

## **CHAPITRE 3**

### **LES LANGAGES DE MODELISATION**

#### **3.1 Introduction**

Ayant à notre disposition les méta-modèles de diagramme de classe et de réseau de pétri, nous allons définir, dans ce chapitre, un chemin de migration horizontal entre ces deux différentes technologies, par la définition d'une transformation qui permet de passer de modèle statique et abstrait(diagramme de classe) à un modèle exécutable et dynamique (réseau de pétri).

Nous allons utiliser un langage bien défini pour exprimer l'ensemble des règles qui constituent la définition de la transformation. La définition des règles de transformation entre modèles se fera au niveau des méta-modèles

#### **3.2 UML (Unified Modeling Language)**

##### **3.2.1 Définition**

C'est le langage de modélisation le plus largement utilisé. On est déjà rendu à la version 2.0. Son méta-modèle est écrit dans MOF et par abus de langage on peut dire que ce méta-modèle d'UML est instance de MOF.

Le rôle d'UML est essentiel comme langage de modélisation normalisé d'OMG. UML prend aussi avantage de MOF pour l'écriture des définitions de transformations.

Notons ici que ce langage est adéquat pour l'expression de la structure d'un système, mais souffre d'une insuffisance vis à vis de la modélisation d'un comportement c'est à dire des aspects dynamiques. En effet les diagrammes de machines à états finis de l'UML deviennent très compliqués quand on s'en sert pour décrire les opérations induites par un système relativement simple.

UML n'est pas une méthode ou un processus, est un langage de modélisation objet, UML a été adopté par toutes les méthodes Objet. [20]

### 3.2.2 La modélisation UML

Le méta-modèle UML fournit une panoplie d'outils permettant de représenter l'ensemble des éléments du monde objet (classes, objets, ...) ainsi que les liens qui les relie. Toutefois, étant donné qu'une seule représentation est trop subjective, UML fournit un moyen astucieux permettant de représenter diverses projections d'une même représentation grâce aux *vues*. Une vue est constitué d'un ou plusieurs *diagrammes*.

On distingue deux types de vues :

- **Les vues statiques** : c'est-à-dire représentant le système physiquement
  - diagrammes d'objets.
  - diagrammes de classes.
  - diagrammes de cas d'utilisation.
  - diagrammes de composants.
  - diagrammes de déploiement.
- **Les vues dynamiques** : montrant le fonctionnement du système
  - diagrammes de séquence.
  - diagrammes de collaboration.
  - diagrammes d'états-transitions.
  - diagrammes d'activités.

### 3.2.3 Diagramme de classe

Le diagramme de classes constitue un élément très important de la modélisation : il permet de définir quelles seront les composantes du système final : il ne permet en revanche pas de définir le nombre et l'état des instances individuelles. Néanmoins, on constate souvent qu'un diagramme de classes proprement réalisé permet de structurer le travail de développement de manière très efficace, il permet aussi, dans le cas de travaux réalisés en groupe (ce qui est pratiquement toujours le cas dans les milieux industriels), de séparer les composantes de manière à pouvoir répartir le travail de développement entre les membres du groupe.

Enfin, il permet de construire le système de manière correcte. [21]

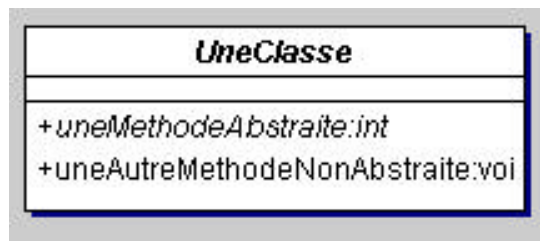
#### ➤ Les classes

La notion de classe est essentielle en programmation orientée objets : elle définit une abstraction, un type abstrait qui permettra plus tard d'instancier des objets. On distingue généralement entre classes abstraites (qui ne peuvent pas être instanciées) et classes "normales", qui servent à définir des objets.

