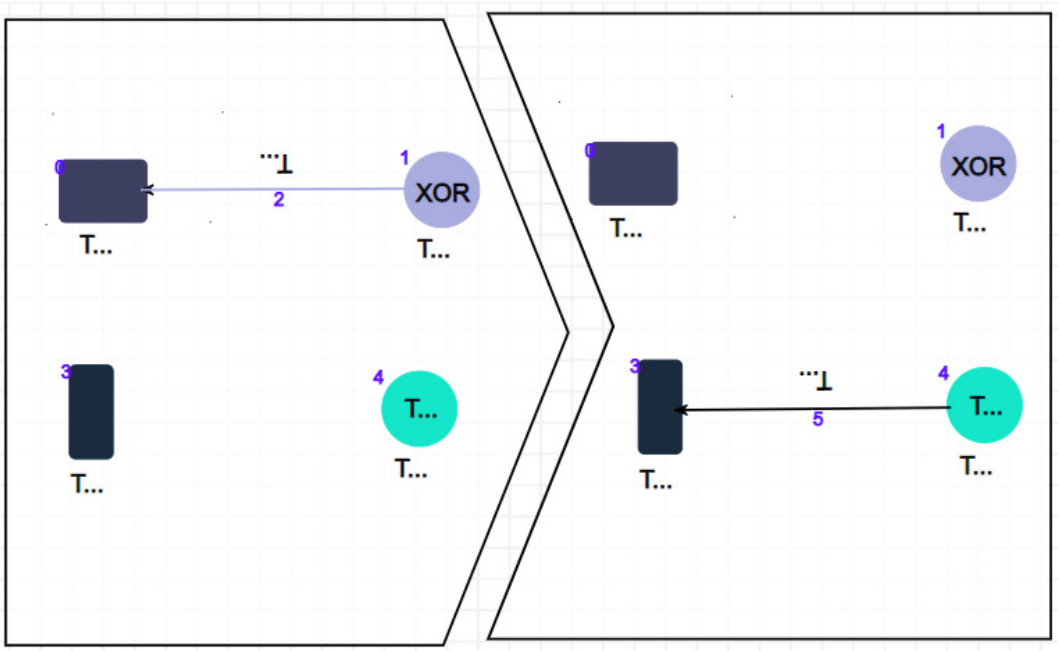


Figure 3.20 : Règle R_outXOR2outPL

Règle 6 R_ANDrule

Cette règle génère une nouvelle place avec un arc entrant et un arc sortant aux transitions (Liés aux activités).

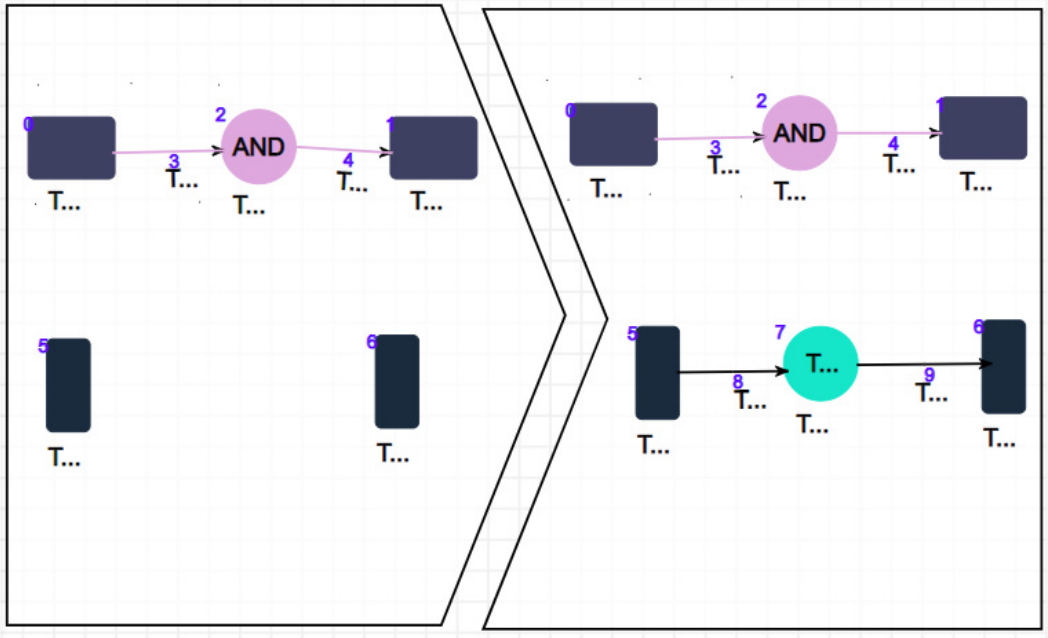


Figure 3.21 : Règle R_ANDrule

Règle 7 R_DeleteXOR

Cette règle supprime tous les connecteurs XOR du modèle BPMN.

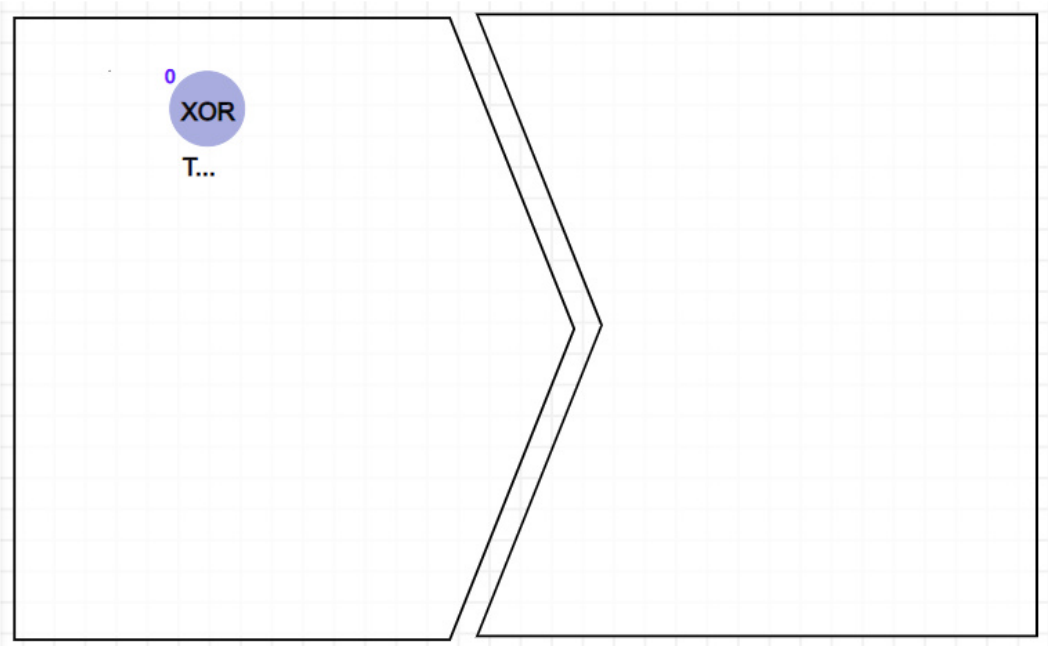


Figure 3.22 : Règle R_DeleteXOR

Règle 8 R_DeleteAND

Cette règle supprime tous les connecteurs AND du modèle BPMN.

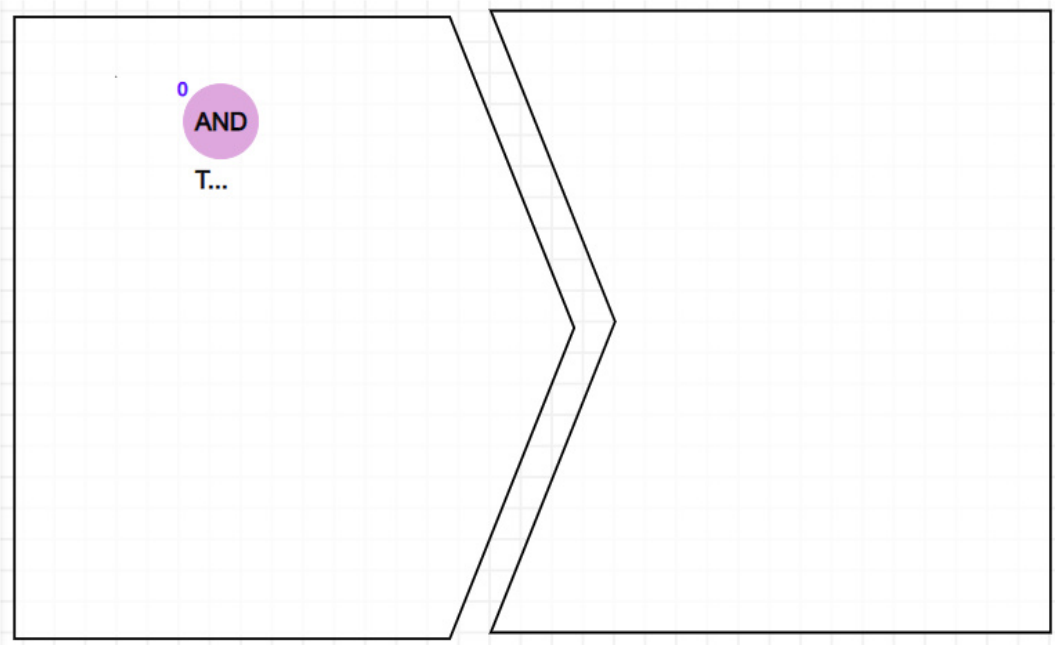


Figure 3.23 : Règle R_DeleteAND

Règle 9 R_DeleteActivity:

Cette règle supprime tous les activités du modèle BPMN.

