République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Superieur et de la Recherche Scientifique

UNIVERSITÉ ECHAHID HAMMA LAKHDAR
EL OUED

FACULTÉ DES SCIENCES EXACTES

Mémoire de fin d'étude

MASTER ACADEMIQUE

Domaine: Mathématiques et Informatique

Filière: Informatique

Spécialité: Systèmes Distribués et Intelligence Artificielle

THEME

Conception et Réalisation d'un outil de Génération automatique de Spécification Maude à Partir des réseaux de Petri

Présenté par : Hachi Yacine & Benhacine Djilani

Devant le jury composé de :

	Président
	Examinateur
Mr. Boucherit Ammar	Rapporteur

Année universitaire 2015 – 2016

Remerciements

« Avant toute chose, nous remercions dieu de nous avoir donné toute cette force, qui nous a permis d'arriver a bon port » .

En premier lieu , nous tiendrons à remercier notre encadreur M. Ammar Boucherit

qui nous a aidé et conseillé durant cette année.

Nos remerciements vont également à tous les Enseignants de département d'informatique de l'université d'el oued pour avoir accepter de juger ce travail.

Enfin, nous remercions tous ceux que nous ont aidé de prés ou de loin contribuer pour ce travail.

Hachi Yacine

Ben Hcin Djilani

Sommaire

Introduction Générale

Chapitre 01 : Les Réseaux de Petri

Chapitre 02 : La Logique de Réécriture

Résumé

Les Réseaux de Petri (RdP) ont largement connu comme étant un outil mathématique très général permettant de modéliser le comportement de systèmes dynamiques à événements discrets quelque soit leurs domaines d'application (Informatique, Télécommunication, Production, …). Par conséquent, il devient nécessaire de pouvoir vérifier formellement les propriétés pertinentes des systèmes modélisés.

Ce projet s'inscrit dans ce contexte et focalise sur le développement d'un outil permettant de générer des spécification Maude automatiquement à partir des réseaux de Petri. Ce qui permet ensuite la vérification d'exigences temporelles et l'analyse des propriétés des spécifications par Maude LTL Model-Checker.

Abstract

Petri Nets (PN) largely had known as a very general mathematical tool to model the behavior of dynamic systems whatever their application areas (Computer science, Telecommunications, Manufacturing ...). Therefore, it becomes necessary to formally verify the relevant properties of such modeled systems.

This project falls within this context and focuses on the development of a tool to automatically generate Maude specification from Petri nets. Of course, this will allow the verification of timing requirements and analysis of the properties of Maude specifications via LTL Model Checker.

الملخص

تعتبر شبكات بتري من بين الوسائل الرياضية الفعالة لتمثيل سلوكات (الجانب الديناميكي) الأنظمة الديناميكية بغض النظر عن مجال تطبيقها (إعلام آلي، الإتصالات، التصنيع، ...).
ولهذا أصبح من الضروري العمل على التحقق الصارم من خصائص الأنظمة الممثلة بها.
يندرج عملنا في هذا الإطار ويركز على تطوير أداة لتوليد التمثيل المكافئ آليا لشبكات بتري في منطق إعادة الكتابة وهو ما يسمح بالتحقق من الخصائص المتعلقة بالزمن عن طريق الأداة الخاصة بذلك في نظام مود.

Introduction Générale

1. Contexte général et Problématique

En effet, les réseaux de Petri ont largement connu comme étant un outil de modélisation à la fois élégant et d'une grande puissance. C'est notamment dans le contexte systèmes dynamiques à événements discrets ainsi que les protocoles de communication informatiques et des systèmes informatiques, que cet outil prend toute son importance. Par conséquent, il devient nécessaire de chercher des approches pour pouvoir assurer formellement le bon fonctionnement en termes de surêté et de vivacité des systèmes complexes modélisés.

D'autre part, la logique de réécriture est une logique qui permet de raisonner d'une manière correcte sur les systèmes concurrents non-déterministes grâce à l'expressivité étonnante dont elle possède du fait qu'elle englobe et unifie plusieurs modèles formels qui expriment la concurrence tels que les réseaux de Petri, les systèmes de transitions, CCS, LOTOS ...etc.

En plus, la génération de code est une opération permettant de générer du code automatiquement. Son but est d'automatiser l'opération répétitive de production de code afin de minimiser les risques d'erreurs et de permettre au programmeur de se concentrer sur l'écriture de code à plus grande valeur ajoutée.

Notre travail s'inscrit dans ce contexte où on veut exploiter sa richesse ainsi de faciliter pour les dévloppeur de bénéficier des outils formelles qu'offre son langage Maude tel que son model-checker.

2. Objectifs et Contributions

L'objectif principal du présent travail est de concevoir et réaliser un outil de génération automatique de spécification Maude des modèles des systèmes basés sur les réseaux de Petri (RdP). Notre premier plan était d'utiliser un des éditeurs des réseaux de Petri en exploitant sa représentation XML du RdP. Mais après une longue discussion et recherche sur ce point, on a conclu que le développment d'un éditeur simple adapté à notre besoin sera plus avantageux en terme que notre trovail ne sera pas liés d'aucun autre outil externe.

Afin d'atteindre le but noté, on a développé un éditeur simple des réseaux de Petri. Bien sûr, ce travail a été basé sur l'utilisation d'un ensemble de règles de transformation bien-fondé des RdPs vers la logique de réccriture. Dans la pratique, on a représente le RdP sous forme d'une matrice d'incidence pour être utiliser par la suite dans le processus de génération automatique.

3. Organisation du mémoire

Ce mémoire se compose de deux parties comportant deux chapitres chacune. La première partie (Notions de Base) présente tout d'abord les concepts de basé des réseaux de Petri et de la logique de réécriture.

La seconde partie (Conception et Implémentation) présente notre contribution dans la conception et la l'implémentation de l'outil de génération automatique des spécification Maude à partir des réseaux de Petri.

- Dans le premier chapitre, on a présenté brèvement les notions élémentaires des réseaux de Petri afin de faciliter leur exploitation dans la deuxième partie.
- Le deuxième chapitre a été consacré à la présentation des principaux concepts du formalisme de la logique de réécriture, et de son langage Maude.
- Dans le troisième chapitre, nous présentons notre vue pour la modélisation et la conception de l'outil demandé.
- Le quatrième chapitre présente l'outil devéloppé ainsi que le processus de génération dès l'édition du RdP jusqu'à la génération de spécification Maude finale.

Première Partie

Notions de Base

Chapitre 01

Les Réseaux de Petri

Chapitre 01

Les réseaux de Petri

1.Introduction :

C'est Carl Adam Petri qui inventé les réseaux de Petri en 1962 dans une partie de sa thèse de doctorat (Communication par les automates). Le formalisme des réseaux de Pétri est un outil permettant l'étude de systèmes dynamiques et discrets. Il permet d'obtenir une représentation mathématique modélisant le système. L'analyse de cette représentation (du réseau de Pétri) peut révéler des caractéristiques importantes du système concernant sa structure et son comportement dynamique. Les résultats d'une telle analyse sont utilisés pour évaluer le système et en permettre la modification et/ou l'amélioration le cas échéant. **[AD 68] .**

2.Concepts de base :

2.1. Définitions :

Définitions Informelles :

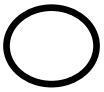
Un **réseau de Petri** (RdP) est un quadruplet $R = <P,T,Pre, Post>$

- *P* est un ensemble fini de **places**, Pi={P1, P2, ...}.
- *T* est un ensemble fini de **transitions, T={T1, T2, ...}.**
- *Pre* : P x T $\rightarrow$ N est l'application « place d'entrée ».
- *Post* : P x T $\rightarrow$ N est l'application « place de sortie ».

Les Places et transitions sont reliées par des arcs orientés. On dira qu'un RdP est un graphe biparti orienté.

A chaque arc, on attribut un poids (nombre entier). Par défaut ce nombre est égal à 1 **[MB].**

2.2. Modélisation d'un système (événement – condition):

Condition: modélisée à l'aide d'une place. 

Evénement: modélisé à l'aide d'une transition. 

2.2.1.Condition :

Une condition est un prédicat logique d'un état du système, Elle est soit vraie, soit fausse.

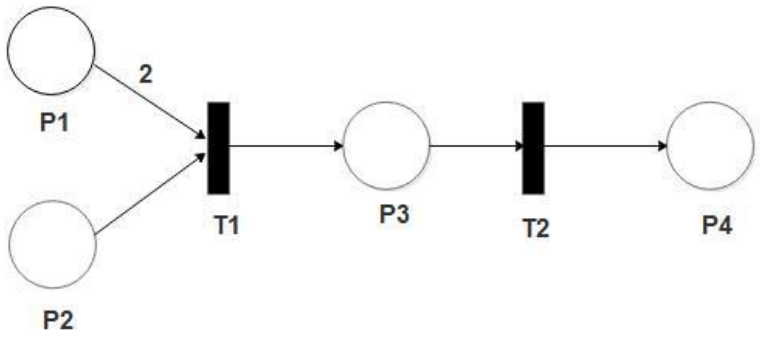
2.2.2.Evénement :

Les événements sont des actions se déroulant dans le système, Le déclenchement d'un événement dépend de l'état du système. Un état du système peut être décrit comme un ensemble de conditions. **[SBS].**

2.3.Déclenchement, pré-condition, post-condition:

Les condition nécessaires au déclenchement d' un événement sont les pré-conditions de l'événement. Lorsqu'un événement se produit, certaines de ses pré-conditions peuvent cesser d'être vrais alors que d 'autres conditions, appelées post-conditions de l'événement deviennent vraies **[MB].**

Exemple :



pour le RdP ci-dessus, on a :

P= {p1,p2,p3,p4} ; T={t1,t2} ; Pre(p1,t1)= 2; Pre(p2,t1)=1 ; Pre(p3,t2)=1

Post(p3,t1)=1; Post(p4,t2)=1

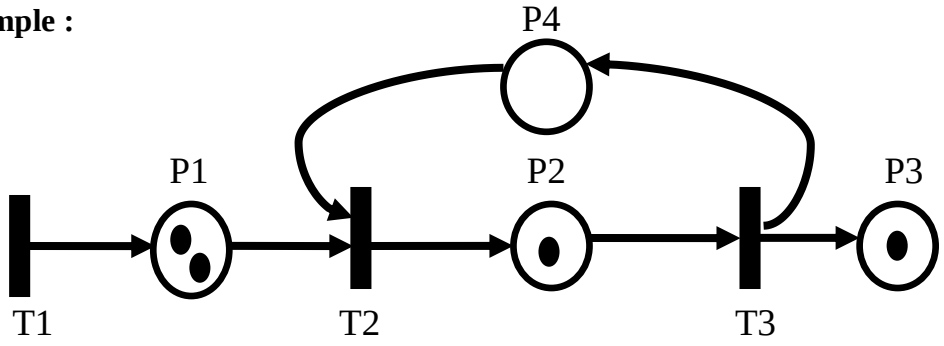
Le reste: Post(p_i,t_j)=0 et Pre(p_i,t_j)=0

3. Evolution d'un RdP:

3.1. Marquage :

Chaque place contient un nombre entier (positif ou nul) de marques ou jetons. Le nombre de marque contenu dans une place Pi sera noté soit M(Pi) soit mi. Le marquage du réseau à l'instant i, Mi est défini par le vecteur de ces marquages mi c'est à dire Mi= (m1, m2, …,mn). Le marquage dit initial décrit l'état initial du système modélisé (M0). **[SBS].**

Exemple :



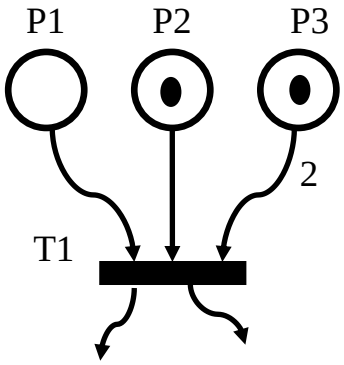
Le marquage de ce réseaux est :

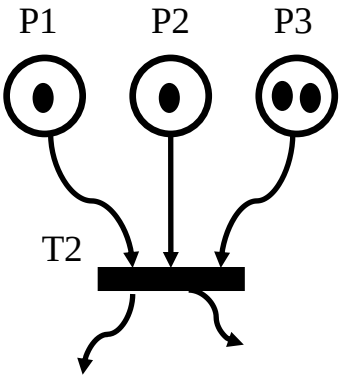
M(p1)= 2 ; M(p2)= 1 ; M(p3)= 1 ; M(p4)= 0 ;

3.2. Franchissement d'une Transition :

Une transition est dite « franchissable » si chacune de ses *places d'entrée* contient un nombre de jetons supérieur ou égal à celui indiqué sur la flèche correspondante.

Exemples :



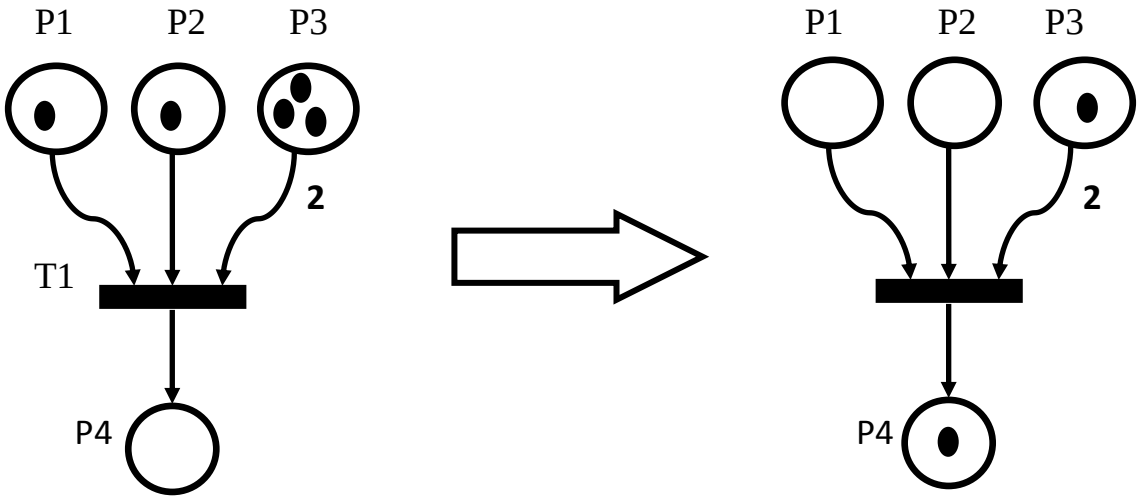


T1 n'est pas franchissable car P1 et P3 ne contiennent pas suffisamment de jetons !