✓ ***Classes abstraites***

Une classe abstraite ne peut donc pas être utilisée pour fabriquer des instances d'objets, elle sert uniquement de modèle, que l'on pourra utiliser pour créer des classes plus spécialisées par dérivation (héritage). Une classe abstraite est assez proche de la notion d'interface, d'ailleurs, la notion d'interface est généralement implémentée par une classe abstraite, dont toutes les méthodes sont purement virtuelles. [21]

Sa représentation en UML correspond à la figure 3.1

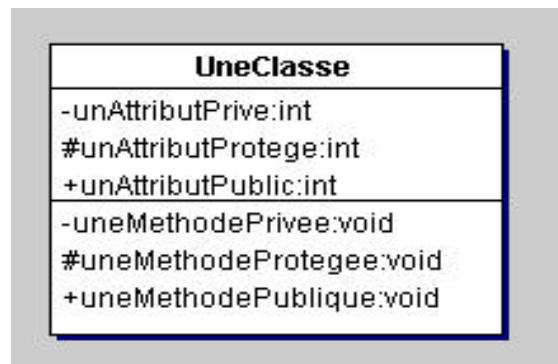


**Figure 3.1 : classe abstraite**

Le + dénote la publication du membre concerné. Une classe abstraite peut posséder des membres privés ou des attributs privés ou publics, d'autre part, les méthodes peuvent faire l'objet d'une implémentation, même si la méthode est purement virtuelle.

✓ ***Classes non abstraites***

Une classe "normale" ne contient pas de membres abstraits. Sa représentation en UML correspond à la figure 3.2.



**Figure 3.2 : Classe non abstraite**

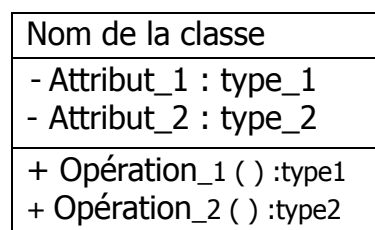
Noter les petits signes "cabalistiques" précédant l'identification des méthodes : [21]

- - dénote un membre privé
- + dénote un membre public
- # Dénote un membre protégé

#### ✓ *Représentation graphique*

Une classe est un classeur. Elle est représentée par un rectangle divisé en trois à cinq compartiments.

Le premier indique le nom de la classe, le deuxième ses attributs et le troisième ses opérations. Un compartiment des responsabilités peut être ajouté pour énumérer l'ensemble de tâches devant être assurées par la classe mais pour lesquelles on ne dispose pas encore assez d'informations. Un compartiment des exceptions peut également être ajouté pour énumérer les situations exceptionnelles devant être gérées par la classe.



**Figure 3.3 : Représentation UML d'une classe.**

✓ *Caractéristiques d'une classe*

- *Nom d'une classe*

Le nom de la classe doit évoquer le concept décrit par la classe. Il commence par une majuscule. Pour indiquer qu'une classe est abstraite, il faut ajouter le mot-clef **abstract**.

Il est écrit dans le rectangle du haut. Dans une classe classique, le nom est écrit en romain.

Le nom des classes abstraites est écrit en italique. Les classes Template ont, dans leur angle supérieur droit, un rectangle dont la bordure est en pointillé et qui contient les types des paramètres.

- *Les attributs de la classe*

Les attributs (on parle parfois de propriétés) définissent des informations qu'une classe ou un objet doivent connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut doit être unique dans la classe.

Le type de l'attribut peut être un nom de classe, un nom d'interface ou un type de donné prédéfini. La multiplicité d'un attribut précise le nombre de valeurs que l'attribut peut contenir. Lorsqu'une multiplicité supérieure à 1 est précisée, il est possible d'ajouter une contrainte pour préciser si les valeurs sont ordonnées ou non.

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe (*statique* en Java ou en C++) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut mais n'en possèdent pas une copie. Un attribut de classe n'est donc pas une propriété d'une instance mais une propriété de la classe et l'accès à cet attribut ne nécessite pas l'existence d'une instance. Graphiquement, un attribut de classe est souligné.

- *Les Méthodes (Opération) de la classe*

Dans une classe, une opération (même nom et même types de paramètres) doit être unique. Quand le nom d'une opération apparaît plusieurs fois avec des paramètres différents, on dit que l'opération est surchargée. En revanche, il est impossible que deux opérations ne se distinguent que par leur valeur retournée.

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres. Cette méthode n'a pas accès aux attributs de la classe. L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe. Graphiquement, une méthode de classe est soulignée.