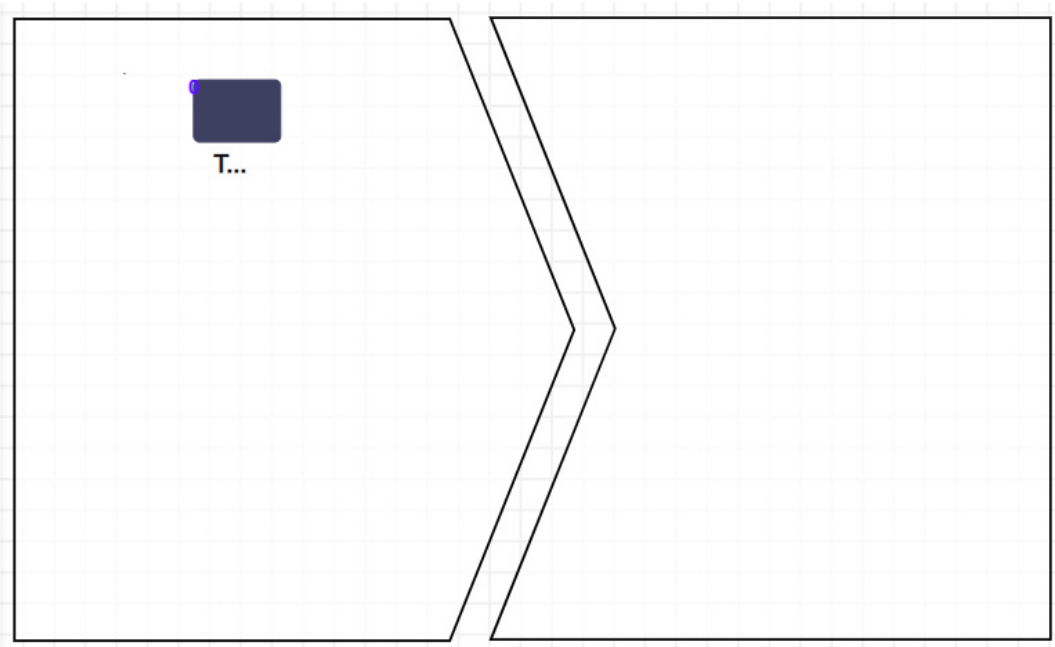


Figure 3.24 : Règle R_DeleteActivity

Processus de la transformation

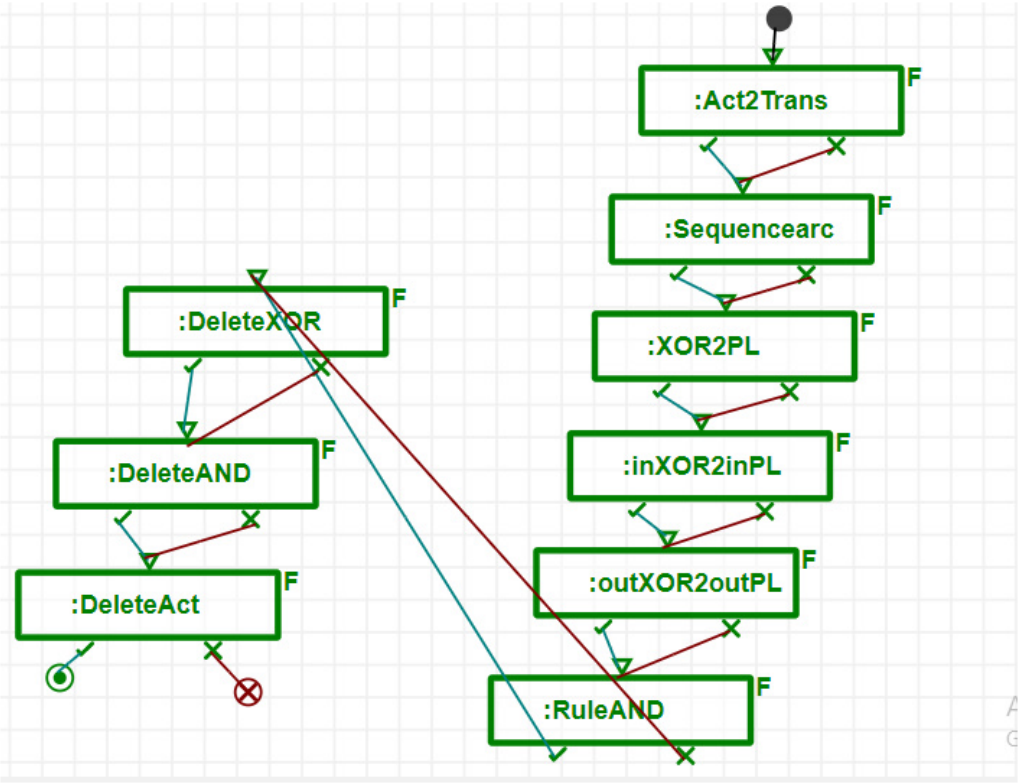


Figure 3.25 : Processus d'application des règles

La figure 3.25 montre le processus d'application des règles dans l'outil AToMPM. Ce processus est construit en utilisant le langage MoTiF.

4. Exemple de transformation d'un modèle BPMN

Exemple

Nous essayerons à travers cet exemple de montrer en détails l'application de la transformation proposée.