T2 est franchissable

❖ Le franchissement est une opération qui consiste à retirer des jetons des places en entrée et à ajouter des jetons dans les places en sortie de la transmission franchie. Le nombre de jetons retirés ou ajoutés est égal au poids de l'arc reliant la transition à la place.

Exemples :

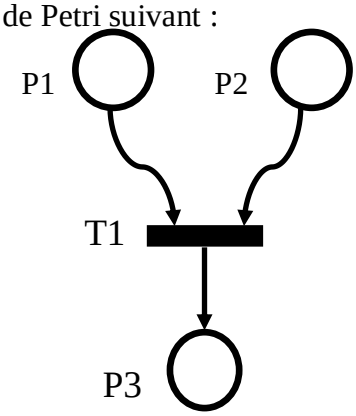


Avant franchissement T1 Après franchissement T1

3.3.Synchronisation :

La synchronisation est l'action de coordonner plusieurs opérations entre elles en fonction du temps.

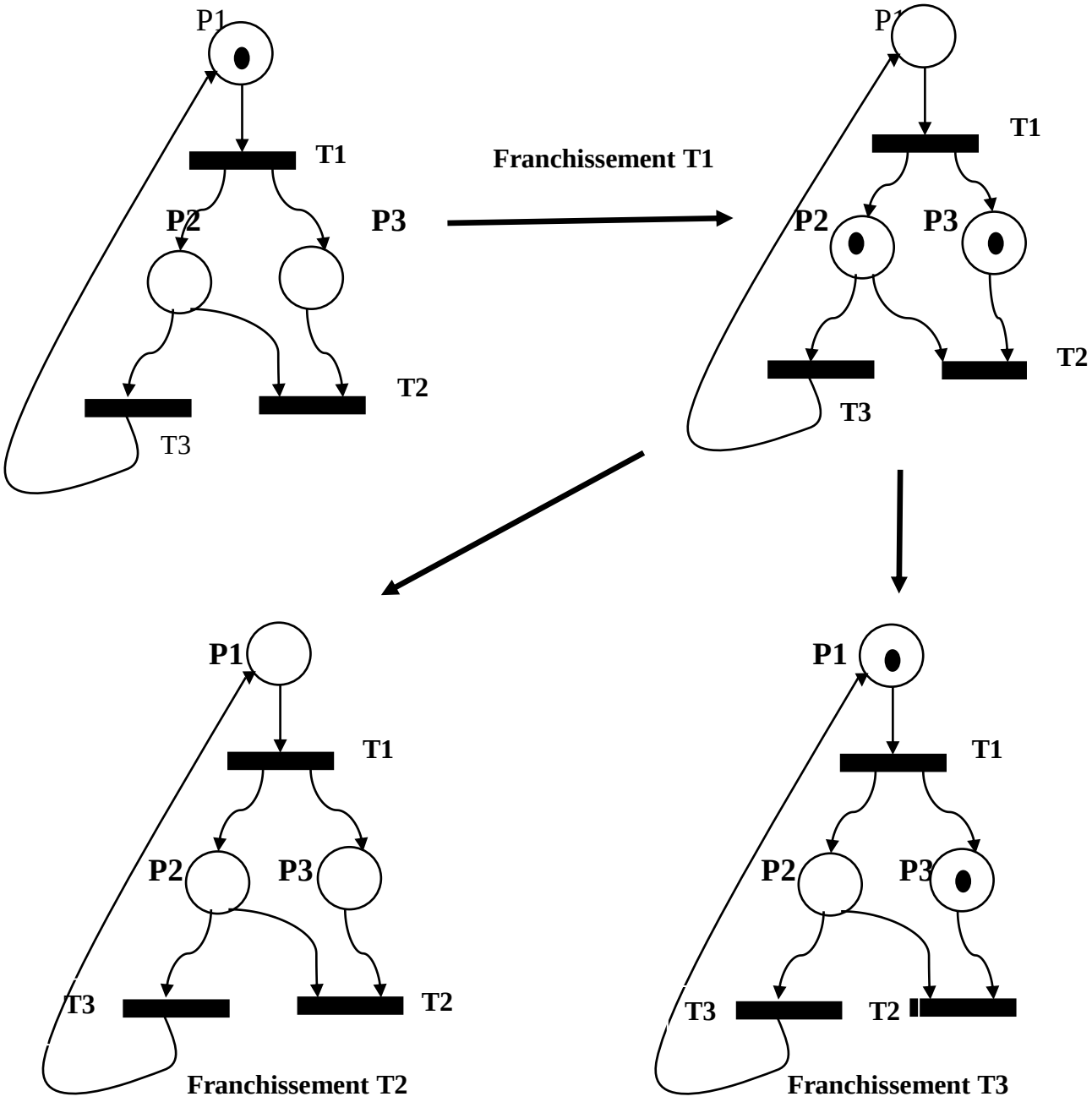
Exemples 1: soit le réseau de Petri suivant :



❖ Le franchissement de la transition T1 Nécessite la validité de l'arc de P1 et P2 en même temps.

➢ L'évolution d'un RdP correspond à l évolution de son marquage au cours de temps (évolution de l'état de système), il se traduit par un déplacement de marque ce qui s'interprète comme la (consommation/production) de ressources déclenchée par des événements ou des actions.

Exemples : on a l'évolution de RdP suivant :



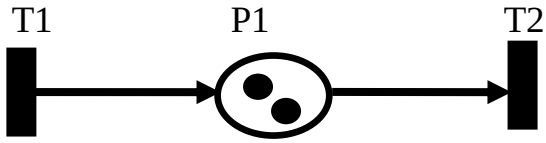
❖ Deux cas spéciaux des réseaux de Petri :

On a deux cas spéciaux des réseaux de Petri :

- **Transition génératrice :** cette transition est toujours produire des jeton.
- **Transition absorbante :** cette transition est toujours consomme des jeton.

Le réseau de Petri présenté dans l'exemple suivant illustre cette nouvelle notion

Exemple : soit le réseau de Petri suivant :

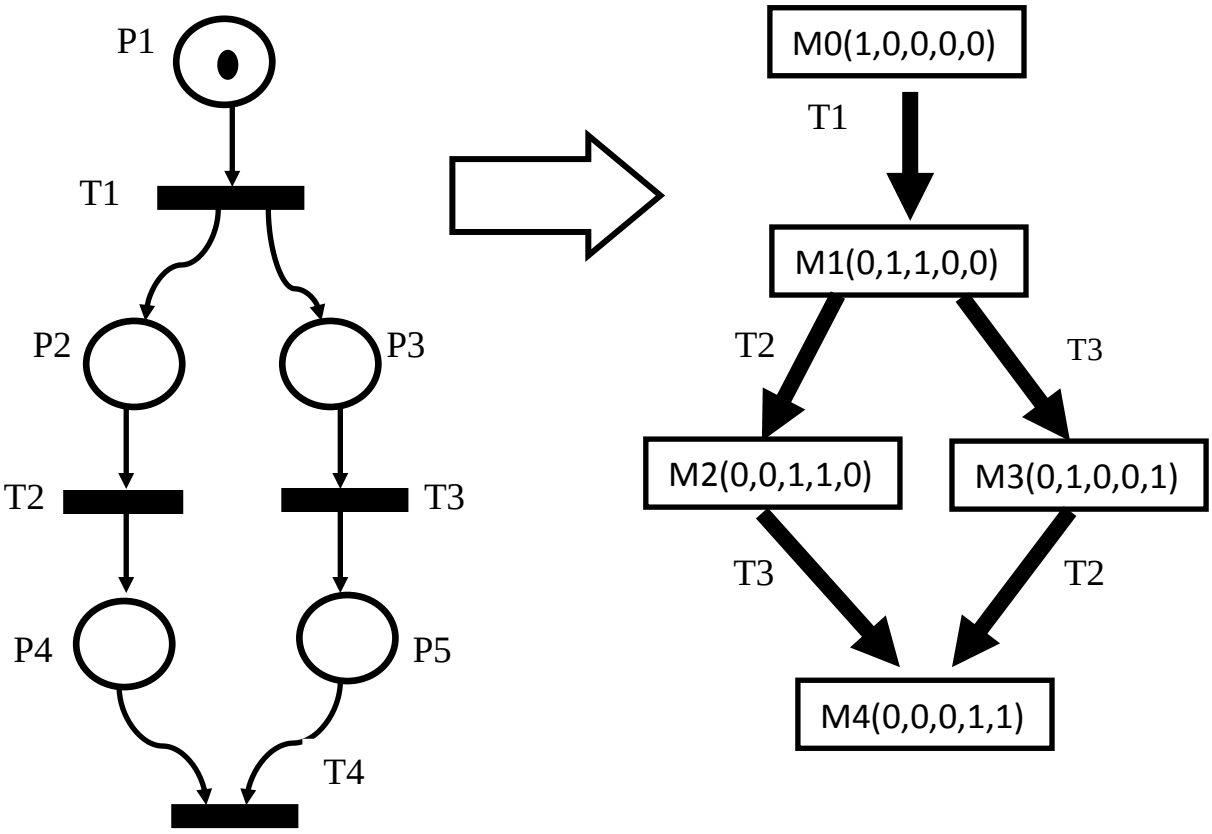


Dans ce réseau de Petri ,T1 est une transition génératrice et T2 est une transition absorbante.

4.Graphe de transitions (ou encore Graphe d'états):

C'est une représentation graphique des possibilités d'évolution du RdP , Elle est obtenue en partant du marquage initial et en étudiant à chaque marquage obtenu Mi après le franchissement d'une transition Tj les différentes possibilités d'évolution du RdP, Ceci correspond aux différentes transitions validées par le marquage Mi. **[SBS].**

Exemples : Soit le réseau de Petri suivant :



- Le franchissement, à partir de M0, de T1 puis T2 puis T3 conduit au marquage M4. On appellera T1T2T3 séquence de franchissement et on notera :

  $M0 \xrightarrow{T1T2T3} M4$

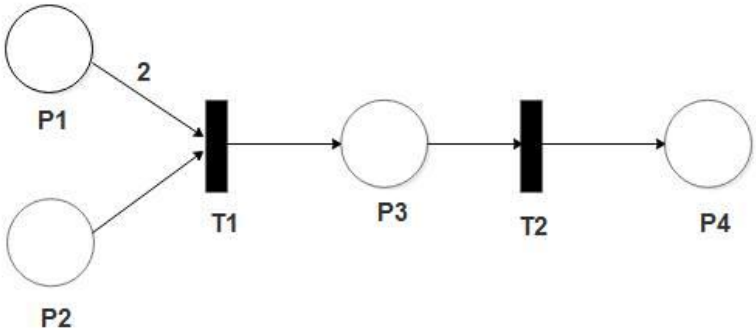
- Le franchissement, à partir de M0, de T1 puis T3 puis T2 conduit au marquage M4. On appellera T1T3T2 séquence de franchissement et on notera :

  $M0 \xrightarrow{T1T3T2} M4$

5.Matrice d'incidence :

La matrice d'incidence fait la synthèse de tous les liens entre places et transitions du RdP. Cette matrice est en général rectangulaire et possède un nombre de colonnes égal au nombre de transitions du réseau, ainsi qu'un nombre de lignes égal au nombre de places du réseau. Chaque élément de la matrice témoigne de la présence ou de l'absence de lien entre chaque place et chaque transition, ainsi que du poids attaché à l'arc en question. La direction de cet arc est transcrite par le signe de l'élément en question **[MB].**

Exemple : soit le réseau de Petri suivant



La matrice d'incidence de ce réseaux de Petri est :

	T1	**T2**
P1	2	0
P2	1	0
P3	-1	1
P4	0	-1

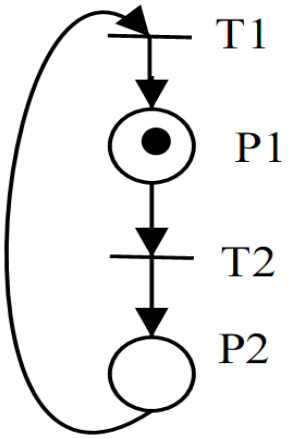
6. Quelques propriétés des RdPs :

L'évolution du marquage d'un RdP se fait par le franchissement de transition ,lorsqu'au de cours de son évolution ,certains transitions ne sont jamais franchies, cela indique que l'événement associé à la transition ne se produit pas et que le marquage d'une parti du RdP n'évolue pas, cela indique que le sous système modélise par cette partie-là ne fonctionnera pas , il y a donc un problème au niveau de conception de système, l'idée d'être capable de détecter systématiquement ce phénomène par l'analyse de propriété du modèle RdP du système afin de disposer d'un outil d'aide à la conception des systèmes.