✓ *Classe active :*

Une classe est passive par défaut, elle sauvegarde les données et offre des services aux autres. Une classe active initie et contrôle le flux d'activités. Graphiquement, une classe active est représentée comme une classe standard dont les lignes verticales du cadre, sur les côtés droit et gauche, sont doublées.

➤ **Relations entre classes**

✓ *Notion d'association :*

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées. Un attribut peut être considéré comme une association dégénérée dans laquelle une terminaison d'association est détenue par un classeur.

✓ *Association binaire :*

Une association binaire est matérialisée par un trait plein entre les classes associées. Elle peut être ornée d'un nom, avec éventuellement une précision du sens de lecture (► ou ◄).

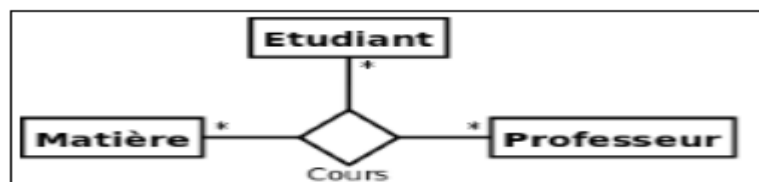
Quand les deux extrémités de l'association pointent vers la même classe, l'association est dite réflexive.



**Figure 3.4 : Exemple d'association binaire.**

✓ *Association n-aire :*

Une association n-aire lie plus de deux classes. On représente une association n-aire par un grand losange avec un chemin partant vers chaque classe participante. Le nom de l'association, le cas échéant, apparaît à proximité du losange.



**Figure 3.5 : exemple d'association n-aire**

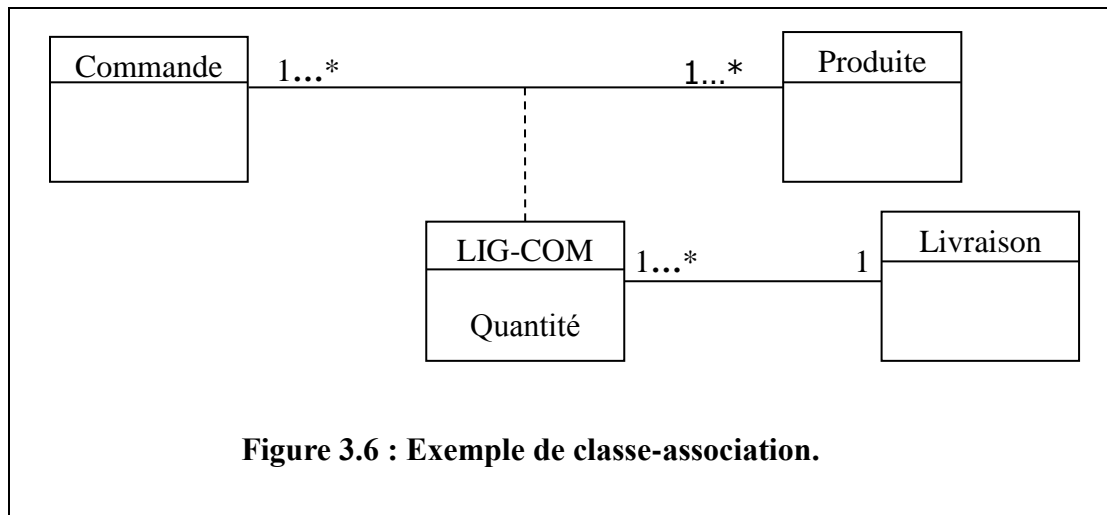
✓ *Multiplicité ou cardinalité :*

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

- 1...1 noté 1 : Un et un seul
- 0...1 : Zéro ou un
- 0...\* noté \* : De Zéro à n
- 1...\* : De un à n
- n...m : De n à m

✓ *Classe-association :*

Il peut arriver que l'on ait besoin de garder des informations (attributs ou opérations) propres à une association. Une classe de ce type est appelée classe association. Une classe association est une classe comme une autre qui peut entretenir des relations avec d'autres classes.