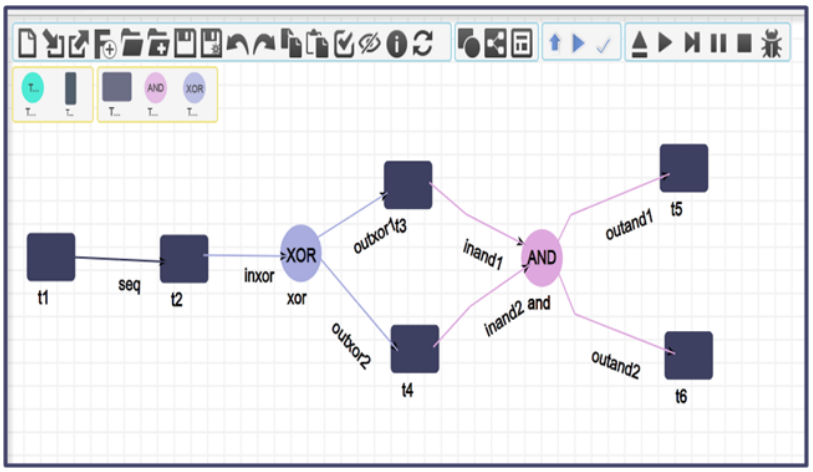


Figure 3.26 : Exemple d'un modèle BPMN source

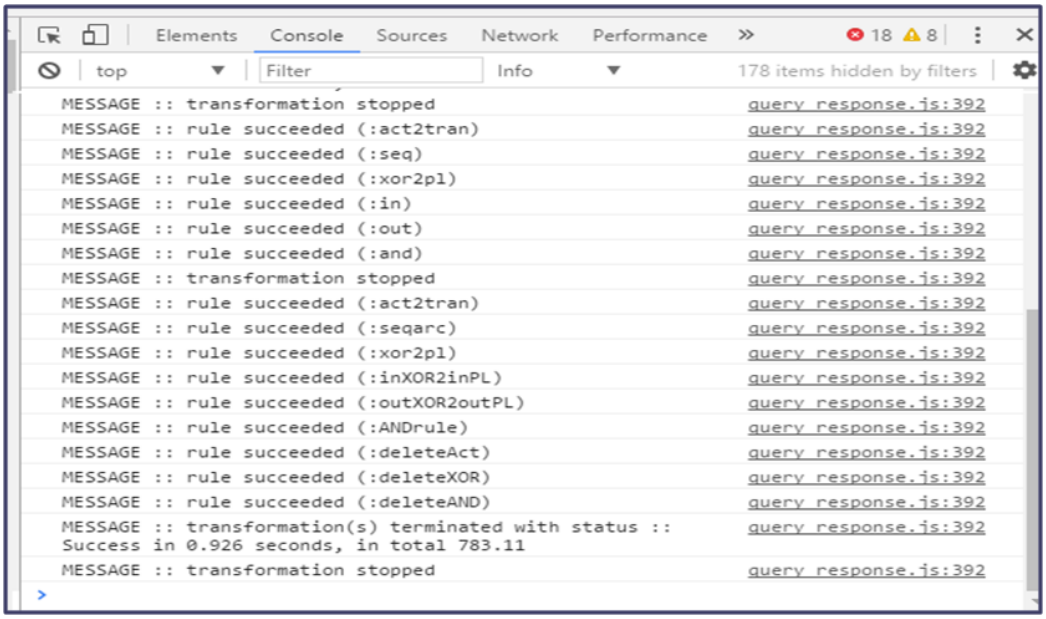


Figure 3.27 : Exécution de la transformation dans l'outil AToMPM

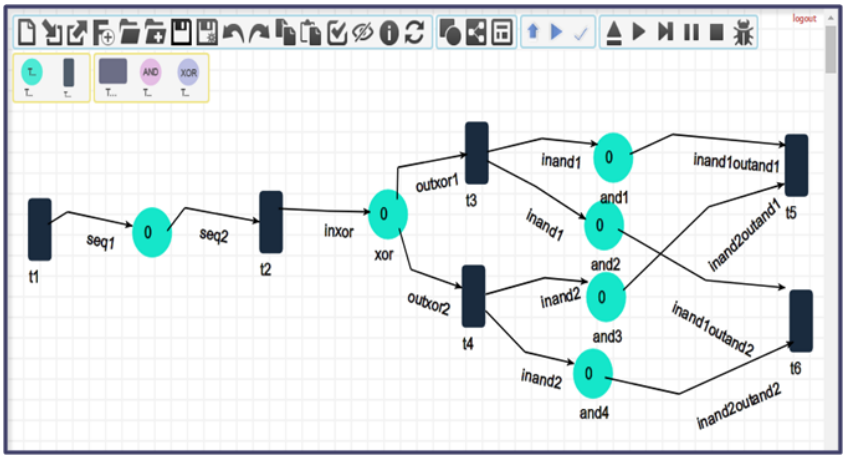


Figure 3.28 : Le réseau de Petri généré par la transformation

5. Conclusion

Dans ce chapitre, nous avons présenté les principes de transformations de graphes. Premièrement, nous avons introduit les concepts suivants : la notion de graphe, le système de transformation de graphes, les grammaires de graphes et les outils de transformation de graphes. Deuxièmement, nous avons présenté une transformation des modèles BPMN en réseaux de Petri à l'aide de l'outil AToMPM. Cette transformation a été développée dans les étapes suivantes :

1. Définir un méta-modèles pour les modèles BPMN,.
2. Définir un méta-modèles pour les réseaux de Pétri.
3. Définir une grammaire de graphes de la transformation des modèles BPMN en leur équivalent dans les réseaux de Petri.

Finalement, nous avons appliqué la transformation proposée sur un modèle BPMN.

VI Chapitre 04 : Méthodes Formelles et ses Applications

1. Introduction

Définition : Système critique

c'est un système dont lequel les ***erreurs*** peuvent conduire a une ***catastrophe humaine*** ou ***matérielle***.

Figure 4.1 : Exemple des Systèmes Critiques

Exemple : Explosion d'Ariane 5 : Bugs & Erreurs des logiciels

Figure 4.2 : La fusée Ariane 5 de l'Agence Spatiale Européenne

- L'explosion d'Ariane 5 après 37 s de son départ, le 4 juin 1996, qui a coûté un demi milliard de dollars.
- ***Cause*** : une ***faute logicielle*** d'une composante dont le fonctionnement n'était pas indispensable durant le vol.
- Il contenait une conversion d'un flottant sur 64 bits en un entier signé sur 16 bits.
- Pour Ariane 5, la valeur du flottant dépassait la valeur maximale pouvant être convertie.
  ⇒ Nécessité de « ***vérifier*** » certains logiciels/systèmes en utilisant les ***Méthodes Formelles***: Test, Démonstrateur de théorèmes, Model-checking.

Définition : Méthodes Formelles

Les méthodes formelles (MFs) sont des techniques basées sur des principes mathématiques pour décrire le comportement de systèmes et les propriétés à vérifier. Elles fournissent un cadre très rigoureux pour spécifier, développer et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc, afin de démontrer leur validité par rapport à une certaine spécification [20]*. Dans la littérature, il existe plusieurs classifications des méthodes formelles. Selon la référence [22]*, on peut distinguer les trois classes suivantes:

1) Algébriques :

- CCS (Milner:79)
- CSP (Hoar:85)
- ACP (Bergstra:85)
- LOTOS (Bolognesi et al : 87)
- Pi-calculus (Milner:92)
- RT-LOTOS (Courtiat et al :93)
- ET-LOTOS (Leduc et al : 93)
- D-LOTOS (Saidouni et al : 2003)

2) Langages :

- Estelle
- Maude (Logique de réécriture)
- Mobile Maude
- Real Time Maude

3) Réseaux de Petri :

- Places/Transitions
- Prédicat
- Colorés
- Algébriques
- Temporel
- Temporisé
- Temporisés stochastiques

2. Modèles Formels: Réseaux de Petri

2.1. Définition Formelle et Représentation des RdPs

Définition : Réseaux de Petri (RdP)

Les RdPs sont un outil utilisé pour la spécification et la vérification des systèmes. ils ont été introduits initialement par C.A PETRI (1962). Ils disposent d'une représentation graphique facilement compréhensible par les utilisateurs et également d'une fondation mathématique solide permettant l'application de techniques de vérification sur les systèmes modélisés avant leur déploiement.

Un réseau de Petri R est un quadruplet {P, T, Pre, Post}, avec :

- P = {p1, p2,..., pn} un ensemble de places (les états du système).
- T = {t1,..., tn}, un ensemble de transitions, avec P∩T = Ø.
- Pré: PXT -> N est la fonction d'incidence avant.
- Post : PXT -> N est la fonction d'incidence arrière.

Pre(pi, tj) représente le poids de l'arc allant de pi vers tj.

Post(pi, tj) représente le poids de l'arc allant de tj vers pi.

Complément : Représentation des Réseaux de Petri

Un RdP possède deux représentations : Graphique & Matricielle. La représentation graphique est un graphe biparti orienté et valu:

- Places: cercles
- Transitions: bars or rectangles
- Arcs: flèches étiquetées par des valeurs
- Jetons: points noirs (black dots)

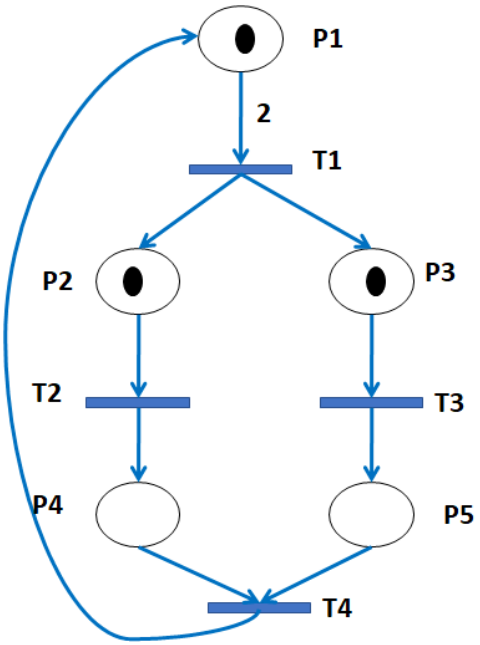


Figure 4.3 : Représentation graphique d'un Réseau de Petri

La matrice **pré:**

	T1	T2	T3	T4
P1	2	0	0	0
P2	0	1	0	0
P3	0	0	1	0
P4	0	0	0	1
P5	0	0	0	1

La matrice **post:**

	T1	T2	T3	T4
P1	0	0	0	1
P2	1	0	0	0
P3	1	0	0	0
P4	0	1	0	0
P5	0	0	1	0

Le marquage initial:

	M0
P1	1
P2	1
P3	1
P4	0
P5	0

Figure 4.4 : Représentation Matricielle d'un Réseau de Petri

Complément : Marquage

Un réseau de pétri marqué est définie par le couple {R, M}, où R est un réseau de pétri, et M : P-> N est une application appelé marquage qui associe à chaque place de R un nombre de jetons. Le marquage initial est noté m0 son rôle est de spécifier l'état initial du système. Par exemple, le marquage initial de RdP dans la figure 4.3 est : m0 (1, 1, 1 , 0, 0).