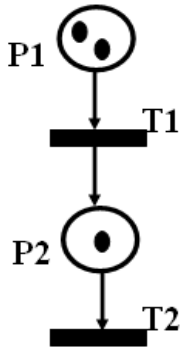
6.1. Vivacité de RdP :

Si une transition Tj est vivante alors, à tout instant, on sait que Tj peut être franchie dans le futur, Dans le cas d'un RdP modélisant un système fonctionnant en permanence, si une transition n'est pas vivante et si une fonction du système est associée au franchissement de cette transition, cela veut dire qu' à partir d'un certain instant, cette fonction ne sera plus disponible dans le futur, ce qui peut traduire une erreur ou une panne. **[SBS].**

Exemple :



1 .RdP vivant

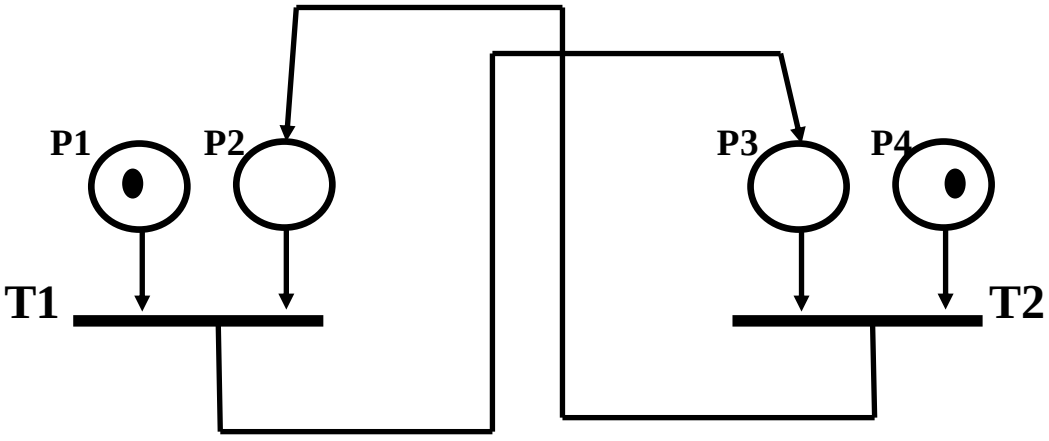


2 .RdP non vivant

6.2. Blocage de RdP :

un RP est dit être en situation *de blocage* lorsque, dans un marquage donné M, *aucune transition* n'est franchissable. **[SBS].**

Exemple :



- Le processus A est arrêté car il a besoin d'une ressource détenue par B, alors que le processus B est arrêté car il a besoin d'une ressource détenue par A.

7.Exemple de Modélisation par réseaux de Petri :

le producteur-consommateur est un exemple simple de réseaux de Petri

Objectif :

Pour modaliser la coordination entre deux processus dont un est le producteur et l'autre le consommateur d'une ressource.

- Le producteur produit un objet (item) et le dépose dans un tampon (bac).
- Le consommateur prend l'objet dans le bac et le « consomme ».
- Contrainte: avant que le consommateur ne puisse exécuter l'action « consommer », le producteur doit avoir fini l'action « produire ».

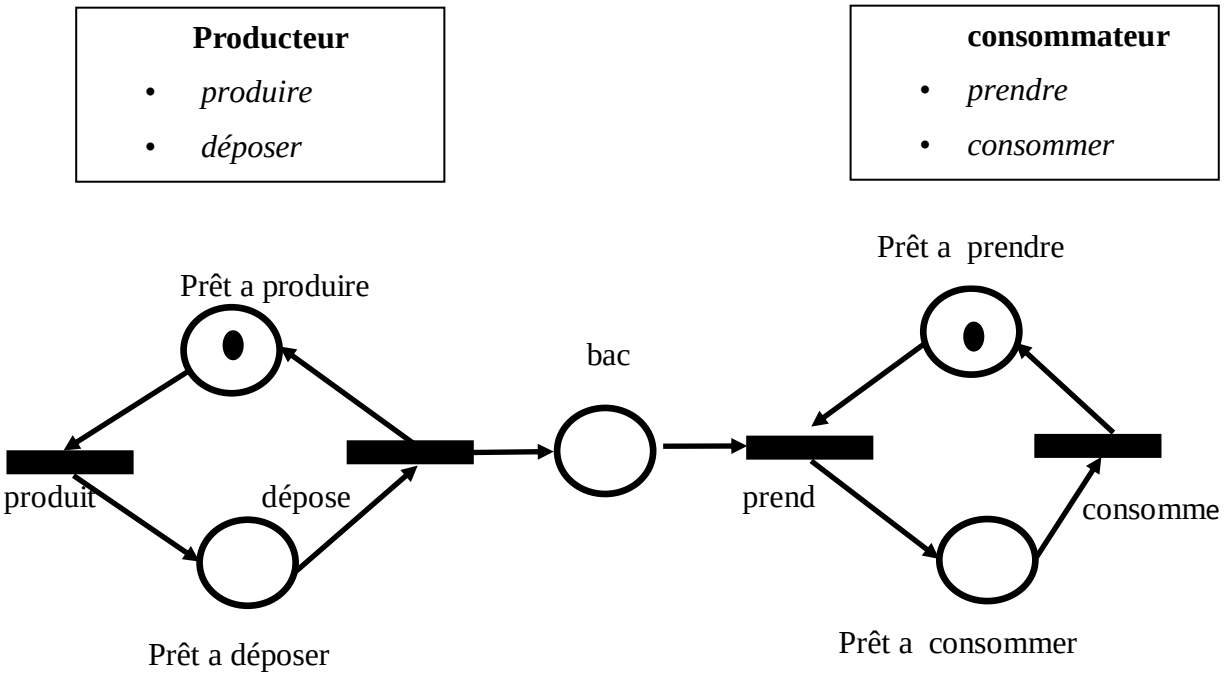


Figure 1 : modélisation de producteur-consommateur par réseaux de Petri

8.Domaine d'applications des réseaux de Petri :

Les réseaux de Petri ont largement utilisés pour la modélisation et on les retrouve dans des différents domaines, dont :

1. **L'industrie :**

   - ✓ la modélisation des chaines de production (de fabrication).

2. **L’informatique :**

- ✓ la modélisation des systèmes informatiques.
- ✓ la modélisation des protocoles de communication.

3. **la chimie :**

- ✓ Les réactions chimiques peuvent être modélisées par les réseaux de Petri.

9 . Conclusion

Nous avons présenté dans ce chapitre les concepts liés aux réseaux de Petri. ceux-ci ont l’avantage d'être visuels et dynamiques. Leur simulation permet de vérifier que le modèle correspond bien à ce qu'on veut qu'il fasse. En plus, les réseaux de Petri étant particulièrement adaptés pour traiter les problèmes de synchronisation et de fonctionnement en parallèle.

Chapitre 02

La Logique de Réécriture

Chapitre 02

La Logique de Réécriture

1 .Introduction

La réécriture est un paradigme général d'expression du calcul dans des diverses logiques computationnelles, Dans la logique de réécriture, une réécriture d'un terme consiste à le remplacer par un terme équivalent, conformément aux lois d'algèbre de termes. Cette logique a été présentée par José Meseguer [Mes 92], comme une conséquence de son travail sur les logiques générales pour décrire les systèmes concurrents. Cette logique formalise le processus de réécriture pour calculer une relation de réécrivabilité entre les termes algébriques. Elle permet de raisonner sur des changements complexes possibles correspondant aux actions atomiques axiomatisées par les règles de réécriture ,La logique de réécriture est proposée comme un cadre logique dans lequel d'autres logiques peuvent être représentées, et comme un cadre sémantique pour spécifier plusieurs systèmes et langages dans des domaines variés. Elle offre des techniques d'analyse formelle permettant de prouver des propriétés du système à spécifier, et de raisonner sur ses changements. Plusieurs modèles de la concurrence ont fait déjà l'objet d'une intégration dans la logique de réécriture, nous citons : Les systèmes de transitions étiquetés , les réseaux de Petri , Comme il existe d'autre applications tel que : Les systèmes orientés objets, spécification des langages et des systèmes, les systèmes concurrents et/ou parallèles, vérification formelle des spécifications exprimées en logique de réécriture, etc [Clav 99 b].

2 .Algèbre des termes

Les termes (et donc les algèbres des termes) constituent une notion de base en logique, parce que les formules qui permettent l'établissement des démonstrations sont toujours construites à partir des termes. Pour cette raison et dans le but d'éclaircir encore mieux les différentes

notions concernant la logique de réécriture, nous allons donner une définition précise de ses termes ainsi que des définitions d'autres concepts nécessaires à la compréhension de ces derniers **[Mem C].**

2.1 Signature

Définition

Une signature ∑ est une paire (S,Op) où :

- S est un ensemble de noms de sortes : $s_1, s_2, \ldots, s_m$
- Op est un ensemble d'opérations f : $s_1\ s_2 \ldots s_n \rightarrow s_{n+1}$

Pour toute opération f

- $(s_1, s_2, \ldots, s_n, s_{n+1})$ est appelé le profil de l'opération f
- $s_1\ s_2 \ldots s_n$ domaine de f
- s_{n+1} co-domaine de f
- n est l'arité de f.

Les constantes sont des opérations d'arité nulle **[Mem C]**.

Exemple

Signature des booléens :

Sorts bool } **S**

Opns true : → bool
false : → bool
not(_) : bool → bool
and : bool bool → bool
_or_ : bool bool → bool

} **Op**

} ∑

2.2 Les termes:

Définition

Soit ∑=(S, Op) une signature et V est un ensemble infini dénombrable de variables, L'ensemble des termes T est le plus petit ensemble contenant les variables et les constantes

(T =C U V) stable par l'application des symboles de fonctions de Op à des termes. Autrement dit, un terme est un mot qu'on peut obtenir en appliquant récursivement un nombre fini de fois les règles ci dessous:

- tout symbole de constante est un terme.
- toute variable est un terme.
- Si f est un symbole de fonction d'arité n et si t_1, …, t_n sont des termes, alors $f(t_1,\ldots,t_n)$ est un terme **[Mem C].**

2.3 ∑-équation

Définition :

Soit ∑=(S, Op) une signature et V un ensemble de variable, une ∑-équation est une paire (t, t') avec t et t' deux éléments de $T_{\sum}(V)$. Sachant que la notation $T_{\sum}$ (V) est utilisé pour désigner l'ensemble des termes T formés à partir de l'ensemble des variables V vérifiant la signature ∑ . Une ∑-équation (t, t') est notée t = t' **[Mem C].**

2.4 Spécification algébrique

Définition :

Une spécification algébrique est le couple Spec = (∑, E) où :

- ∑ : est une signature.
- E : est un ensemble de ∑-équations.

Exemple

Spécification algébrique des booléens

Sorts bool

Opns true :→ bool

false :→ bool

not(_) : bool → bool

and : bool bool →bool

_or_ : bool bool → bool

Vars x : bool

Eqns (not(true)=false)

(not(false)=true)

(true and x = x)

(false and x = false)

(true or x = true)

(false or x = x)

3. Logique de Réécriture

La logique de réécriture est apparue juste après les papiers de José Meseguer en 1990. C'est une logique de changements pouvant être associés aux états et aux calcul dans les systèmes concurrents. Elle a la bonne propriété d'être un cadre sémantique unificateur de plusieurs systèmes et modèles concurrents. En particulier, elle soutient très bien le calcul concurrent orienté objet. La logique de réécriture est un modèle de calcul et un cadre sémantique expressif pour la concurrence, le parallélisme, la communication, et l'interaction.

3.1. Théorie de réécriture:

Dans la logique de réécriture, un système concurrent est décrit par une théorie de réécriture $R = (\Sigma, E, L, R)$ où (Σ, E) désigne la signature (une théorie équationnelle) définissant la structure algébrique particulière des états du système (multi-ensemble, arbre binaire…) qui sont distribués selon cette même structure. La structure dynamique du système est décrite par les règles de réécriture étiquetées R (L est un ensemble d'étiquettes de ces règles). Les règles de réécriture précisent quelles sont les transitions élémentaires et locales possibles dans l'état actuel du système concurrent.

Chaque règle (notée $[t] \rightarrow [t']$) correspond à une action pouvant survenir en concurrence avec d'autres actions. Donc, la logique de réécriture est une logique qui capture clairement le changement concurrent dans un système **[MM96].**

3.2. Théorie de réécriture étiquetée :

Une théorie de réécriture R est un 4-uplet (Σ, E, L, R) tel que :

1. Σ est un ensemble de symboles de fonctions, et de sortes.
2. E un ensemble de Σ-équations (l'ensemble des équations entre les Σ-termes).
3. L est un ensemble d'étiquettes.

4. R est un ensemble de règles de réécriture, défini ainsi $R \subseteq L \times (T_{\Sigma,E}(X))^2$; chaque règle est un couple d'éléments, le premier est une étiquette, le second est une paire de classes d'équivalence de termes $T_{\Sigma,E}(X)$ sur la signature (Σ,E), modulo les équations E, avec $X = \{x_1, \dots, x_n, \dots\}$ un ensemble infini et dénombrable de variables **[Mem C].**

3.3. Les systèmes de réécriture conditionnels:

En ajoutant des conditions à l'application des règles de réécriture, on étend naturellement les systèmes de réécriture à des systèmes de réécriture conditionnels**.** Un système de réécriture conditionnel naturel(natural conditional rewriting system) a des règles de réécriture de la forme R ([t], [t'], $C_1,\dots,C_K$), la notation suivante est utilisée:

R : [*t*] → [*t'*] **if** *C1*∧……∧ *Ck.*

Où une règle ***R*** exprime que la classe d'équivalence contenant le terme ***t*** peut se réécrire en la classe d'équivalence contenant le terme ***t'*** si la condition de la règle *C1*∧...∧ *Ck.* est vérifiée. Cette dernière est appelée condition de la règle et peut être abrégée par la lettre ***C***, et la règle de réécriture, dans ce cas, est dite conditionnelle. La partie conditionnelle d'une règle peut être vide, dans ce cas les règles sont appelées règles de réécriture inconditionnelles et sont notés par : R : [t] → [t′]

Une règle de réécriture peut être paramétrée par un ensemble de variables ***{ x₁… xn }*** qui apparaissent soit dans ***t, t'*** ou ***C***, et nous écrivons:

R: [t(x1, … , xn)] → [t'(x1, … , xn)] **if** C(x1, … , xn) **[Mem E].**

4. Règles de déduction

Le calcul dans un système concurrent est une séquence de transitions (règles de réécriture) exécutées à partir d'un état initial donné. Il correspond à une preuve ou une déduction dans la logique de réécriture. Cette déduction est intrinsèquement concurrente et permet de raisonner correctement sur l'évolution du système d'un état à un autre **[Mes 90] .**

4.1.Règles de déduction:

Etant donné une théorie de réécriture R,= (Σ, E,L, R), nous disons que la séquence [t] →[t'] est prouvable dans R et on écrit R ├ [t] →[t'] si et seulement si [t] →[t'] est obtenue par une application finie des règles de déduction suivantes:

1. *La réflexivité* :

Pour chaque terme [t] $\in T\sum,E(X)$, $\overline{[t] \rightarrow [t]}$ où $T\sum,E(X)$ est l'ensemble des Σ termes avec variables construits sur la signature Σ et les équations E.