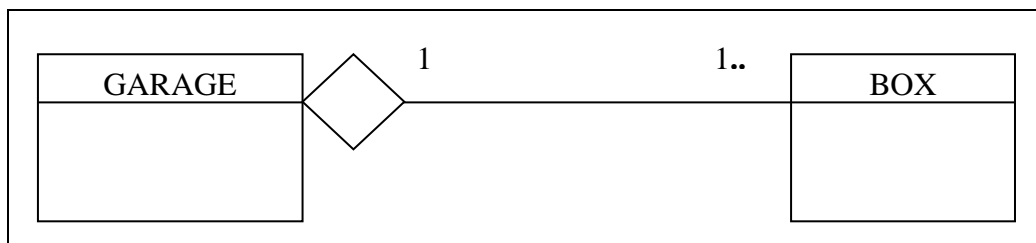


#### ✓ Agrégation :

Une agrégation est un type particulier d'association. Elle traduit la volonté de renforcer la dépendance entre les classes. C'est une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. Graphiquement, on ajoute un losange vide (◇) du côté de l'agregat. [23]

Les critères suivants impliquent une agrégation :

- une classe fait partie d'une autre classe,
- une action sur une classe implique une action sur une autre classe,
- les objets d'une classe sont subordonnés aux objets d'une autre classe.



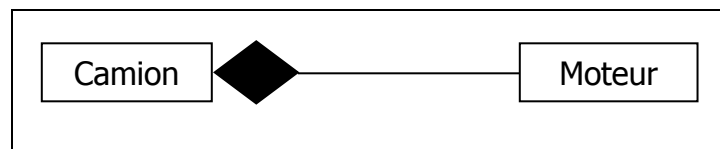
**Figure 3.7 : Exemple de relation d'agrégation.**



L'inverse n'est pas toujours vrai, l'agrégation n'implique pas nécessairement tous les critères ci-dessus. Un agrégat peut être multiple.

✓ **Composition :**

La composition est un cas particulier d'agrégation dans laquelle la vie des composants est liée à celle de l'agrégat. Dans la composition, l'agrégat ne peut être multiple. La composition se représente par un losange noir. [23]



**Figure 3.8 : Exemple de relation**

Une composition est une association contraignante : la suppression d'un objet agrégat entraîne la suppression des objets agrégés.

✓ **Généralisation et Héritage :**

La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). Dans le langage UML, la relation de généralisation se traduit par le concept de relation d'héritage. [22]

Les propriétés principales de l'héritage sont :

- La classe enfant possède toutes les caractéristiques de ses classes parents, mais elle ne peut accéder aux caractéristiques privées de cette dernière.
- Une classe enfant peut redéfinir une ou plusieurs méthodes de la classe parent.
- Toutes les associations de la classe parent s'appliquent aux classes dérivées.
- Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue.
- Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple.

**✓ Dépendance :**

Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre des éléments du modèle. Elle est représentée par un trait discontinu orienté. Elle indique que la modification de la cible peut impliquer une modification de la source.

**3.2.4 Le langage OCL (Object Constraint Language)**

Forme d'instruction dans un langage textuel qui peut être naturel ou formel. En général, une contrainte peut être attachée à n'importe quel élément de modèle ou liste d'éléments de modèle.

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes que l'on souhaiterait. L'OMG a défini formellement le langage textuel de contraintes OCL (Object Constraint Language), qui permet de définir n'importe quelle contrainte sur des modèles UML.

L'objectif du langage OCL est aussi, et surtout, d'être indépendant de toute plate- forme d'exécution. Une fois les contraintes OCL spécifiées sur un modèle UML, il est envisageable de générer une application conforme en Java, PHP ou .Net. Pour que l'application soit conforme, il suffit que les contraintes OCL soient respectées. Vérifier cette conformité est toutefois une difficulté qui n'a pas encore rencontré de solution industrielle.

OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme :

- des invariants sur des classes ;
- des préconditions et des post conditions à l'exécution d'opérations :
- les préconditions doivent être vérifiées avant l'exécution,
- les post conditions doivent être vérifiées après l'exécution ;
- des gardes sur des transitions de diagrammes d'états-transitions ou des messages de diagrammes d'interaction ;
- des ensembles d'objets destinataires pour un envoi de message, des attributs dérivés, etc.

**3.3 Réseau de pétri**

Les Réseaux de Pétri (abréviation RdP) ont été introduits par le mathématicien Allemand Carl Adam Pétri dans sa thèse "*Communication avec des Automates*" en

Allemagne à Bonn en 1962. Les réseaux de Pétri offrent un outil formel et une bonne représentation graphique qui permettent de modéliser et d'analyser les systèmes discrets, particulièrement les systèmes concurrents et parallèles.