2.2. Franchissement des Transitions et Graphe de marquage

Définition : Franchissement d'une transition

- Pour une transition t, on dénote •t (resp. t•) l'ensemble de places d'entrée (resp. de sortie) de la transition t.
- Une transition est franchissable (“is enabled”) si et seulement s'il y a assez de jetons dans chaque place en entrée

  $\forall p \in {}^\bullet t : M(p) >= pré(p,t)$
- Notation M [t > : t est franchissable dans le marquage M
- Le franchissement d'une transition modifie le marquage en consommant des jetons dans les places d'entrée et en produisant des jetons dans les places de sortie.
- Si une transition t est sensibilisée pour un marquage M, t peut être franchie à partir de M, produisant un nouveau marquage M′, dénoté M −(t)-> M′, où

  $\forall p \in P, M'(p) = M(p) - pré(p, t) + post(p, t)$

  La figure 4.5 montre le franchissement de la transition t1.

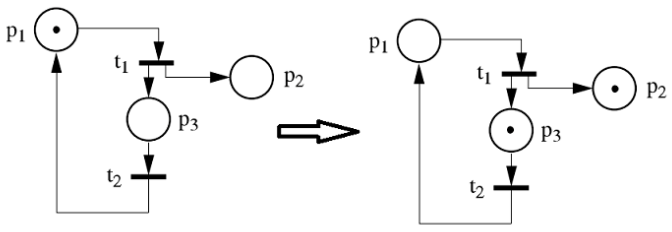


Figure 4.5 : Franchissement de la transition t1

Définition : Graphe de marquage (graphe d'accessibilité)

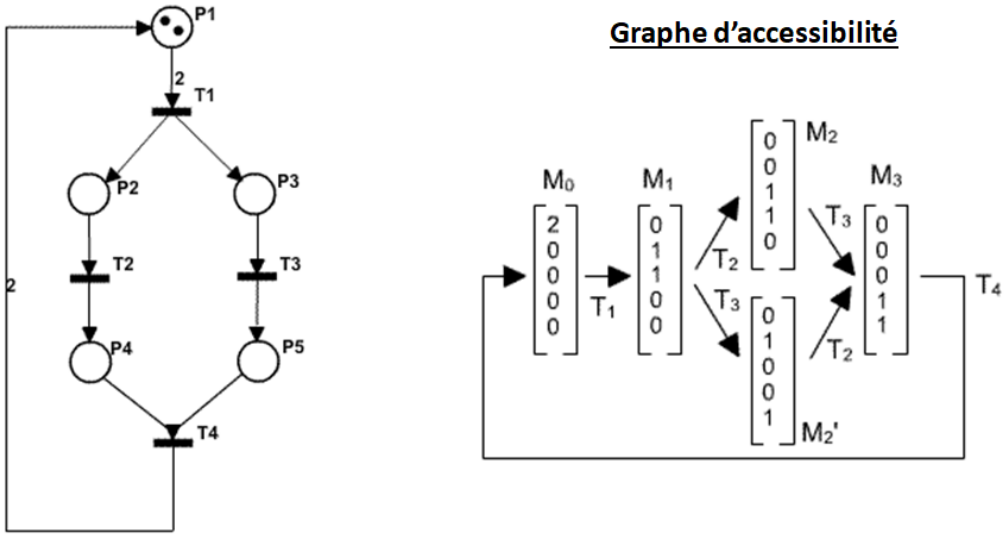


Figure 4.6 : Exemple d'un RdP et son graphe de marquage

- A(R,M0) est l'ensemble des marquages accessibles.
- Le graphe d'accessibilité GA(R,M0) de ce réseau à partir du marquage initial M0 est un graphe dirigé et étiqueté dont les sommets sont les éléments de A(R,M0).
- Un arc relie deux marquages Mi et Mj si et ssi il existe une transition t de R tel que: Mi −(t)-> Mj.
  La figure 4.6 montre Un RdP et son graphe de marquage.

2.3. Propriétés des Réseaux de Petri

Définition

- ***Accessibilité*** (reachability): un marquage m est accessible (à partir de m0) s'il existe une séquence de transitions t1,t2,.....,tn qui conduit de m0 à m.
  C à d :m0 [t1> m1 [t2> m2.....mn [tn> m
- ***Deadlock-free***: si chaque marquage accessible permet au moins une transition.
- Une transition t d'un réseau de Petri (R,M0) est ***quasi-vivante*** :
  ssi ∃ M accessible à partir de M0, tel que M [t >
- Une transition t d'un réseau de Petri (R,M0) est ***vivante***: ssi pour chaque marquage accessible M′, il existe un marquage M′′accessible à partir de M′ où t est sensibilisée.
- Un réseau de Petri (R,M0) est ***vivant*** (resp. quasi-vivant) si et ssi toutes ses transitions sont ***vivantes*** (resp. quasi-vivantes)

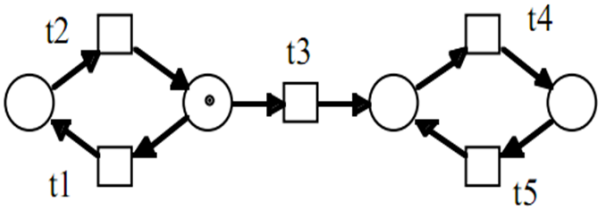


Figure 4.7 : Exemple d'un RdP qui est deadlock-free mais non vivant

Complément

- ***Borné*** : un RdP est dit K-borné si le nombre de jetons dans chaque place p est toujours inférieur ou égale à k pour chaque marquage accessible à partir de m0. La figure 4.8 montre un RdP non borné.
- ***Safe*** (1-borné): un Rdp est safe s'il est 1-borné.
- ***Réversible*** : si le marquage initial m0 est accessible depuis n'importe quel autre marquage.

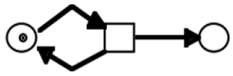


Figure 4.8 : Exemple d'un RdP non borné (unbounded)

3. Intégration des Méthodes Formelles avec l'IDM

Méthodes formelles (MFs)

- Les MFs permettent à la fois de modéliser le système et de vérifier les propriétés attendues.
- Les MFs reposent sur l'utilisation de langages formels dotés une sémantique mathématique rigoureuse.
- Principaux obstacles d'utilisation des MFs dans les activités de développement des systèmes sont liés à :
    - La difficulté réelle de manipuler les concepts théoriques et les méthodes d'analyse associées.
    - La difficulté pour les développeurs d'exprimer les propriétés du système d'une façon aisée.

Ingénierie Dirigée par les Modèles (IDM)

- La vérification des modèles comptent aujourd'hui parmi les enjeux les plus importants de l'IDM.
- L'IDM joue un rôle essentiel dans l'introduction des MFs dans les activités de développement des systèmes.
- Celle-ci repose sur:
    - La définition de langages formels par le biais de la méta-modélisation.
    - L'utilisation de transformations de modèles pour générer des modèles décrits dans ces langages formels.

Combinaison de l'IDM avec les MFs

	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 4.9 : Avantages et Inconvénients de l'IDM et des Méthodes Formelles

Chacune des deux approches a des points forts et des points faibles. Donc, les deux approches peuvent être combinées dans le sens où les inconvénients de l'un peuvent être surmontés grâce aux apports de l'autre [1]*.

4. Vérification d'une Transformation

4.1. Pourquoi la vérification d'une transformation ?

Problématique

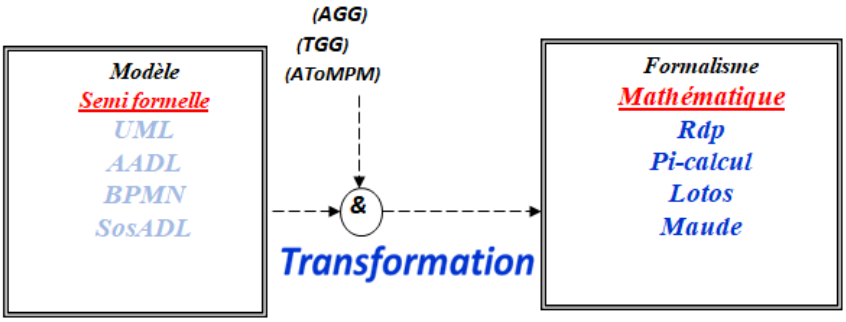


Figure 4.10 : Exemple d'approches de transformation de modèles

- Plusieurs outils de la transformation sont développés : TGG, AGG, AToMPM, . . . etc.
- Plusieurs transformations sont réalisées notamment des modèles semi-formelles (UML, BPMN, AADL) vers des modèles formels.
- Mais ces ***transformations*** et leurs ***outils*** souffrent de ***manque de vérification***
- ***Propriétés*** à Vérifier:
    - Terminaison de la transformation
    - Préservation de la sémantique du modèle source
    - Confluence
    - Invariants
    - Complétude
    - Absence d'interblocage & boucle infinie ...
- Donc, il est intéressant d'appeler à de nouvelles méthodes, techniques et outils pour le développement des transformations de modèles.