2. *La congruence :*

Pour chaque fonction $f \in \sum_n$, $n \in N$.

$$\frac{[t_1] \rightarrow [t'_1] \quad \ldots\ldots \quad [t_n] \rightarrow [t'_n]}{[f(t_1 \ldots\ldots t_n)] \rightarrow [f(t'1 \ldots\ldots t'n)]}$$

3. Le remplacement :

pour chaque règle r : $[t(x_1 \ldots\ldots x_n)] \rightarrow [t'(x_1 \ldots\ldots x_n)]$ dans R :

$$\frac{[w_1] \rightarrow [w'_1] \quad \ldots\ldots \quad [w_n] \rightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w}'/\overline{x})]}$$

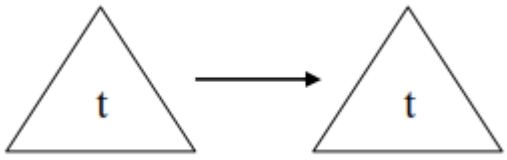
Sachant que $t(\overline{w}/\overline{x})$ dénote la substitution simultanée de xi par w_i dans t avec $\overline{x}$ représentant $x_1 \ldots\ldots x_n$ **[MR 04].**

4. La transitivité :

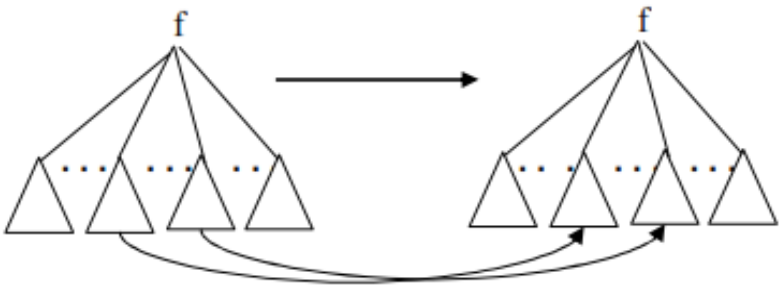
$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

4.2. Représentation graphique des règles de déduction:

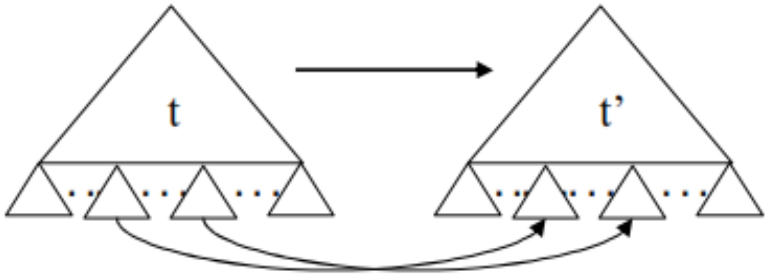
Réflexivité:



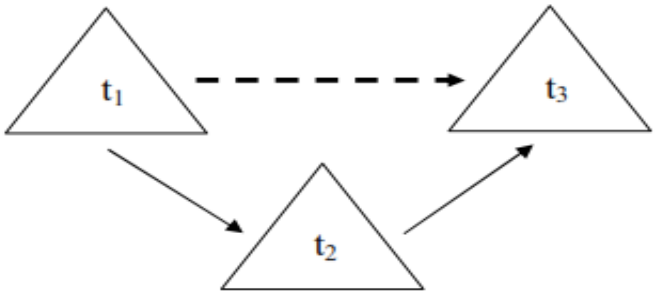
2.Congruence:



3.Remplacement:



4.Transitivité:



Exemple 1 :

Soit la théorie de réécriture suivante exprimant l'ensemble des booléens tel que :

$\sum$ = ({bool}, {T: →bool, F : →bool, _and_ : bool × bool→bool,})

E = {x and y = y and x, x and (y and z) = (x and y)and z}

L = {l0, l1}

R = {l0 : [T] →[T and T], l1 : [F] →[F and T]}

Etant donné la formule suivante :

[F and F] →[F and T and F and T and T]

1) Nous pouvons prouver cette formule dans la théorie proposée en utilisant la règle de réécriture l1 comme prémisse pour la règle de déduction congruence pour l'opération _and_ ∈ ∑ :

$$\frac{[F] \rightarrow [F \text{ and } T] \quad [F] \rightarrow [F \text{ and } T]}{[F \text{ and } F] \rightarrow [F \text{ and } T \text{ and } F \text{ and } T]}$$ ***Congruence***

2) Nous allons prendre la règle l_0 et la réflexivité comme prémisse pour la règle de déduction congruence avec la même opération _and_ ∈ ∑ :

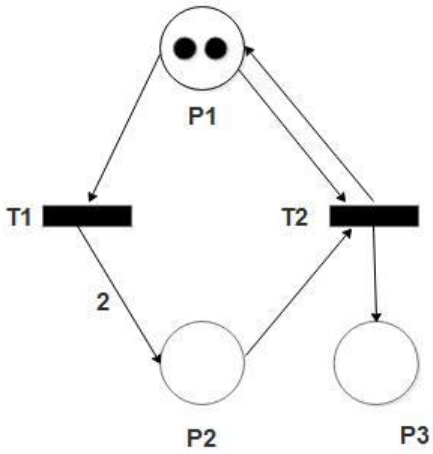
$$\frac{[F \text{ and } T \text{ and } F] \rightarrow [F \text{ and } T \text{ and } F] \quad [T] \rightarrow [T \text{ and } T]}{[F \text{ and } T \text{ and } F \text{ and } T] \rightarrow [F \text{ and } T \text{ and } F \text{ and } T \text{ and } T]}$$ ***Congruence***

3) Ces deux dernier résultats seront utilisés comme des prémisses pour la règle de déduction transitivité pour obtenir le résultat attendu :

$$\frac{[FandF] \rightarrow [FandT \; andFandT] \quad [Fand \; TandFandT] \rightarrow [FandTand \; FandTandT]}{[F \text{ and } F] \rightarrow [F \text{ and } T \text{ and } F \text{ and } T \text{ and } T]}$$ ***Transitivité***

Exemple 2 :

Un exemple de système concurrent intégré au niveau de la logique de réécriture est montré à travers le réseau de Petri ordinaire suivant :



Ce réseau de Petri peut être représenté par la théorie de réécriture suivante :

Rdp= (∑, E, L, R), tels que:

∑= ({place, marking}, {p1, p2, p3 : →place, null:→ place, _ : place → marking, _._: marking × marking → marking})

E = {x.null = x, x.y = y.x, x.(y.z) = (x.y).z} .

L = {T1, T2}

R = {T1 : [p1] → [p2. p2]:, T2 : [p1.p2] → [p1.p3]}

Dans cette théorie de réécriture, les places et le marquage du réseau de Petri sont spécifiés par le biais des sortes place et marking. Ainsi, l'ensemble des opérations suggérées sert à générer les places et le marquage du réseau. L'ensemble des règles de réécriture permet de décrire les transitions du réseau, la transition T1 permet de consommer un jeton de la place p1, et de générer deux jetons au niveau de la place p2. La transition T2 permet de consommer un jeton de la place p1 et un autre de la place p2, et de générer un jeton au niveau de la place p1 et un autre dans la place p3.

- Un comportement simple de ce réseau peut être habituellement vu par son évolution de l'état [p1.p1], par exemple, vers un autre état [p1. p3. p3]

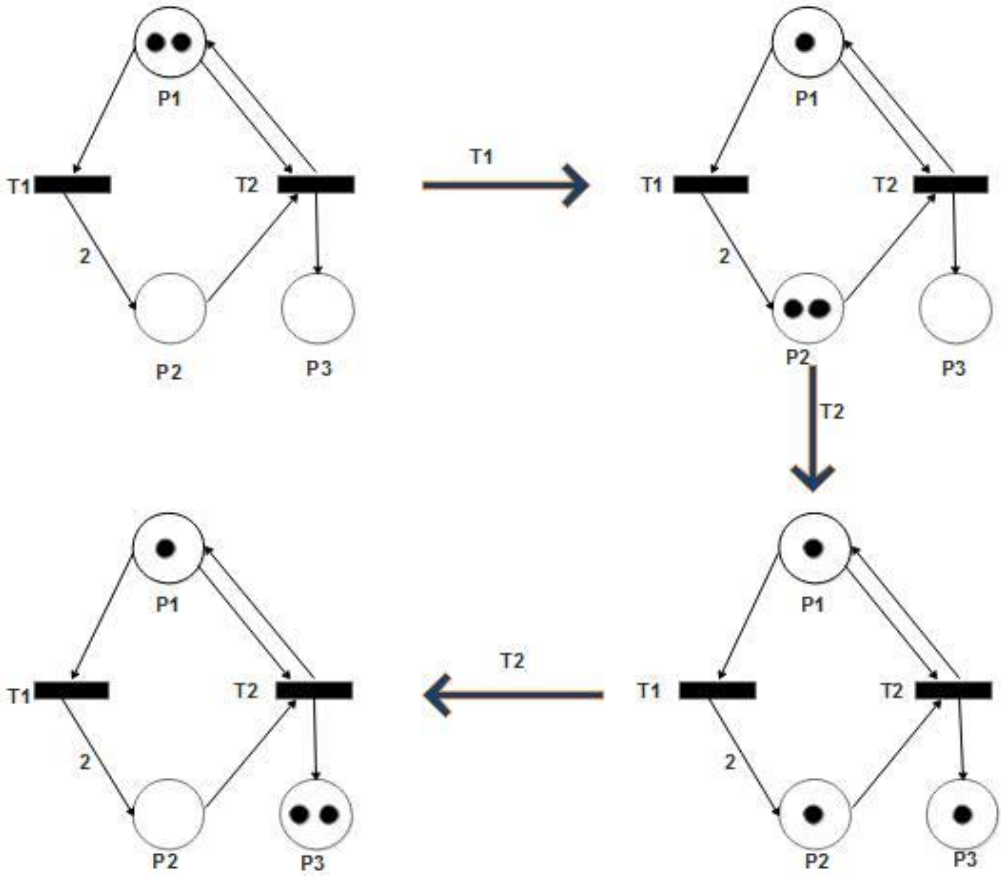


Figure : Un exemple3 d'évolution de réseau de Petri

Ce comportement est déduit par la preuve : Rdp $\vdash [p_1.p_1] \rightarrow [p_1.\ p_3.\ p_3]$

Ayant comme terme de preuve :

$$(p_1.\ T1)\ ;\ (T2.\ p_2)\ ;\ (T2.\ p_3)$$

noté généralement :

$$T1\ ;\ T2\ ;\ T2$$

Et obtenu ainsi :

$$\frac{P_1\colon [P_1] \rightarrow [P_1] \quad T1\colon [P_1] \rightarrow [P_2.P_2]}{P_1.T1 :[P_1.P_1] \rightarrow [P_1.P_2.P_2\ ]} \quad \ldots\ldots\ldots\ldots\ldots\ldots \textbf{*Congruence *}$$

$$\frac{T2\colon [P_1.P_2] \rightarrow [P_1.P_3] \quad P_2\colon [P_2] \rightarrow [P_2]}{T2.P_2 :[P_1.P_2.P_2] \rightarrow [P_1.P_3.P_2\ ]} \quad \ldots\ldots\ldots\ldots\ldots\ldots \textbf{*Congruence *}$$

$$\frac{P_1.T1\colon [P_1.P_1] \rightarrow [P_1.P_2.P_2\ ] \quad T2.P_2 :[P_1.P_2.P_2] \rightarrow [P_1.P_3.P_2]}{(P_1.T1);(T2.P_2) :[P_1.P_1] \rightarrow [P_1.P_3.P_2\ ]} \quad \ldots\ldots\ldots\ldots\ldots\ldots \textbf{* Transitivité *}$$

$$\frac{P_3\colon [P_3] \rightarrow [P_3] \quad T2 : [P_1.P_2] \rightarrow [P_1.P_3]}{T2.P_3 :[P_1.P_2.P_3] \rightarrow [P_1.P_3.P_3]} \quad \ldots\ldots\ldots\ldots\ldots\ldots \textbf{*Congruence *}$$

$$\frac{(P_1.T1);(T2.P_2)\colon [P_1.P_1] \rightarrow [P_1.P_3.P_2\ ] \quad T2.P_3:[P_1.P_2.P_3] \rightarrow [P_1.P_3.P_3]}{(P_1.T1);(T2.P_2)\ ;(T2.P_3) :[P_1.P_1] \rightarrow [P_1.P_3.P_3]} \quad \textbf{* Transitivité *}$$

Nous constatons bien qu'a ce niveau, le comportement d'un système concurrent (modéliser dans notre cas par un réseau de Petri) est décrit formellement par des déductions dans cette logique.

5. Langages Basés sur la Logique de Réécriture:

La programmation déclarative a eu le mérite de faciliter considérablement les aspects conceptuels de la tâche de programmation. Cependant, la logique sur laquelle se base généralement les langages déclaratifs ne prend pas en charge les caractéristiques inhérentes aux systèmes réels, notamment l'action et le changement.