La facette graphique des réseaux de Pétri, nous aide à comprendre aisément le Système modélisé. [24]

### 3.3.1 Définition

#### ➤ *Définition informelle*

Informellement un RdP est un graphe orienté comprenant deux sortes de nœuds : des places et des transitions. Ce graphe est constitué de telle sorte que les arcs du graphe ne peuvent relier que des places aux transitions ou des transitions aux places. On représente graphiquement les places par des cercles et les transitions par des barres (Figure 3.9). Les places servent à représenter les états du système modélisé, tandis que les transitions représentent les changements d'état ou les événements. Quelques interprétations typiques des transitions et leurs places d'entrées et de sorties sont présentées dans Le tableau (Table 2.1). [24]

Places d'entrées	transition	Places de sortie
Pré-conditions	Evènement	Post-conditions
Données d'entrée	Traitement	Données de sortie
Signaux d'entrée	processeur	Signaux de sorties
Ressources demandées	Tâche	Ressources libérées
Conditions	Clauses en logique	Conclusions
Buffers	Processeur	Buffers

Table 2.1 : Quelques interprétations typiques de transitions et de places.

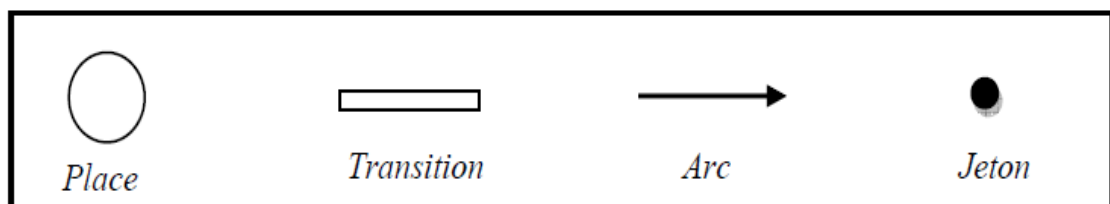


Figure 3.9 : Représentation graphique des éléments de RdP.

À chaque arc est associé un nombre entier strictement positif appelé poids de l'arc. Lorsque le poids n'est pas signalé, il est égal à "1" par défaut. Le RdP dont tous ses Arcs sont de poids "1" est appelé RdP ordinaire. Dans le cas où les arcs peuvent avoir des poids supérieurs à "1", il s'agit de RdP généralisé.

➤ **Définition formelle**

Formellement un RdP est un quintuple,  $R = (P, T, F, W, M_0)$  tel que : [24]

- $P = \{p_1, p_2 \dots p_m\}$  ensemble fini de places;
- $T = \{T_1, T_2 \dots T_m\}$  ensemble fini de transitions ;
- $F = (P \times T) \cup (T \times P)$  ensemble d'arcs;
- $W: F \rightarrow \mathbb{N} - \{0\}$  fonction de poids;
- $M_0: P \rightarrow \mathbb{N}$  marquage initial;
- $P \cap T = \emptyset$  et  $P \cup T \neq \emptyset$ .

La structure de RdP est donnée par le quadruplet  $Q = (P, T, F, W)$ ,  $Q$  représente le RdP Sans aucune spécification de marquage initial  $M_0$ . Un RdP marqué est noté par  $R = (Q, M_0)$ .

### 3.3.2 Représentation graphique de RdPs

Le réseau de Pétri est représenté par deux type de sommets alternés, les places  $P_i$  et les transitions  $T_i$ . Ces sommets sont reliés par des arcs orientés. Tout arc doit relier une place à une transition ou bien une transition à une place. [25]

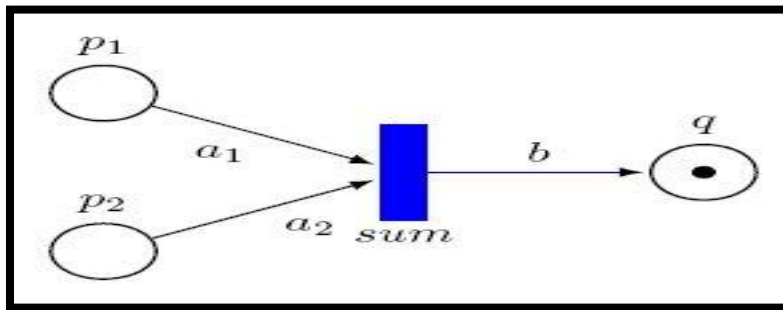


Figure 3.10 : Un réseau de Pétri simple

### 3.3.3 Marquage

On appelle marquage une distribution de jetons sur les places. Le marquage initial noté  $M_0$  est la distribution initiale de jetons dans le réseau à l'instant initial.