4.2. Approche de trois dimensions

Définition

Cette approche[12]* est composée de trois dimensions qui sont présentées dans la Figure 4.11. Ces dimensions sont : la transformation elle-même, le type de propriété à vérifier et la technique de vérification à utiliser pour modéliser la transformation, spécifier les propriétés attendues et vérifier si la transformation satisfait ses propriétés attendus.

- ***Transformation*** : la notion de transformation de modèles constitue l'élément central de l'ingénierie diriges par les modelés. Un modèle qui est conformé à un méta modèle source est transformé dans un autre modèle qui est conforme à un méta modèle cible en utilisant un ensemble de règles de transformation.
- ***Propriétés à vérifier*** : les propriétés sont les propriétés fonctionnelles de la transformation elle-même et les propriétés liées aux modelés source et cible.

- ***Techniques de vérification*** : les techniques utilisées pour vérifier formellement la correction d'une transformation sont le ***Model checker*** et les ***démonstrateurs de théorèmes***.

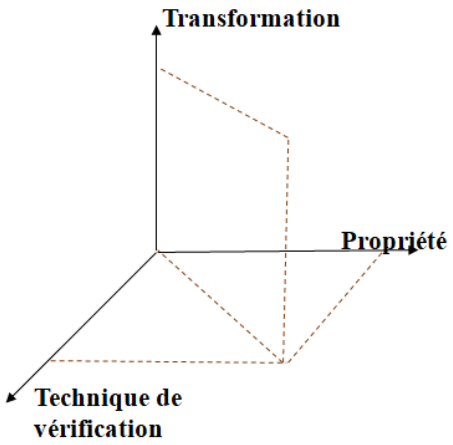


Figure 4.11 : Approche de trois dimensions [Amrani et al. 2015]

Complément : Propriétés à vérifier

Il existe deux classes de propriétés [12]*:

1. Propriétés fonctionnelles de la transformation elle-même, il existe 2 types:
   1.1) Terminaison:
   - Assurer que la transformation termine après un nombre des étapes finie.
   - Assurer l'existence d'un modèle cible.

   1.2)Confluence:
   - l'ordre d'application des règles de transformation n'est pas important.
   - Assurer que la transformation toujours produire le même résultat.
2. Propriétés liées aux modelés source et cible, il existe trois types:
   2.1) Conformité:
   - Un modèle est valide s'il est conforme à son méta modèle.
   - Est vérifié automatiquement dans des Framework de modélisation.

   2.2) Relation syntaxique :
   - Certains éléments(source)seront transformé en d'autres éléments (cible)

   2.3) Relation sémantique :
   - Liée la signification des deux modèles (source et cible)
   - Construire les LTS de la sémantique des deux modèles.

Méthode : Model Checker

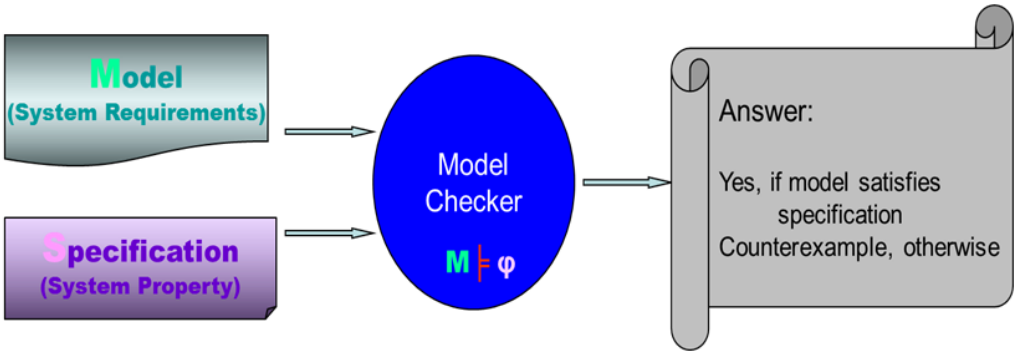


Figure 4.12 : Principe du Model Checker

Le principe de fonctionnement du modèle Checker consiste en trois phases. La première phase est la modélisation du comportement du système. La deuxième phase est la spécification des propriétés attendues du système dans la logique temporel LTL ou CTL. Finalement, il répond avec oui si la propriété est satisfaite sinon un contre-exemple est généré automatiquement qui montre un scénario possible d'erreur. Cependant,modèle Checker souffre de problème de l'explosion combinatoire de graphe d'états.

Méthode : Démonstrateur de théorèmes

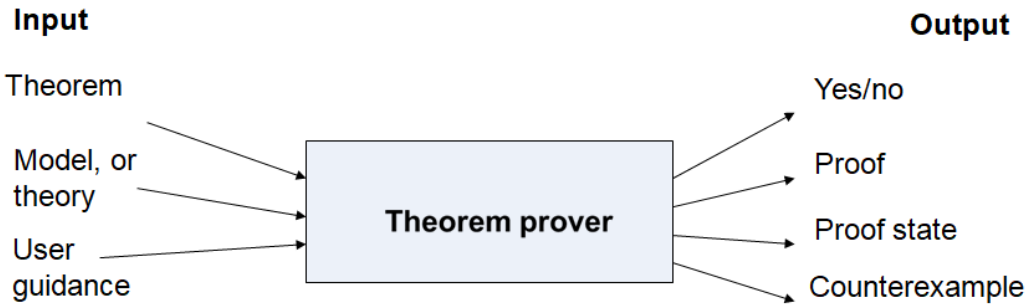


Figure 4.14 : Principe de démonstrateurs de théorèmes

Dans le cas du démonstrateur de théorèmes, le système et les propriétés attendues sont exprimés en utilisant des descriptions dans une logique mathématique par exemple Coq utilise le langage Galina. La preuve d'une propriété consiste en une ou plusieurs étapes et chaque étape peut faire appel aux axiomes, aux règles, aux définitions ou aux lemmes qui ont été déjà prouvés. L'avantage est qu'ils peuvent spécifier et effectuer des preuves sur des systèmes infinis à travers des techniques comme l'induction. Cependant,les démonstrateurs de théorèmes sont très coûteux, nécessitent des compétences avancées en matière de preuves ainsi que parfois de l'interaction de l'utilisateur pour la construction d'une preuve.

4.3. Approches de Vérification formelle de transformations de modèles

Méthode : Vérification d'une transformation basée sur le Model checker

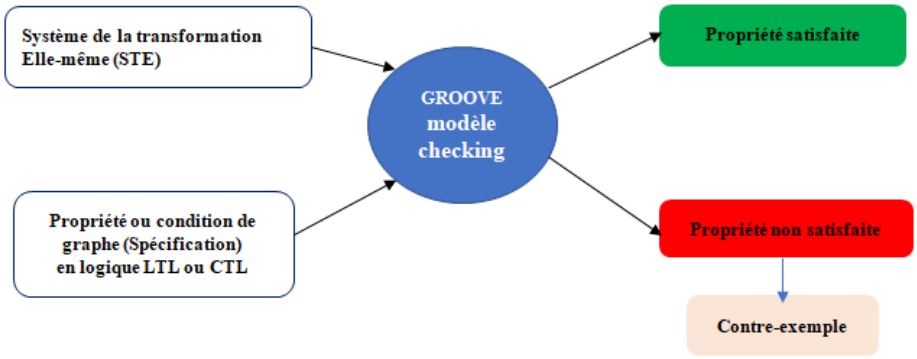


Figure 4.13 : Approche GROOVE modèle-checking

Cette approche [19]* permet de vérifier les transformations de graphes en utilisant l'outil GROOVE et son modèle checking. GROOVE considère la transformation comme un Système de Transition étiquetée (STE). Cette approche consiste en trois parties:

1. GROOVE génère le STE de la transformation contenant un ensemble des états et des transitions.
    - Chaque transition connecte deux états et est étiquetée avec le nom de la règle appliquée.
    - Chaque état contient les propriétés ou les conditions de graphes qui sont valides.
2. L'utilisateur exprime les conditions à vérifier avec la logique LTL ou CTL.
3. Modèle checking répond avec oui si le système de la transformation (STE) satisfait la propriété (la spécification) sinon il répond avec non et génère un contre-exemple.