Pour mettre en évidence ce constat, Meseguer a défini un langage, appelé MAUDE, basé sur une logique d'action qui est la logique de réécriture. Ce langage implémente et concrétise les différents concepts de la logique de réécriture. Dans la section suivante, nous présentons en

détails ce langage **[Clav 04] .** Il existe bien d'autres langages à base de la logique de réécriture, les plus connus sont :

- **ELAN** fournit un environnement pour les systèmes de spécification et prototypage des déductions dans un langage basée sur des règles commandées par des stratégies. Son but est de soutenir la conception des preuves de théorèmes.
- **CAFEOBJ** est une nouvelle génération des langages de spécifications algébriques qui a été développée au Japon. L'équipe de CAFEOBJ est guidée par le prof. Kokichi Futatsugi, à JAIST (Japon), qui est un des fondateurs originaux d'OBJ2. CAFEOBJ est un langage de spécifications algébrique *exécutable*, et un successeur moderne d'OBJ, il est prévu pour les spécifications des systèmes réels, la vérification formelle des caractéristiques, prototypage rapide, ou même de programmation.
- **MAUDE** Le langage Maude est un langage de programmation formel et déclaratif basé sur la théorie mathématique de la logique de réécriture Il a été développé ainsi que la logique de réécriture par 'José Meseguer' et son groupe dans le laboratoire d'informatique en SRI International. C'est un langage de haut niveau et un système de haute performance qui supporte deux types de calcul celui de la logique équationnelle et de la logique de réécriture pour une large gamme d'applications **[Mem E].**

5.1 Système Maude :

Maude est un langage déclaratif autour duquel est construit un système performant. Les programmes Maude sont des théories de réécriture et les calculs concurrents dans Maude représentent des déductions dans la logique de réécriture. Les buts du projet de Maude est de supporter les spécifications formelles exécutables, et d'élargir le spectre d'utilisation de la programmation déclarative, spécifier et réaliser des systèmes de haute qualité dans des secteurs comme : le génie logiciel, réseaux de communication, l'informatique répartie, bio-informatique, et le développement formel d'outils **[Mem E].**

5.2. Différents Modules de MAUDE:

En Maude, un module fournira une collection de sortes et une collection d'opérations sur ces sortes, ainsi que les informations nécessaires pour réduire et de réécrire des expressions entrées par l'utilisateur dans l'environnement Maude.

Il y a trois types de module dans Maude pour la spécification des systèmes **[Mem E]**:

1) Modules fonctionnels,
2) Modules systèmes et,

3) Modules orientés objet.

Nous allons essayer de présenter brièvement chaque module:

5.2.1. Modules Fonctionnels:

Les modules fonctionnels définissent les types de données et les opérations qui sont utilisés par les équations.

Un module fonctionnel est introduit par les mots clés **fmod** <*Corps du module*> **ndfm** où le corps du module spécifie une théorie ($\sum$, $E \cup A$, Φ) dans la logique équationnelle d'appartenance.

La signature $\sum$ inclut des sortes (indiqués par le mot clé **`sort`**), des sous-sortes (spécifiés par le mot clé **`subsort`**) et des opérateurs (introduits avec le mot clé **`op`**).

La syntaxe des opérateurs est définie par les utilisateurs en indiquant la position des arguments par le symbole (_). L'ensemble ***E*** désigne les équations et les tests d'appartenance (qui peuvent être conditionnels) et ***A*** est un ensemble d'axiomes équationnels introduits comme attributs de certains opérateurs dans la signature $\sum$.Les équations sont spécifiées par le mot clé **`eq`** ou le mot clé **`ceq`** (pour les équations conditionnelles) et les tests d'adhésion ou d'appartenance sont introduits avec les mots clés **`mb`** ou **`cmb`** (pour les tests d'appartenance conditionnels).

Dans un module fonctionnel, les équations sont utilisées comme des règles de simplification par lesquelles chaque expression, après substitution des variables, peut être évaluée et simplifiée à sa forme réduite dite forme canonique. Le résultat de la simplification d'un terme initial est unique quel que soit l'ordre d'application des équations. Les variables peuvent être déclarées dans les modules avec les mots clés **`var`** ou **`vars`**, ou introduites directement dans les équations et les tests d'adhésion, sous la forme d'une expression `var : sort`.

Exemple 1

Reprenons l'exemple 2 des réseaux de Petri président Nous pouvons maintenant lui associer d'abord un module fonctionnel:

```
fmod  PN-SIGNATURE  is
 sorts  PLACE  MARKING .
  ops  p1 p2  p3: → Place .
  op  nill : → Place .
  op _ : Place → Marking .
  op _._ : Marking  Marking → Marking .
```

```
 vars  X  Y  Z : Marking .
  eq  X . nill = X .
  eq  X . Y = Y . X .
  eq  X . (Y . Z) = (X . Y) . Z .
endfm.
```

5.2.2. Modules systèmes:

Les modules systèmes sont des théories de réécriture .Ils permettent de spécifier le comportement d'un système concurrent. Les modules systèmes ajoutent à la définition des modules fonctionnels un ensemble de règles de réécritures (conditionnelles et inconditionnelles). Ils sont introduits par les mots clés ***mod*** <*Corps du module*> **endm** où le corps du module spécifie une théorie de réécriture R = *(* $\sum$ *, E*$\cup$ *A, Φ, R*). Les règles de réécriture R sont introduites avec les mots clés **rl** ou **crl.** Elles sont spécifiées dans Maude avec la syntaxe :

`crl` [*l*] : *t* => *t'* **`if`** *cond* .

Si la règle est non conditionnelle, le mot clé **crl** est remplacé par **rl** et la clause « **if** *cond* » est omise.

Exemple 2

Nous reprenons l'exemple 1, pour lequel nous ajoutons des règles de réécriture associées aux transitions de réseau de Petri pour décrire son comportement**.**

```
mod PETRI-NET is
Extending PN-SIGNATURE .
rl p1 => p2 . p2.
rl p1.p2 => p1.p3 .
endm
```

Le module PN-SIGNATURE de l'exemple1 est importé par le module système PETRI-NET.

5.2.3. Modules orientés objet:

les systèmes concurrents orientés objet peuvent être définis par le biais des modules orientés objets présentés par le mot-clé omod... Endom utilisant une syntaxe plus pratique que celle des modules systèmes. Particulièrement tous les modules orientés objet incluent implicitement le module CONFIGURATION +. Bien que les modules système de Maude soient suffisants pour la spécification des systèmes orientés objets, il y a des avantages conceptuels importants fournis par la syntaxe de modules orientés objets. Ces derniers

permettent à l'utilisateur de penser et exprimer ses idées en orientés l'objet. Les modules orientés objets sont transformés en des modules systèmes pour des buts d'exécution **[Mem C].**

Exemple 3:

Soit le module orienté objet suivant :

```
(omod CPT is
protecting QID .
protecting INT .
subsort Qid < Oid .
class Cpt | som : Int .
msgs credit debit : Oid Int -> Msg .
msg de_à_transferer_ : Oid Oid Int -> Msg .
vars A B : Oid .
vars M N N' : Int .
rl [credit] :
credit(A, M)
< A : Cpt | som : N > => < A : Cpt | som : N + M > .
crl [debit] : debit(A, M) < A : Cpt | som : N > => < A : Cpt |
som : N - M > if N >= M .
crl [transferer] : (de A à B transferer M) < A : Cpt | som : N
> < B : Cpt | som : N' > => < A : Cpt | som : N - M > < B :
Cpt | som : N' + M > if N >= M .
endom)
```

Cet exemple décrit un compte bancaire, avec les changements qui peuvent surgir sur ce compte. Nous avons une classe appelée Cpt (spécifiant le compte) qui possède un attribut appelé som (présentant la somme actuelle du compte) de type entier. Cette classe possède trois méthodes (messages), la première appelée crédit sert à ajouter une autre somme au compte, la deuxième est appelée debit sert à retirer une somme du compte, la dernière est appelée de_à_transferer_ qui sert à transférer une somme d'un compte à un autre compte. Le déroulement de ces méthodes est exprimé par les règles de réécriture données au niveau du module CPT.

Pour déclarer les classes nous utilisons le mot clé class suivi par le nom de la classe, et puis nous trouvons une barre verticale après laquelle nous pouvons trouver les noms des attributs avec ses types. La déclaration des messages est similaire à celle des opérations, mais nous utilisons msg ou msgs au lieu de op et ops. La syntaxe d'un objet est la suivante :

<nom de l'objet : sa classe | nom de attribut1 : sa valeur,…, nom de attributn : sa valeur>

5.3 Conclusion

Ce chapitre a été consacré pour la présentation des concepts de base de la logique de réécriture. C'est un cadre formel de raisonnement correct et d'analyse d'une grande variété de systèmes concurrents en particulier, ayant des états et évoluant en termes de transitions. Ses différents concepts sont implémentés à travers l'environnement Maude**.**

Deuxième Partie

Conception et Implémentation

Chapitre 03

Modélisation et Conception

Chapitre 03

Modélisation et Conception

1. Introduction :

Dans ce chapitre nous allons présenter la modélisation et la conception de l'outil de génération automatique de spécification Maude. En effet, notre application n'est pas d'une grande complexité et par conséquent nous n'avons utilisé la totalité des diagrammes d'UML.
Dans notre cas, nous commençons par le diagramme de cas utilisation, puis pour chaque cas d'utilisation un diagramme de séquences et diagramme Etats-Transitions seront présentés. Enfin, le diagramme de classes a été présenté afin d'éclairsir l'apect structurel de l'outil.

2. Présentation de langage UML :

UML (Unified Modeling Language, que l'on peut traduire par "langage de modélisation unifié") est constitué de 13 diagrammes standardisés pour modéliser l'aspet structurel et comportemental des systèmes. En effet, ce langage est né de la combinaison de plusieurs méthodes existantes, et est devenu la référence en terme de modélisation objet. **[Grady B].**

3. La conception de l'application

3.1.Description générale

L'objectif principal de notre travail est de transformer de réseaux de Petri vers une spécification en utilisant la logique de réécriture. La première phase est de construire un éditeur simple de réseaux de Petri, puis a partir de cet éditeur on dessine le modèle du système représenté à l'aide d'un réseau de Petri. La deuxième phase consiste à construire la matrice d'incidence du réseaux de Petri dessiné précédemment. Ensuite, cette matrice sera utilisé pour faire la transformation vers la logique de réécriture en respectant la syntaxe du langage Maude.

3.2.les règles de transformation

3.2.1 Réseau de Petri vers la logique de réécriture

Le principe à utiliser pour la transformation de réseaux de Petri vers la logique de réécriture peut être résumé dans les étapes suivantes :

- Le nombre de règles de réécritures est égale au nombre de transitions composant le réseau de Petri.
- Si on suppose que la forme générale d'une règle de réécriture est comme suit :

  `rl [étiquette] : LHS => RHS .`

  - Etiquette : cette étiquette représente le nom de la transition.
  - LHS : la partie gauche de la règle (left-hand side)
  - RHS : la partie droite de la règle (right-hand side).

  Pour chaque transition, on met les noms des places d'entrée dans la partie gauche (LHS) et les noms des places de sortie dans la partie droite(RHS). Bien sûr, si un arc a un poids différent de 1, on répète le nom de la place lui connecté autant de fois que ce poids.

Exemple

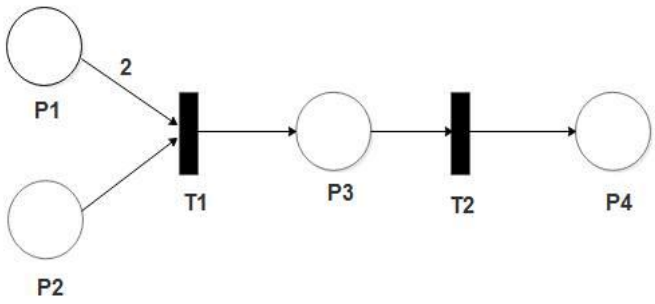


Figure 1 : réseaux de Petri

- ce réseaux de Petri possède deux transition donc on aura deux règles de réécriture**.**
- Les règles de réécritures de ce réseau de Petri sont :

  rl [T1] : P1 P1 P2 => P3.

  rl [T2] : P3 => P4.

3.2.2 Matrice d'incidence vers la logique de réécriture :

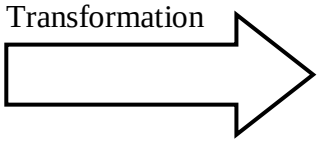
D'autre part, on peut utiliser la matrice d'incidence pour faciliter la transformation automatique du réseaux de Petri vers la logique de réécriture. Dans ce cas, chaque colonne représente une transition où les éléments positifs sont des entrée et doivent être mis dans la partie gauche de la règle (LHS), et les élément négatifs sont des sorties et doivent être mis dans la partie droite (RHS).

Cette procédure est illustrée dans l'exemple suivant :

Exemple :
On construire la matrice d'incidence du réseaux de Petri précédent Figure 1 comme suit:

	T1	**T2**
P1	2	0
P2	1	0
P3	-1	1
P4	0	-1

Matrice d'incidence



rl [T1] : P1 P1 P2 => P3.
rl [T2] : P3 => P4.