Un marquage définit l'état du système.

Le marquage d'un RdP est une application  $M : P \rightarrow \mathbb{N}$  donnant pour chaque place le nombre de jetons qu'elle contient. Le marquage de la place  $p_i$  est noté par  $M(p_i)$  qui est un nombre entier.[24]

### 3.3.4 Evolution de réseau de pétri

L'évolution d'état du réseau de Pétri correspond à une évolution de marquage.

Les jetons qui indiquent l'état du réseau à un instant donné, peuvent passer d'une place à une autre par un **franchissement** ou par un **tir** d'une transition. Dans le cas des réseaux dits à arcs simples ou de poids égal à 1 (cf. la figure 3.11), Le franchissement d'une transition consiste à retirer un jeton dans chacune des places d'entrée de la transition et à en ajouter un dans chacune de sorties de places de celle-ci. [26]

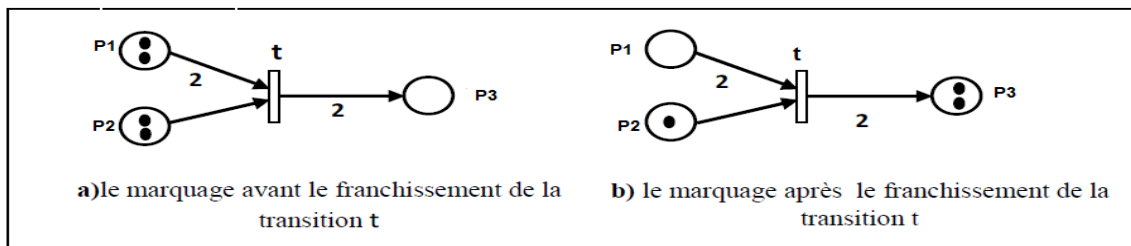


Figure 3.11 : Evolution d'états d'un réseau de Petri

En général l'évolution des états d'un réseau de Pétri marqués simple obéit aux règles suivantes :

- Une transition est **franchissable** ou **sensibilisée** ou encore **étirable** lorsque chacune des places d'entrées possède au moins le nombre de jetons correspondant au poids de l'arc le reliant à la transition.
- Le réseau ne peut évoluer que par franchissement d'une seule transition à la fois, transition sélectionnée parmi toutes celles validées au moment du choix.
- Le franchissement d'une transition est indivisible et de durée nulle.

Ces règles introduisent un certain indéterminisme dans l'évolution des réseaux de Pétri, puisque ceux-ci peuvent passer par différents états dont l'apparition est conditionnée par le choix des transitions tirées. Ce fonctionnement représente assez bien les situations réelles où il n'y a pas de priorité dans la succession des événements.

### 3.3.5 Sémantique du parallélisme et problème de conflits

Dans un réseau de Pétri, plusieurs transitions peuvent être franchissables à un moment donné. Cet aspect de parallélisme du tir des transitions pose un problème de choix pour l'état futur du réseau. En général, ce conflit est résolu par le choix d'une sémantique dite du parallélisme qui définit une stratégie de tir, tel que le tir d'une seule transition à la fois. [25]

### 3.3.6 Outils de modélisation des réseaux de Pétri

L'aspect formel des RdP a encouragé les développeurs à mettre au point une multitude d'outils de simulation et de vérification des RdP selon la technique de vérification de modèle (CPNTools, CPNAMI, PROD, JARP, MARIA, LOLA, Petri Net Kernel, GreatSPN, INA, OPMSE, Artifex, ExSpect, FLOWer, f-net, GD ToolKit, Helena, HPSim, INA, JARP, JFern, JPetriNet, Opera, ORIS, PACE, ...etc). La plupart de ces outils présente un environnement graphique d'édition des RdP avec la possibilité de simuler le modèle et d'analyser des propriétés génériques des RdP. [24]