Méthode : Vérification d'une transformation basée sur le démonstrateur de théorèmes

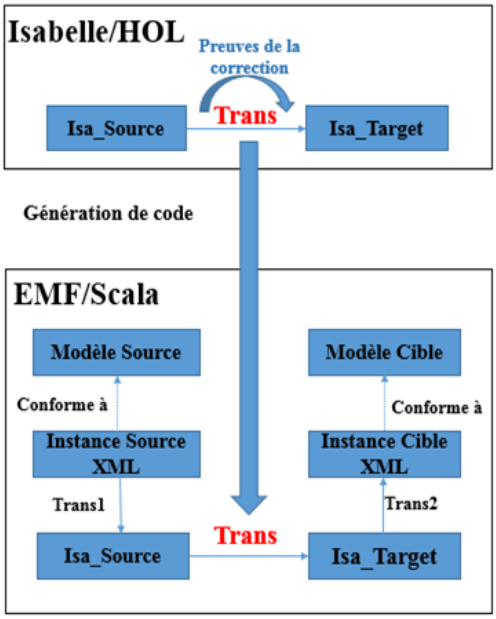


Figure 4.15 :Approche basée sur le démonstrateur de théorèmes

Cette approche [21]* considère la transformation comme une fonction mathématique qui compose de plusieurs fonction récursives. Elle consiste en deux parties:

- En Isabelle/HOL:

1. Description de modèle source
2. Description de modèle cible
3. Description de la transformation
4. Spécification des propriétés attendues
5. Preuve de correction des propriétés attendues

- En EMF/Scala:

1. Définition des méta-modèles en utilisant EMF.
2. Génération de code Scala de la fonction Trans.
3. Définition des fonctions Trans1 et Trans2.
4. Composition des fonctions :Trans2 (Trans (Trans1(modèle source)))-->modèle cible.

5. Conclusion

Dans ce chapitre, nous avons concentré sur les méthodes formelles et leur application pour le développement de logiciels sûrs. Nous avons présenté les concepts des réseaux de Petri : la définition formelle, les représentations des RdPs, le franchissement des transitions, le graphe de marquage et les propriétés des réseaux de Petri telles que la vivacité et la bornitude. Ensuite, nous avons abordé l'intégration des méthodes formelles avec l'IDM. Enfin, nous avons présenté un aperçu sur la vérification et la validation des transformations de modèles à l'aide des techniques de vérification formelles : le Model Checker et les démonstrateurs de théorèmes.

VII Chapitre 05 : Introduction à OCL

1. Introduction

Définition : Object Constraint Language (OCL)

- OCL est un langage formel pour écrire des expressions de manière précise.
- Il est basé sur la logique des prédicats du premier ordre.
- Les contraintes OCL ont une sémantique formelle, par conséquent, peuvent être utilisées pour réduire l'ambiguïté dans les modèles UML.
  Exemple: L'age d'une personne doit être supérieur ou égal à 18 ans.
- Prend en charge les concepts d'objets.
- Mais - OCL n'est pas un langage de programmation:
  - Aucun flux de contrôle.
  - Pas d'effets secondaires.
- Pourquoi OCL? Parce qu'UML ne suffit pas!

Exemple : UML ne suffit pas!

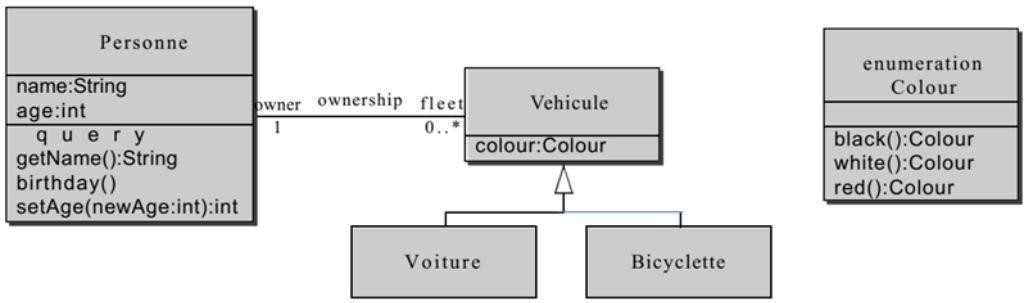


Figure 5.1 : Diagramme de classe d'application de gestion de véhicules

Contraintes !

- Nombre possible de propriétaires qu'une voiture peut avoir
- Âge requis des propriétaires de voitures
- Exigence qu'une personne puisse posséder au plus une voiture noire

Manque de précision: le diagramme de classe ne permet pas d'exprimer ces contraintes.

Attention

- Alors !
  - Nous avons besoin d'un langage pour aider avec la spécification.
  - Nous recherchons un «add-on» au lieu d'un tout nouveau langage avec des capacités de spécifications complètes.
  - Pourquoi pas la logique du premier ordre? - Pas OO.
  - OCL est utilisé pour spécifier les contraintes sur les systèmes OO.
  - OCL n'est pas le seul.
  - Mais OCL est le seul à être standardisé.

Fondamental : Objectifs du langage OCL

- Spécifier des invariants pour les classes et les types.
- Spécifier les conditions préalables et postérieures aux méthodes.
- Comme langage de navigation.
- Pour spécifier des contraintes sur les opérations.
- Tester les Exigences et les spécifications.

2. Typologie des contraintes

Définition : Notion de contexte

Mot-clé ***context*** : une contrainte OCL est toujours définie dans un contexte. Ce contexte est l'instance d'une classe.

Exemple :

- ***context Personne***:
  La contrainte OCL s'applique à la classe Personne, c'est-à-dire à toutes les instances de cette classe.
- ***context Personne::setAge(newAge:int):int***:
  La contrainte OCL s'applique à la méthode setAge(newAge:int).

Définition : Invariants

Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence. Un invariant est une expression OCL booléenne - prend la valeur true / false. Le ***Mot-clé*** d'un invariant est : ***inv***.

Exemple

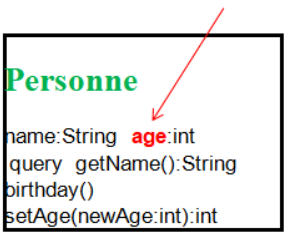


Figure 5.2 : La classe Personne

context Personne

inv: age >= 18

Contrainte :"Pour toutes les instances de la classe Personne (dans la figure 5.2), l'age doit toujours être supérieur ou égal à 18 ans".

Exemple

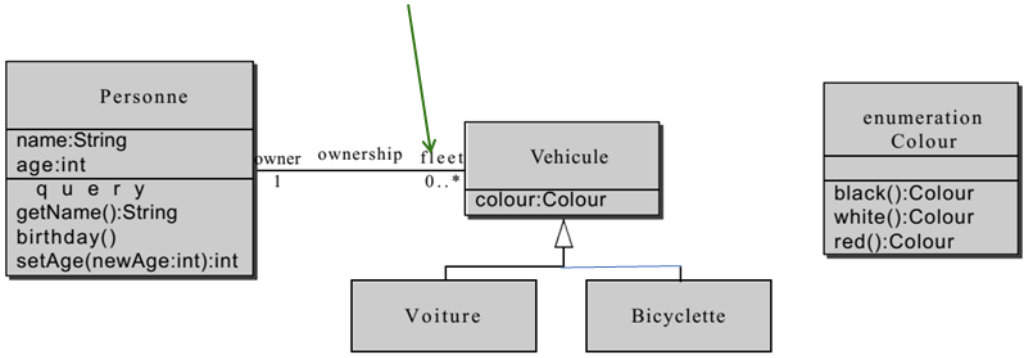


Figure 5.3 : Le rôle fleet : ensemble des véhicules

Les objets de contexte peuvent être désignés dans l'expression à l'aide du mot-clé «***self***».

Context Personne

inv: self.fleet->size <= 5

Contrainte :" Personne ne possède plus de 5 véhicules". La figure 5.3 montre le rôle ***fleet***.

Définition : Pré et Post conditions

OCL peut également spécifier des opérations à l'aide des Pré et Postconditions :

- Précondition : état qui doit être respecté avant l'appel de l'opération.
- Postcondition : état qui doit être respecté après l'appel de l'opération.
- Mots-clés : ***pre*** et ***post***.

Dans la postcondition, deux éléments particuliers sont utilisables :

- Attribut ***result*** : référence la valeur retournée par l'opération.
- mon_attribut***@pre*** : référence la valeur de mon_attribut avant l'exécution de l'opération.

Syntaxe pour préciser l'opération : ***context*** ma_classe::mon_op(liste_param) : type_retour.

Exemple : Pré et Postconditions sur l' opération setAge(...) dans la figure 5.4.

Context Personne::setAge(newAge:int)

pre: newAge >= 0

Post: self.age = newAge

Contrainte : "Si setAge (...) est appelée avec un argument non négatif, l'argument devient la nouvelle valeur de l'attribut age".

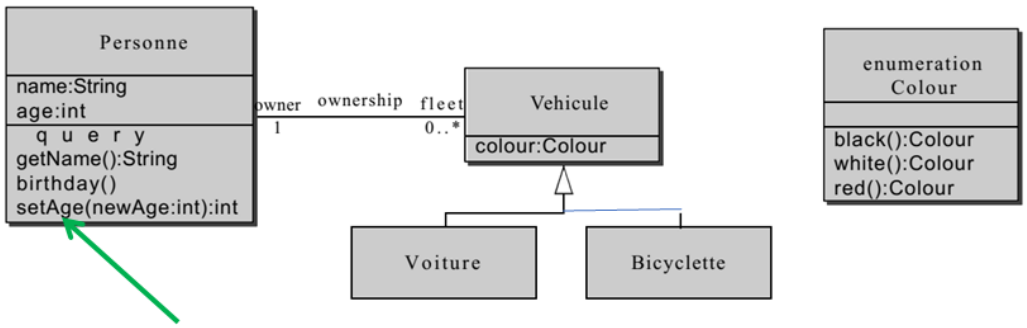


Figure 5.4 : L'opération setAge(...) de la classe personne

Remarque : Pré et Postconditions

On ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution.