Les règles de réécriture

3.3 Les modèles (Templates) des Modules de spécification Maude

La spécification complète du réseau de Petri de la Figure 1 sera représenté dans les deux modules Maude suivants :

```
fmod PETRI_NET is
sorts Place Marking .
subsorts Place < Marking .
op empty : -> Marking .
ops p1 p2 p3 p4 : -> Place .
op _ _ : Marking Marking -> Marking
[assoc comm id: empty ].
endfm
```

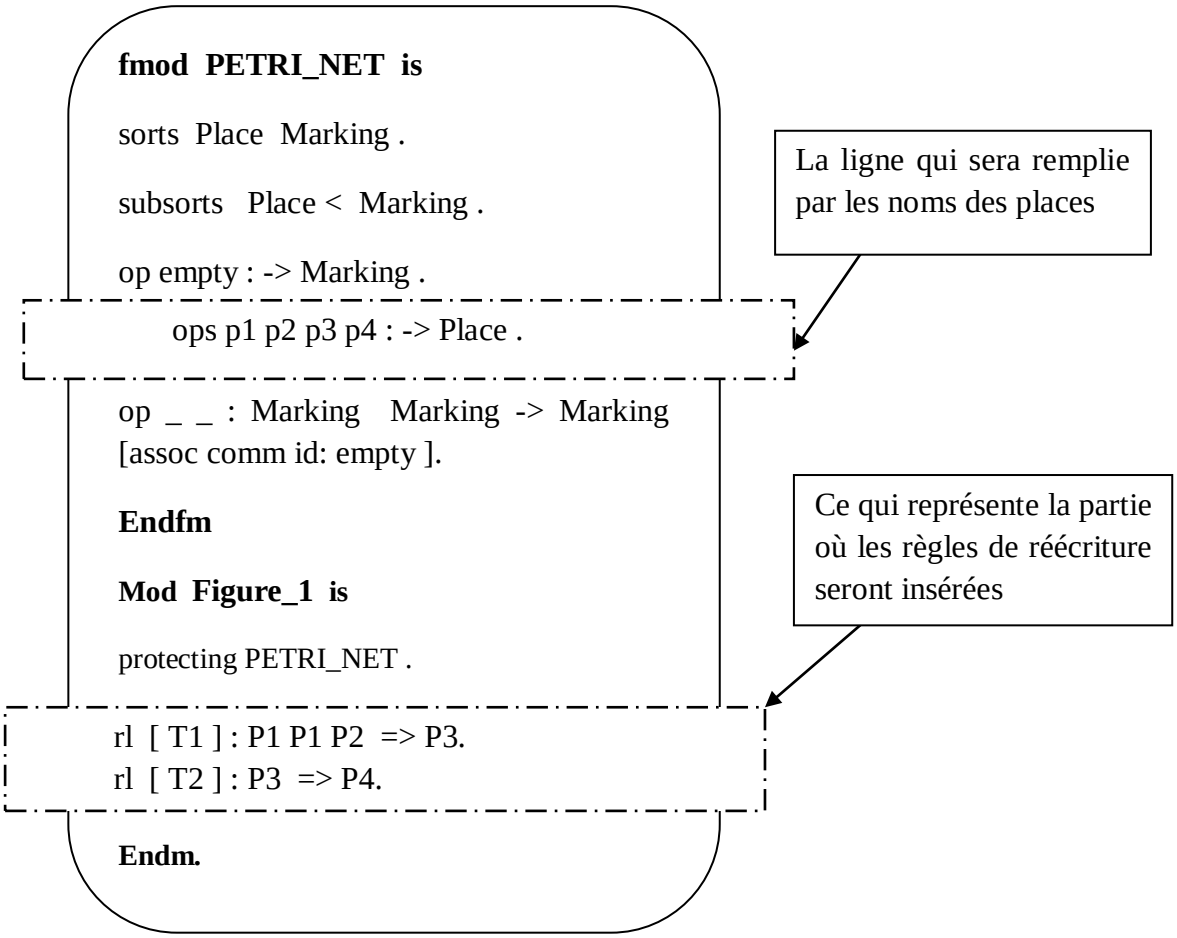
Module fonctionnelle

```
mod Figure_1 is
protecting PETRI_NET .
    rl [ T1 ] : P1 P1 P2 => P3.
    rl [ T2 ] : P3 => P4.
endm
```

Module système

D’après ces deux modules, on constate qu’ il y a des parties constantes dans la spécification de tous les réseaux de Petri. Donc , ces partie constitueront les modèles qui seront utilises ultérieurement dans notre application.

Template Module fonctionnelle et module système



3.4 Les diagrammes utilisés

3.4.1. Le diagramme des cas d'utilisation

L'ensemble des cas d'utilisation décrient exhaustivement les exigences fonctionnelles et techniques du système. Chaque cas d'utilisation correspond donc à une fonction métier du système, selon le point de vue d'un de ses acteurs. Dans notre application l'utilisateur dessine un réseau de Petri dans le but de le transforme vers la logique de réécriture.

Les cas d'utilisations de notre système sont représentés par la **figure 1** suivante:

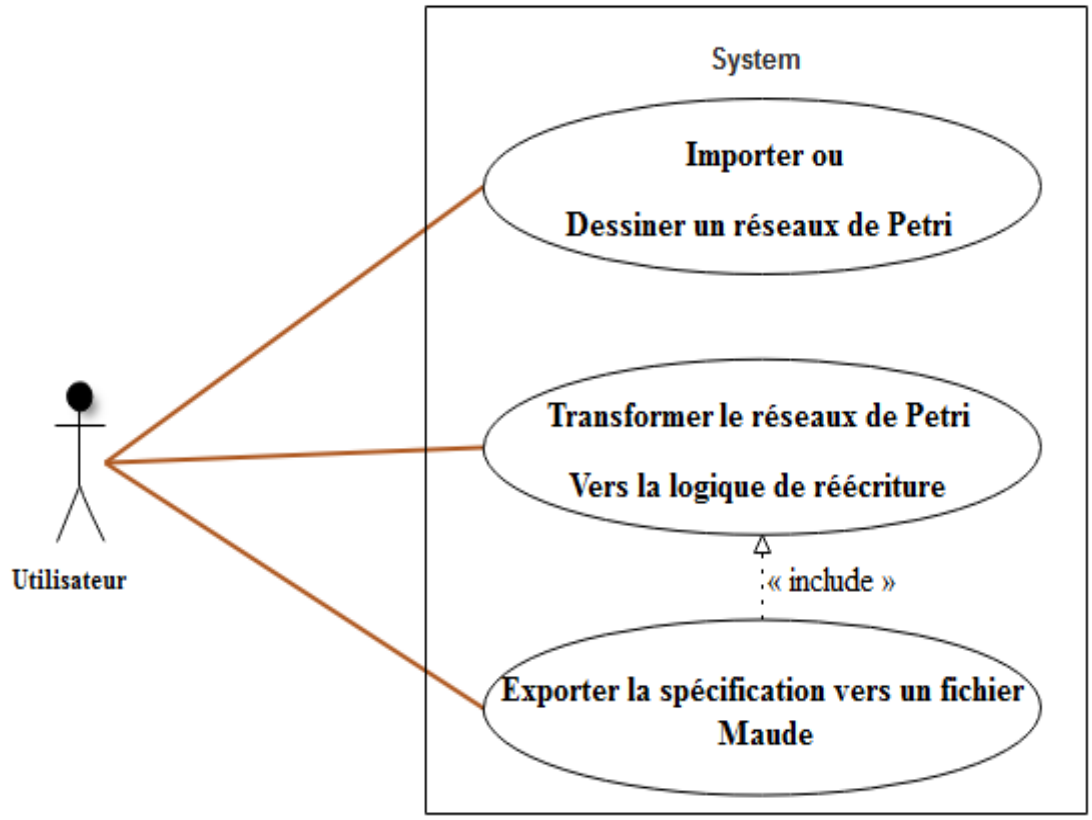


Figure 2: Diagramme cas d'utilisation

3.4. 2.Diagramme de séquences :

Les principales informations contenues dans ce diagramme sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Autrement dit, le diagramme de séquence permet de représenter des collaborations entre les objets selon un point de vue temporel et peuvent servir à illustrer des cas d'utilisation.

3.4.2.1. Diagramme de séquence «Dessiner un réseau de Petri » :

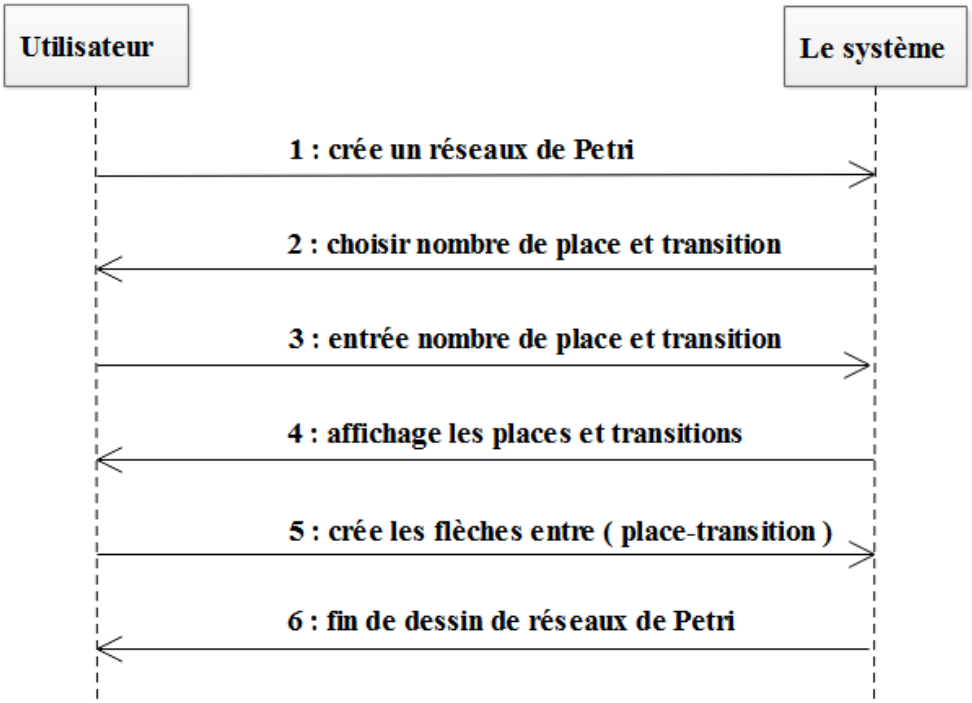


Figure 3: Diagramme de séquence (Dessiner un réseau de Petri) .

- **Le scenario :**
  - l'utilisateur demande pour crée un réseau de Petri.
  - Le système demande d'entrée le nombre de place et nombre de transition.
  - l'utilisateur choisir le nombre de place et nombre de transition.
  - Le système afficher les place et les transition.
  - L'utilisateur crée les liens entre les place et les transitions.
  - Fin de dessin de réseaux de Petri.

3.4.2.2. Diagramme de séquence «Importer un réseau de Petri » :

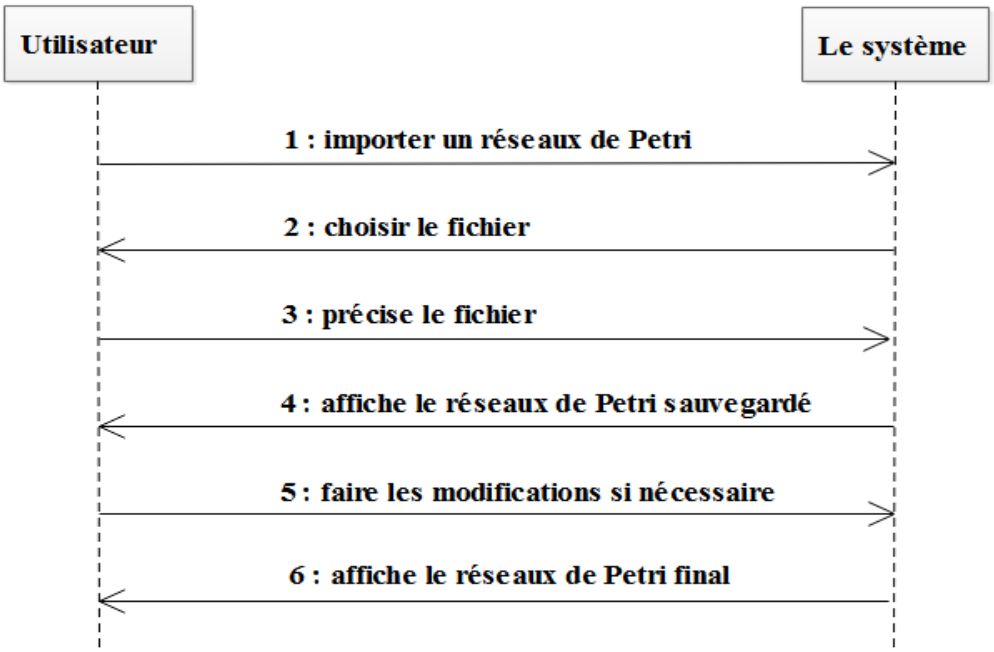


Figure 4: Diagramme de séquence (Importer un réseau de Petri)

- **Le scenario :**
  - l'utilisateur demande pour importer un réseau de Petri.
  - Le système demande de choisir le ficher à importer.
  - l'utilisateur précise le ficher à importer.
  - Le système affiche le réseau de Petri sauvegarder .
  - L'utilisateur faire les modification si nécessaire.
  - Le système affiche le réseau de Petri final .