### 3.3.7 Extensions des réseaux de Pétri

La modélisation d'un système réel peut mener à des réseaux de Pétri de taille trop importante rendant leur manipulation et/ou leur analyse difficile. La question est alors de modifier (étendre) la modélisation par RdP de façon à obtenir des modèles RdP de plus petite taille. Cette question a motivé la nécessité de faire des extensions des réseaux de Pétri. En plus de ça, les points suivants ont motivé cette extension :

- Certaines propriétés ne peuvent pas être exprimées à l'aide des réseaux usuels.
- Besoin d'avoir une information plus précise sur les jetons transitant dans le réseau.

Ceci a donné naissance à plusieurs extensions de réseaux de Pétri tels que les réseaux de Pétri colorés, Les ECATNets, les réseaux de Pétri algébriques, les réseaux de Pétri Prédicat/Transition, etc. [26]

### 3.4 l'analyse des réseaux de pétri

La construction de la structure du RdP qui modélise le système *réel* est une tâche très importante. La tâche suivante qui est de même importance est l'étude ou l'analyse des propriétés de ce réseau. Le but de transformer les processus métiers vers les RdPs est de pouvoir les analyser. Pour ce faire, nous avons choisi l'outil d'analyse INA (Integrated Net Analyser). Et pour faire une analyse sur un réseau de pétri, il y a des méthodes, nous allons représenter comme le suite.[24]

#### 3.4.1 Méthode d'analyse

La modélisation d'un système n'est utile que si elle permet d'analyser ses propriétés. La théorie des réseaux de Petri offre des techniques d'analyse puissantes pour valider des modèles de comportement de systèmes (analyser les propriétés d'un réseau de Petri), nous pouvons citer l'analyse grâce aux graphes des marquages, l'analyse par algèbre linéaire, l'analyse structurelle, et l'analyse par réduction.[26 ]

##### ✓ Analyse par graphes des marquages

Deux situations Peuvent se présenter :

1. Le graphe est fini. C'est la situation la plus favorable car dans ce cas toutes les propriétés peuvent être déduites simplement par inspection de celui-ci. Nous avons déjà vu plusieurs exemples de cette utilisation.
2. Le graphe est infini. Dans ce cas, on construit un autre graphe appelé "graphe de couverture" permettant de déduire certaines propriétés.

##### ✓ Analyse par algèbre linéaire

L'analyse par algèbre linéaire permet d'étudier des propriétés d'un réseau (caractère borné, vivacité) indépendamment d'un marquage initial. De ce fait, on parlera de propriétés structurelles du réseau. Par exemple, on pourra dire qu'un réseau est *structurellement borné* s'il est borné pour tout marquage initial fini. De la même façon, si pour tout marquage initial, le réseau est vivant, on dira que le réseau est structurellement vivant.

✓ **Analyse par réduction**

avec un nombre réduit de places et un nombre réduit de transitions (règles de réduction). Ce dernier doit être :

- 1.équivalent au RdP marqué de départ (pour les propriétés étudiées).
2. suffisamment simple pour que l'analyse de ses propriétés soit simple.

En général, le RdP simplifié ne peut pas s'interpréter comme le modèle d'un système. On peut distinguer deux types de règles de réduction:

1. règles de réduction préservant la vivacité et la bornétude (et leurs propriétés associées),
2. règles de réduction préservant les invariants de marquage.

### **3.4.2 Outils d'analyse des RdP**

La puissance des réseaux de Petri réside dans l'existence d'une batterie d'outils d'analyse tels que INA ( Integrated Net Analyzer) [INA], PEP (Programming Environment based on Petri nets) [Pep], TINA [Tina],etc ...[26 ]

#### **3.4.2.1 INA (Integrated Net Analyser)**

L'INA est un outil d'analyse largement utilisé par la communauté des réseaux de Petri. Il a été développé par le prof. Dr. Peter H. Starke. C'est un programme interactif dirigé par les menus qui permet à l'utilisateur d'éditer (dans une forme textuelle), de réduire, d'exécuter et d'analyser les modèles des RdPs.