Définition : Conception par Contrat

Pre et Post conditions permettent de définir une ***conception par contrat*** entre l'appelant d'une opération et l'appelé:

- Si l'appelant respecte les contraintes de la pré-condition alors l'appelé s'engage à respecter la post-condition.
- Si l'appelant ne respecte pas la pré-condition, alors le résultat de l'appel est ***indéfini***.

Exemple

Context Person::setAge(newAge:int)

pre: newAge >= 0

Post: self.age = newAge

- Si l'appelant à ***setAge*** respecte la pré-condition ***newAge >=0***, alors setAge affectera à l'attribut ***age*** la valeur du parametre ***newAge***. Sinon, la valeur de l'attribut age sera ***indéfinie***.

3. Types de base et opérations

Attention

Pour être bien formée, une expression OCL doit être conforme aux règles de conformité de type. Par exemple, vous ne pouvez pas comparer un entier avec une chaîne de caractères. En plus, chaque classificateur dans un modèle UML devient un type OCL.

Définition : Types de base OCL

Les types et opérateurs prédéfinis dans les contraintes OCL [15]*:

- ***Boolean*** -> true, false
  Opérations: and, or, xor, not, implies, if-then-else
- ***Integer*** -> 1, -5, 2, 489, 26524, ...
  Opérations : *, +, -, /, abs()
- ***Real*** -> 1.5, 3.14, ...

Opérations : *, +, -, /, floor()

- ***String*** -> 'Université Mila...'
  Opérations : toUpper(), concat()

Complément : types de collection OCL

Les ensembles, Multi-Ensembles (Bags) et séquences sont prédéfinis dans OCL et sont des sous-types: ***collection***. OCL propose un grand nombre d'opérations prédéfinies sur les collections. Ils sont tous de la forme: ***collection-> opération (arguments)***. La collection ***OCL-Type*** est la superclasse générique d'une collection d'objets de type T. Il existe quatre types de collections [17]*:

- ***Set***: Défini au sens mathématique. Chaque élément ne peut apparaître qu'une seule fois.
- ***OrderedSet*** : idem mais avec ordre (les éléments ont une position dans l'ensemble).
- ***Bag***: une collection, dans laquelle les éléments peuvent apparaître plusieurs fois (multi-Set).
- ***Séquence***: un multi-ensemble, dans lequel les éléments sont ordonnés.
- Exemples :
  – { 1, 4, 3, 5 } : Set(integer)
  – { 1, 2, 3, 5 } : OrderedSet(integer)
  – { 1, 4, 1, 3, 5, 4 } : Bag(integer)
  – { 1, 1, 3, 4, 4, 5 } : Sequence(integer)

4. Opérations pour les collections OCL

Définition

Les collections sont tous de la forme: collection-> opération (arguments). Il existe plusieurs opérations sur les collections :

- size: Integer
  Retourne le nombre d'éléments dans la collection.
- includes(o:OclAny): Boolean
  Retourne True, si l'élément o est dans la collection.
- count(o:OclAny): Integer
  Compte le nombre de fois qu'un élément est contenu dans la collection.
- isEmpty: Boolean
  Retourne True, si la collection est vide
- notEmpty: Boolean
  Retourne True, si la collection n'est pas vide.
- union(c1:Collection) Retourne l'union avec la collection c1.
- intersection(c2:Collection)
  Retourne l'intersection avec Collection c2 (contient uniquement des éléments, qui apparaissent dans la collection ainsi que dans la collection c2).
- including(o:OclAny)
  Collection contenant tous les éléments de la collection et l'élément o.
- select(expr:OclExpression) Sous-ensemble de tous les éléments de la collection, pour lesquels l'expression OCL expr est vraie.

Méthode : Comment obtenir les collections OCL?

- Une collection peut être générée en énumérant explicitement les éléments.
- Une collection peut être générée en naviguant le long d'une ou plusieurs associations 1:N
    - La navigation le long d'une seule association 1: n donne un ensemble (set)
    - La navigation le long de quelques associations 1: n donne un Bag (multi-ensemble)
    - La navigation le long d'une seule association 1: n avec la contrainte {ordonnée} donne une séquence

5. Accès aux objets et navigation

Fondamental

Dans une contrainte OCL associée à un objet, on peut:

- ***Accéder*** à l'état interne de cet objet (ses attributs)
- ***Accéder*** aux opérations de cet objet : self.operation.
- ***Naviguer*** dans le diagramme : accéder de manière ***transitive*** à tous les objets avec qui il est en relation.

Exemple : Navigation Locale et Directe

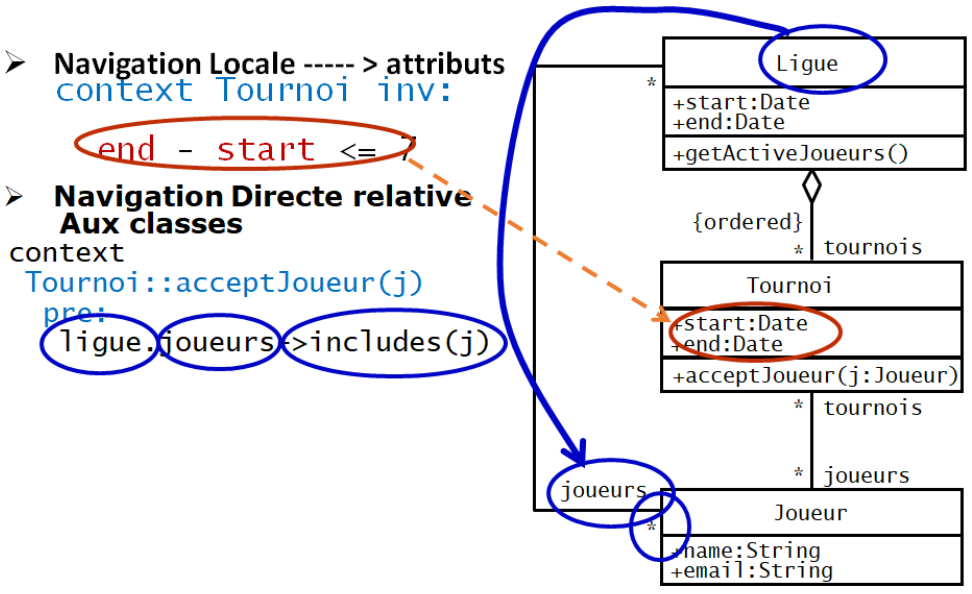


Figure 5.5 : Exemple d'accès aux objets et navigation

- Navigation Locale ----- > attributs

  context Tournoi

  inv: end - start <= 7

  Contrainte :"la durée d'un tournoi doit être inférieur à 7"

- Navigation Directe relative Aux classes

  context Tournoi::acceptJoueur(j)

  pre: ligue.joueurs->includes(j)

  Contrainte : "Si acceptJoueur(j) est appelée avec un argument j, ce joueur (j) doit appartenir à la liste des joueurs de la ligue."

Exemple : Navigation à travers une association : 1..n

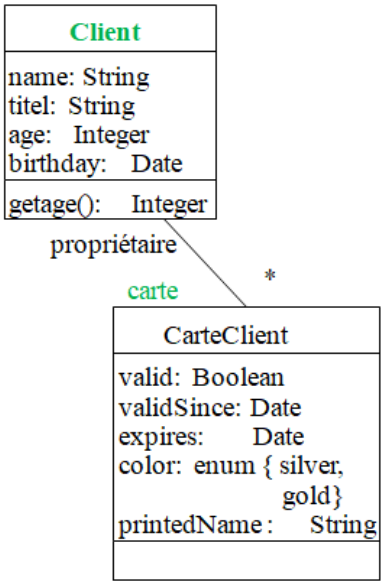


Figure 5.6 : Exemple de navigation : association en cardinalité 1..n

Contrainte :"un client ne doit pas avoir plus de 3 cartes".

context Client

inv: carte->size <= 3

carte désigne un ensemble de cartes de clients.