3.4.2.3. Diagramme de séquence «Exporter la spécification vers un fichier Maude » :

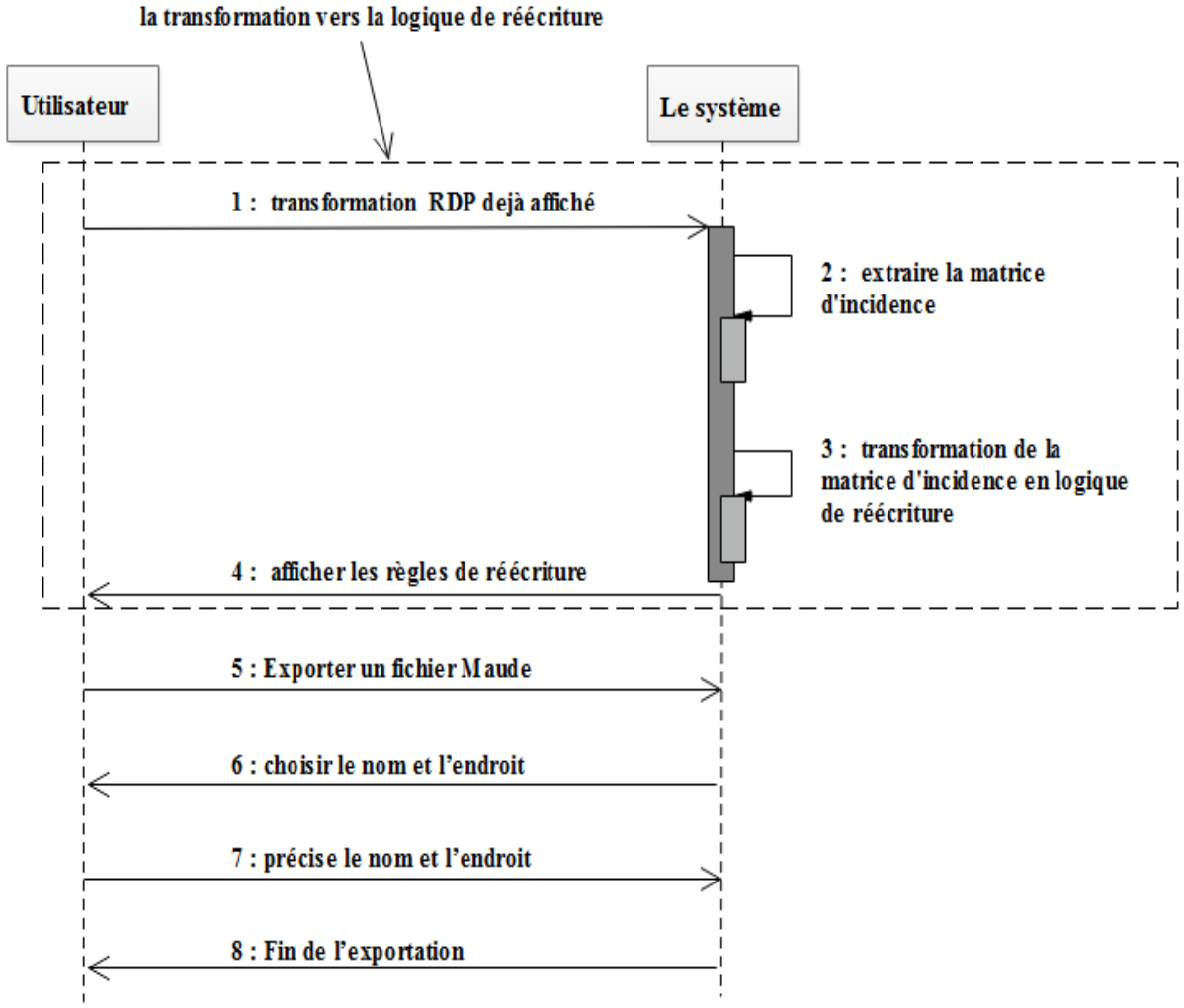


Figure 5: Diagramme de séquence «Exporter la spécification vers un fichier Maude»

- **Le scenario :**
  - l'utilisateur demande de transformer du réseaux de Petri vers les règles de réécriture.
  - Le système extraire la matrice d'incidence.
  - Le système transforme la matrice d'incidence en logique de réécriture.
  - Le système affiche les règles de réécriture .
  - L'utilisateur demande pour exporter un fichier Maude.
  - Le système demande de choisir le nom et l'endroit de fichier.
  - L'utilisateur précise le nom et l'endroit de fichier à exporter.
  - Le système fini l'exportation de fichier.

3.4. 3.Diagramme d'états transition :

3.4. 4.Diagramme de classe :

Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. Ce diagramme fait partie de la partie statique d'UML car il fait abstraction des aspects temporels et dynamiques [wiki].

Le diagramme de classes montre la structure interne du système. Il permet de fournir une représentation abstraite des objets du système qui vont interagir pour réaliser les cas d'utilisation. Le diagramme de classes de notre conception est présenté dans la figure 6.

4. Conclusion

Dans ce chapitre, on présenté la conception de notre outil. Car les fonctionnalités sont assez claires, on a utilisé seulement le diagramme de cas d'utilisation et le diagramme de séquences pour présenter la partie dynamique. La partie statique a été illustrée par le diagramme de classes. dans le chapitre suivant, on va présenter leur implémentation.

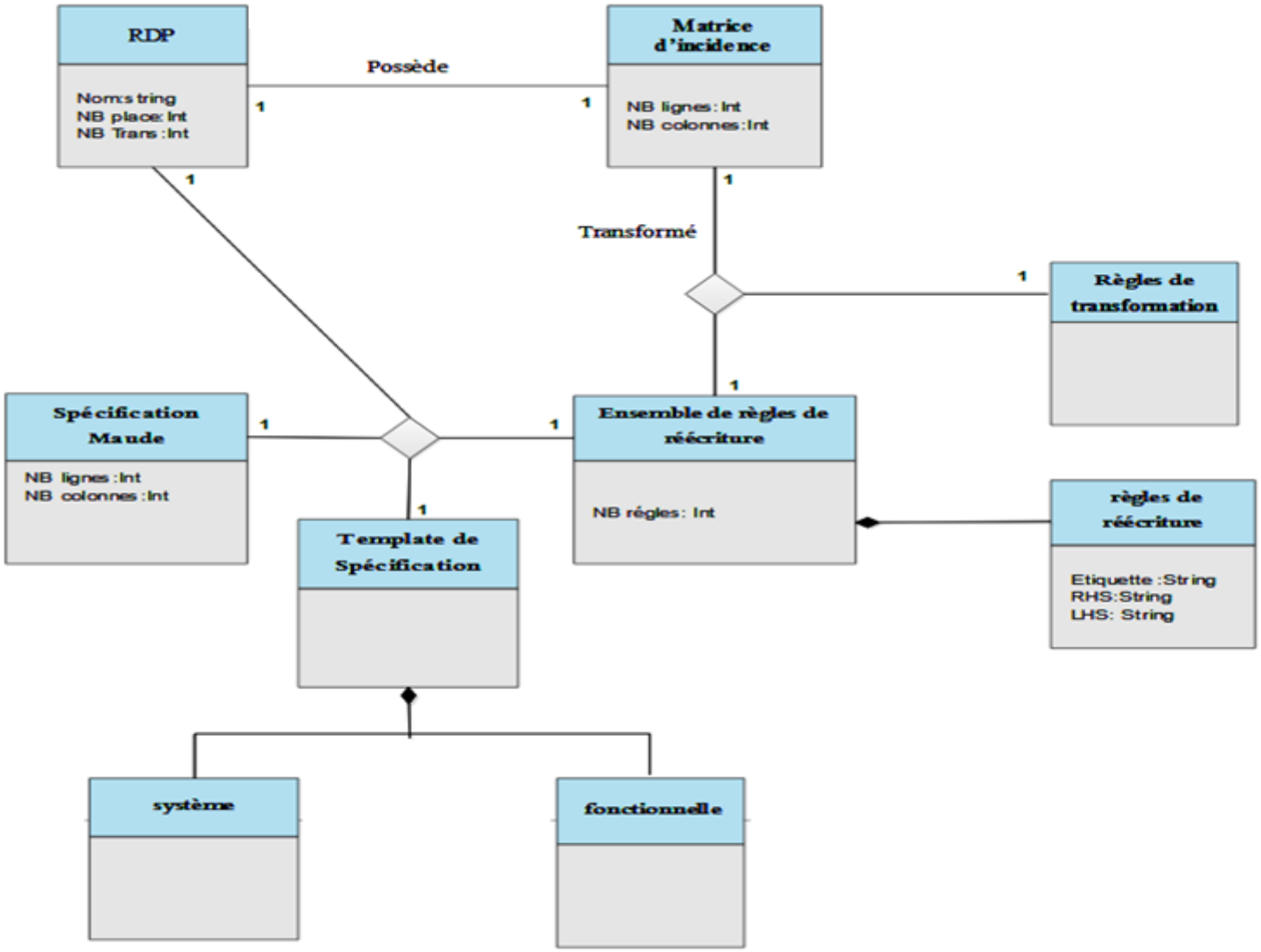


Figure 6. Diagramme de classe de l'outil de génération automatique de spécification Maude

Chapitre 04
Implémentation

Chapitre 04

Implémentation

1. Introduction :

Après avoir présenté dans le chapitre précédent la conception. Nous allons voir dans ce qui suit la mise en œuvre de notre application. On commence tout d'abord par une présentation des outils de programmation choisis tel que le langage de programmation, Et ensuit nous allons décrire l'interface de notre application.

2. Le langage de programmation :

Le langage de programmation utilisé pour implémenter notre application est L'Embarcadero Delphi XE3, édité par la société Embarcadero (anciennement Borland puis CodeGear). Bien sûr, c'est un langage de programmation orienté objet, fonctionnant sous Windows. Delphi apparut alors comme une alternative viable pour beaucoup de développeurs souhaitent créer des programmes standards sous Windows.

3. Description de l'application :

En plus du travail demandé dans notre projet, on a réalisé un éditeur simple pour les réseaux de Petri afin d'éviter l'utilisation d'autres outils intermédiaires . Ensuite, on a essayé de créer une interface graphique aussi simple que possible regroupant toutes les étapes d'exécution de notre application fin de faciliter leur utilisation.

Après le lancement de l'application, la fenêtre principale va apparaître comme suit :

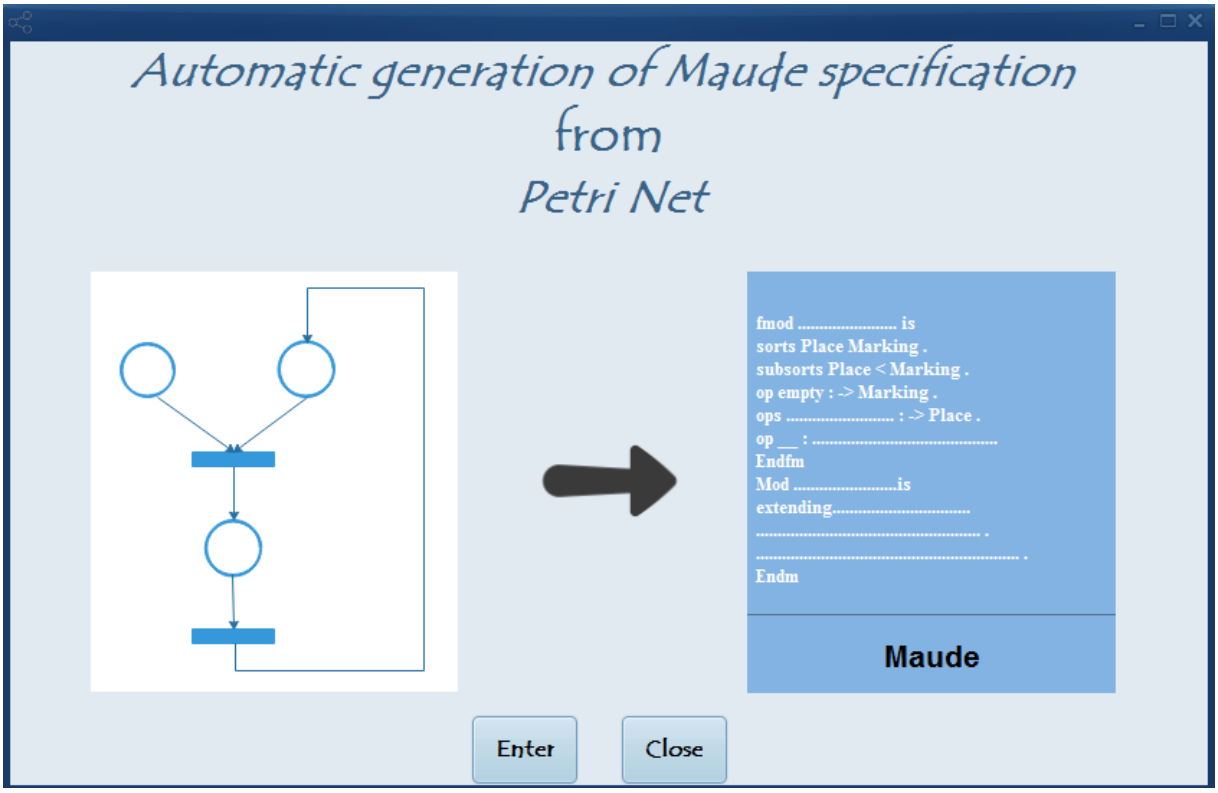


Figure 1: Fenêtre principale de l'application

Le bouton "Enter" permet de lancer l'éditeur de réseaux de Petri comme suit:

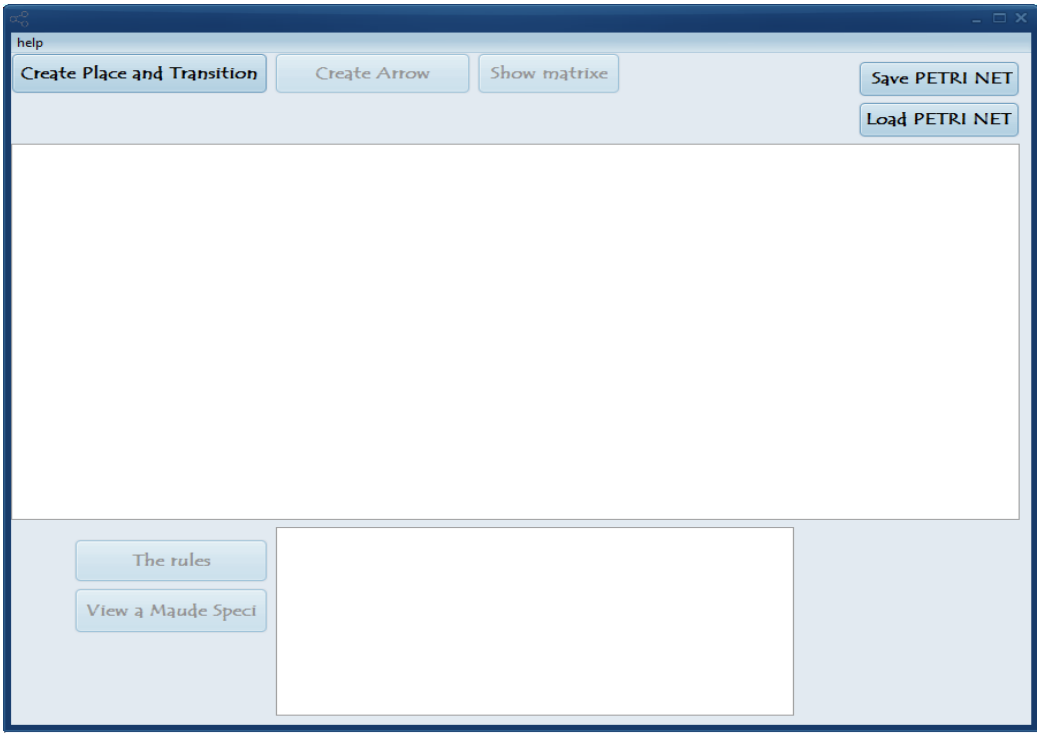


Figure 2: Editeur de réseaux de Petri

Cette fenêtre contient les buttons concernant la création de réseaux de Petri .