Pour analyser un réseau de Petri en utilisant INA, il doit être transformé vers la spécification INA équivalente. C.à.d passer d'une représentation graphique vers une description textuelle. [26]

✓ **Mise en route**

En règle générale, vous devez créer un répertoire sur votre disque dur, et copier le contenu Exécutable INA en ce répertoire, Pour les opérations nécessaires, s'il vous plaît se référer à votre système d'exploitations manuel, pour tous les fichiers dont il a besoin, l'INA recherche dans le répertoire à partir duquel il a été lancé, il enregistre également ses fichiers dans ce répertoire, devez-vous travailler avec différents types de réseaux dans le même temps, il est recommandé de créer des sous-répertoires pour chaque type de réseau ou réseau, Depuis cette façon, les options actuelles sont enregistrées séparément, il ne sera pas nécessaire de les sélectionner à chaque fois, car il est suffisante pour changer le répertoire et redémarrez le programme, Le répertoire dans lequel le fichier exécutable Le de l'INA a été placé doit être enregistrée dans le chemin de recherche.[27]



### ✓ fichiers, extensions, et formats des réseaux

dans cette section, vous trouverez un aperçu des fichiers utilisés et réalisée par l'INA, une Liste des extensions de fichiers, une présentation de certains formats de fichiers.

#### • les fichiers importants

nous allons citer les fichiers importants comme suit :

- OPTIONS.INA : toutes les options que vous avez mis en place au cours de votre session.
- COMMAND.INA : ce fichier est créé, à la fin d'une session à la demande.
- SESSION.INA : INA écrire tous les résultats obtenus lors d'une session dans ce fichier.

davantage fichier qui peut être dans le répertoire courant après une session INA :

- INVART.HLP : fichier auxiliaire pour le calcul de l'invariant
- ININET.PNT : fichier auxiliaire qui contient le réseau initial

#### • Les Extensions de fichier

Par défaut INA utilisé les extensions suivantes :

(.pnt), (.cnt), (.mar), (.tim), (.tmd), (.atm), (.icp), (.ict), (.pri), (.cap), (.red), (.exc), (.val)  
(.inv), (.hlp), (.res), (.Tra), (.sta), (.cir), (.pdc), (.ctl), (.prf)

#### • Les formats de fichier

Il y a plusieurs formats de fichier qui supportants par l'outil INA nous avons choisi le format (.pnt), comme la représentation de la figure suivante :

```
<pnt-file> ::= <netheader> "<cr>"
               <netstruct> "<cr>"
               "@<cr>"
               <placedata> "<cr>"
               "@<cr>"
               <transdata> "<cr>"
               "@<cr>"

<netheader> ::= "P  M  PRE,POST  NETZ  " <netid>

<netid> ::= <netnr> [ ":" <netname> ]

<netnr> ::= <number>

<netname> ::= <name>

<netstruct> ::= <placedef>
               { "<cr>" <placedef> }
```

```

<placedef> ::= { " " <placenr> " " <tokens> " " [ <prelist> ] [ ", " <postlist> ]

<placenr> ::= <number>

<tokens> ::= <number>

<prelist> ::= { <transnr> [ ": " <arcmult> ] " " }

<postlist> ::= { <transnr> [ ": " <arcmult> ] " " }

<transnr> ::= <number>

<arcmult> ::= <number>

<placedata> ::= "place nr.           name capacity time"
               { "<cr>           " <placenr> ": " <placename> "           " <capacity> <time> }

<placename> ::= <name>

<capacity> ::= " " <nr> | "oo"

<time> ::= <number>

<transdata> ::= "trans nr.           name priority time"
               { "<cr>           " <transnr> ": " <transname> "           " <priority>
               <time> }

<transname> ::= <name>

<priority> ::= <number>

```

**Figure 3.12 : syntax d'un fichier .pnt**

### 3.5 Conclusion

Dans ce chapitre, nous avons fait premièrement une présentation sur les outils de modélisation et découvrent les composants et les propriétés de ces outils.

Ces outils concernant le langage UML, on a présenté leurs diagrammes et précisément le diagramme de classe où nous avons souligné tous les éléments principaux qui le composent.

A nous avons présenté Les réseaux de Pétri, ce qu'il représente un outil puissant pour l'analyse et la simulation par rapport aux diagrammes de classe d'UML. Ainsi une présentation générale sur l'outil d'analyse INA.