Exemple : Navigation à travers plusieurs associations : 1..n

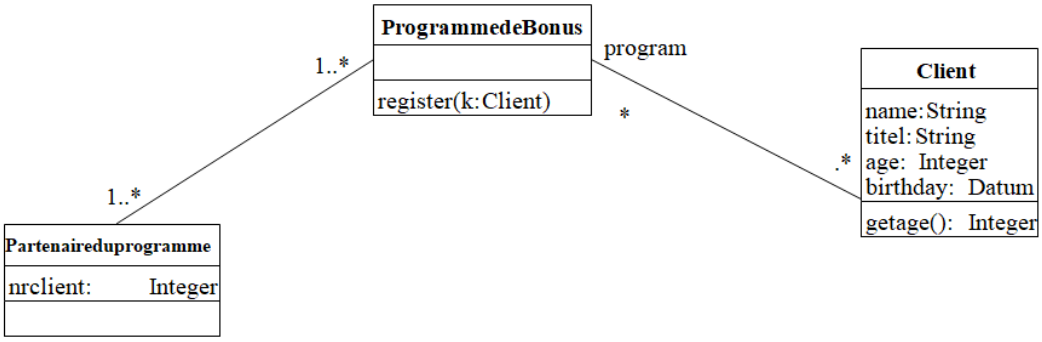


Figure 5.7 : Exemple de navigation : plusieurs associations en cardinalité 1..n

Context Partenaireduprogramme

inv: nrclient = programmedebonus.client->size

Client denote un multi-ensemble de client.

programmedebonus denote un ensemble de programmedebonus.

Exemple : Navigation à travers une association avec contrainte

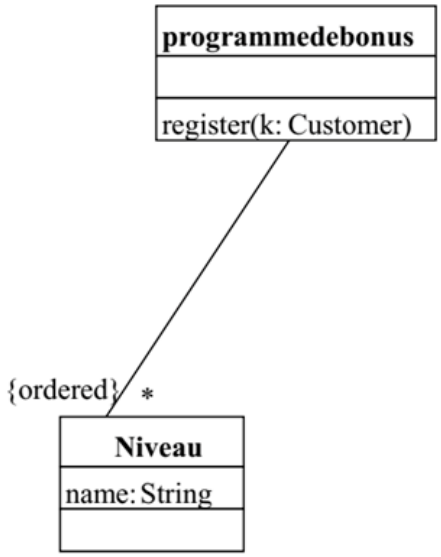


Figure 5.8 : Exemple de navigation : association avec contrainte

La navigation à travers une association avec la contrainte {ordered} produit une séquence.

context programmedebonus

inv: niveau->size = 2

niveau dénote une séquence de niveaux.

6. Conclusion

OCL est un Langage de contraintes orienté-objet. C'est un Langage formel (mais «simple» à utiliser) avec une syntaxe, une grammaire, une sémantique et s'applique entre autres sur les diagrammes UML [14]*. Dans ce chapitre, nous avons abordé les concepts de base du langage OCL : la typologie des contraintes, les types de base, les opérations sur les collections et l'accès aux objet et navigation. Nous avons montré l'efficacité du langage OCL à travers plusieurs exemples.

VIII Exercice

Exercice : 01

Citer les qualités attendues d'un logiciel ?

- ☐ Fiabilité
- ☐ Interopérabilité
- ☐ Généralisation
- ☐ Performance
- ☐ Portabilité
- ☐ Réutilisabilité
- ☐ Parallélisme
- ☐ Reconfigurable

Exercice : 02

Citer les étapes à suivre pour la production d'un logiciel ?

- ☐ Livraison
- ☐ Spécification
- ☐ Génération du code
- ☐ Implémentation
- ☐ Tests
- ☐ Fabrication
- ☐ Conception
- ☐ Analyse de besoins
- ☐ Maintenance
- ☐ Facturation

Exercice : 03

Citez les méthodes agiles ?

- ☐ XP
- ☐ UP
- ☐ DSDM
- ☐ RP
- ☐ ASD
- ☐ CCM

Exercice : 04

Quels sont les principaux concepts de l'IDM ?

- ☐ Modèle
- ☐ Langage
- ☐ Transformation de modèles
- ☐ Transformation de graphes
- ☐ Génération du code
- ☐ Méta-modèle
- ☐ Méta-niveau

Exercice : 05

Citer les niveaux de modélisation dans l'IDM ?

- ☐ Monde virtuel
- ☐ UML
- ☐ Méta-modèle
- ☐ Méta-méta-modèle
- ☐ Syntaxe concrète
- ☐ Monde réel
- ☐ Modèle

Exercice : 06

Citez les domaines d'utilisation des grammaires de graphes ?

- ☐ Exprimer la sémantique opérationnelle
- ☐ Vérifier la correction d'un modèle
- ☐ Transformer des modèles en modèles comportementaux équivalents
- ☐ Modélisation des systèmes complexes
- ☐ Optimiser les modèles
- ☐ Générer du code pour un outil particulier

Exercice : 07

Un réseau de Petri :

- ☐ Est basé sur une fondation mathématique solide.
- ☐ Est une algèbre de processus utilisé dans la modélisation des systèmes concurrents.
- ☐ Est vivant si toutes ses transitions sont vivantes.
- ☐

Est dit K-borné si le nombre de jetons dans chaque place p est toujours supérieur ou égale à k pour chaque marquage accessible à partir de m0.

- ☐ Possède deux représentations : Graphique & Matricielle.

Exercice : 08

Le model-checking est une technique de vérification formelle basé sur les modèles, tel que le modèle source est écrit dans un formalisme de modélisation et les propriétés sont décrites par une logique temporelle (LTL,CTL,..).

- ○ Oui
- ○ Non

Exercice : 09

Citer les propriétés utilisées pour vérifier la correction des transformations de modèles ?

- ☐ Préservation de la sémantique du modèle source
- ☐ Bornitude
- ☐ Terminaison de la transformation
- ☐ Vivante
- ☐ Confluence

- ☐ Complétude
- ☐ Absence d'interblocage & boucle infinie

Exercice : 10

OCL :

- ☐ Est un langage formel
- ☐ Réduire l'ambiguïté dans les modèles UML
- ☐ Est basé sur la logique de réécriture
- ☐ Est un langage de programmation
- ☐ Prend en charge les concepts d'objets.

Exercice : 11

Citer les approches du développement de logiciels sûr qui combinent des méthodes formelles avec semi-formelles.

- ☐ UML/Java
- ☐ UML/ COQ démonstrateur de théorèmes
- ☐ UML/OCL
- ☐ BPMN/ Réseaux de Petri
- ☐ AADL/ Event-B
- ☐ BPMN/Maude (logique de réécriture)
- ☐ UML/Algébre de processus :Pi-calculus
- ☐ AADL/C++

Références

1
Kerkouche Elhillali, Modélisation multi-paradigme, Thèse de Doctorat en sciences , Université Mentouri Constantine, 2012.

11
Makhloufi & Benkara, Mémoire de Master RSD en Informatique, Université de Constantine 2, 2017.

12
Amrani Moussa, et al. Formal verification techniques for model transformations: A tridimensional classification. The Journal of Object Technology 14.3 (2015).

13
Jensen, Kurt, and Lars M. Kristensen. Coloured Petri nets: modelling and validation of concurrent systems. Springer Science & Business Media, 2009.

14
The Object Constraint Language: Precise Modeling with UML, by Jos Warmer and Anneke Kleppe.

15 Marianne Huchard, "Object Constraint Language (OCL) Une introduction", 2007.

17 Cours “OCL”, Laetitia Matignon, Université Claude Bernard Lyon 1, 2012- 2013.

18
Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", Vol.1. World Scientific, 1999.

19 GROOVE home page, https://groove.ewi.utwente.nl/.

2 Cours « Génie logiciels », Pierre Gérard, Université de Paris 13, 20072008.

20
Jeannette M. Wing, "A Specifier's Introduction to Formal Methods", Computer, vol. 23(9):8-23, 1990.

21
Meghzili Said, et al. "Verification of Model Transformations Using Isabelle/HOL and Scala." Information Systems Frontiers 21.1 (2019): 45-65.

22
Cours “ Méthodes formelles pour le parallélisme”, Saidouni Djamel Eddine, Université de Constantine 2.

24
Da Silva, Alberto Rodrigues. "Model-driven engineering: A survey supported by the unified conceptual model." Computer Languages, Systems & Structures 43 (2015): 139-155.

3 Davide Buscaldi, OMG's Model Driven Architecture.

4 Cours, «Ingénierie des modèles », Alain Cariou, Université de Pau, France, 2020/2021.

5
Cours «Ingénierie Dirigée par des les Modèles », Alloua Chaoui, Université de Constantine 2, 2020/2021.

6 Livre de Xavier Blanc , MDA en action, Editions Eyrolles

7 Oscar Nierstrasz, OMG's Model Driven Architecture.

8 AToMPM home page : https://atompm.github.io/.

9
Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jorg Kreowski, Sabine Kuske, Detlef Pump, Andy Schürr and Gabriele Taentzer, "Graph transformation for specification and programming", Science of Computer programming, vol 34, NO°1, pages 1-54, Avril 1999.