- ✓ **Create Place and transition :** ce bouton permet de créer les places et les transition du réseaux de Petri.
- ✓ **Save PETRI NET :** ce bouton permet de sauvegarder un réseau de Petri déjà créé .
- ✓ **Load PETRI NET:** ce bouton permet d'importer un réseau de Petri déjà sauvegardé.
- ✓ **Show Matrixe :** ce bouton permet d'afficher la matrice d'incidence correspondante.

Le clic sur le bouton " Create Place and transition " permet d'afficher une nouvelle fenêtre pour déterminer le nombre des places et transitions composant le réseau de Petri comme suit:

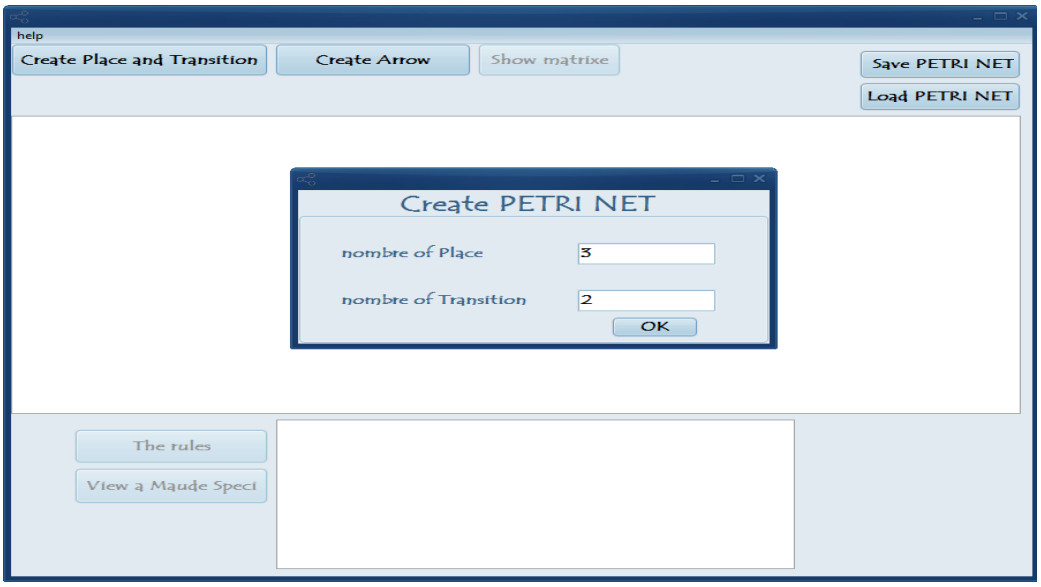


Figure 3: créer un réseau de Petri

- ✓ **Create Arrow :** ce bouton permet de créer les liens entre les places et les transition du réseaux de Petri comme le fenêtre suivante :

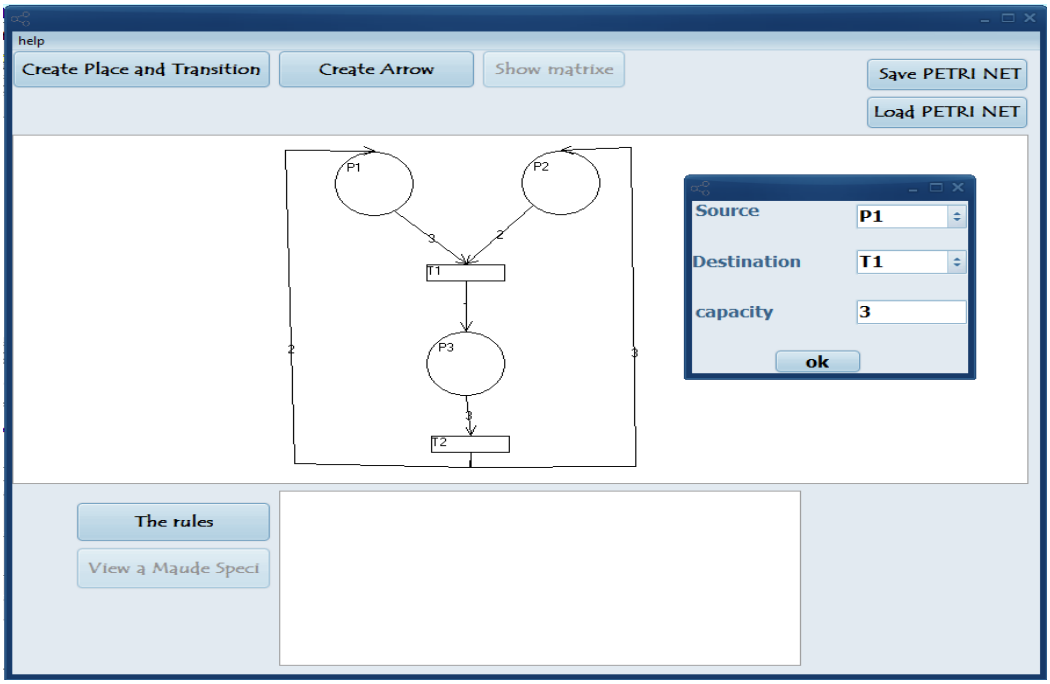


Figure 4: créer les liens entre place-transition

- ✓ **The rules :** ce bouton permet d'afficher les transitions (changements) locales dans le systèmes en terme des règles de réécriture comme suit:

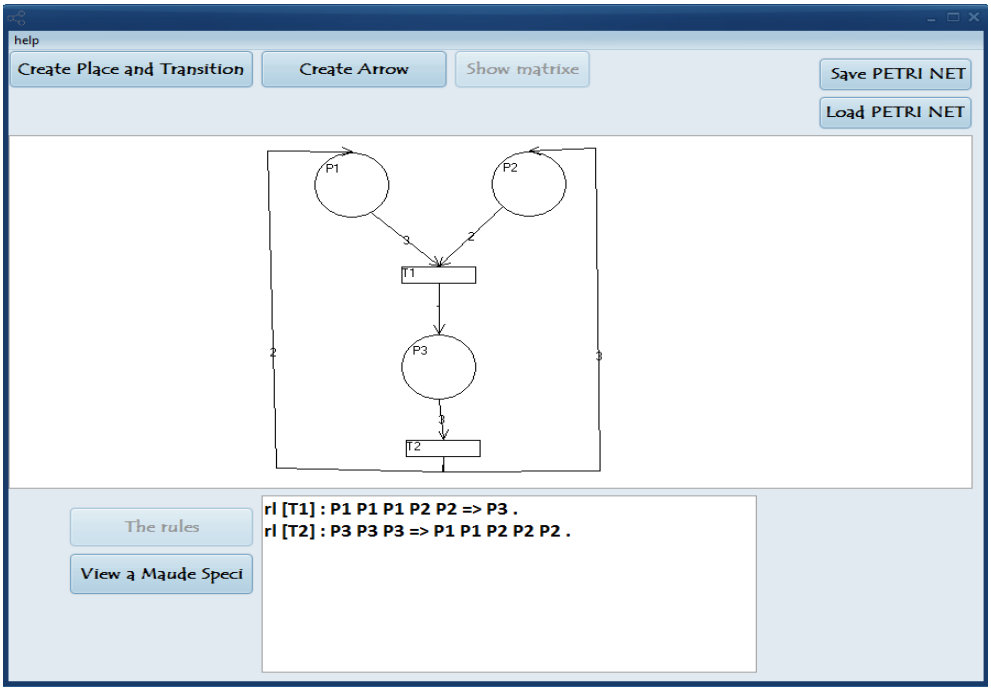


Figure 5:Les règles de réécriture

- ✓ **View a Maude Speci:** ce bouton permet d'afficher la spécification complète correspondante en Maude :

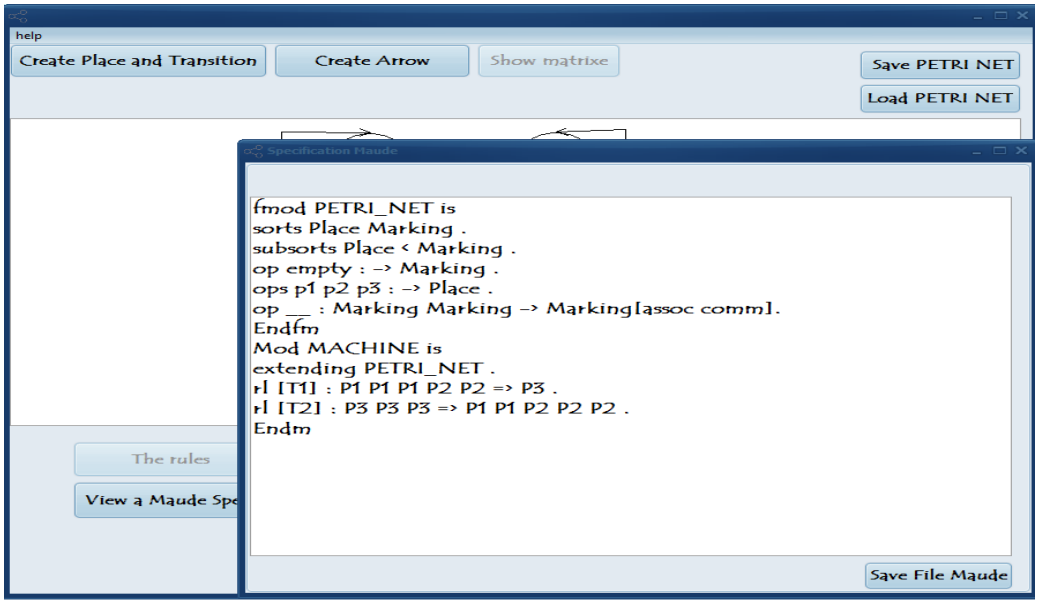


Figure 6: Spécification Maude

✓ **Save File Maude :** cette buttons permet d'exporter la spécifications Maude dans un fichier comme le fenêtre suivante :

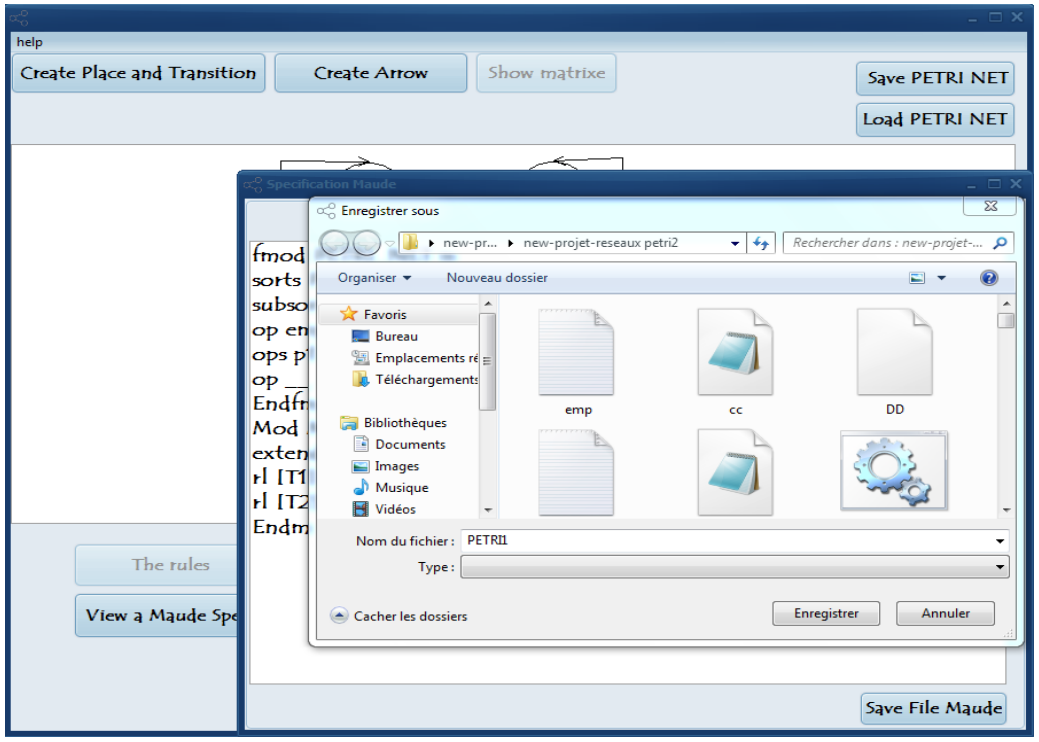


Figure 7: Enregistrer la spécifications Maude

Conclusion générale et Perspectives

Dans le but d'aider les développeur à construire des systèmes fiables et sûre, on a réalisé un outil de génération automatique de spécification Maude à partir des modèles de réseaux de Petri. L'opération de génération automatique de spécification permet d'une part de gagner du temps surtout dans le cas des systèmes de grande taille en terme de nombre de places et transitions. D'autre part, elle permet d'éviter les erreurs causées par le développeur lui même.
A ce moment, nous avons l'intention de rédiger un article expliquant les fonctionnalité de notre outil qui n'a pas été réalisé et de l'enrichir pour générer des spécifications Maude à partir des réseaux de Petri de haut niveau