REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DEMENTOURI CONSTANTINE

FACULTE DES SCIENCES DE L'INGENIEUR

DEPARTEMENT INFORMATIQUE

Thèse de Doctorat en Sciences en Informatique

Thème

Modélisation Multi-Paradigme: Une Approche Basée sur la Transformation de Graphes

Présenté par : **El-hillali Kerkouche**

Date de soutenance: 04 / 07 /2011

Composition du Jury

Pr. Djamel Eddine Saidouni	Université Mentouri Constantine	Président
Dr. Allaoua Chaoui	Université Mentouri Constantine	Rapporteur
Pr. Elbay Bourennane	Université de Bougogne, Dijon France	Co-Rapporteur
Dr. Mohamed Khireddine Kholladi	Université Mentouri Constantine	Examinateur
Dr. Djamel Meslati	Université d'Annaba	Examinateur
Dr. Brahim Belattar	Université de Batna	Examinateur

Dédicaces

A

La mémoire de mon père

A

Ma chère mère

A

Mes chères sœurs

A

Ma petite sœur Affaf

A

Mes beaux frères

A

Ma nièce: Roufaida

A

Mes neveux: Mohamed, Amine et Hamza

A

Tous mes amis

Omar

Remerciements

Mes premiers remerciements vont à mon encadreur Dr. Allaoua CHAOUI, MC. à l'université Mentouri de Constantine, pour son dévouement total durant la réalisation de ce travail, et surtout pour ses précieux conseils qu'il ma prodigué.

Je remercie le Professeur El-Bay Bourennane du LE2I-Dijon pour son aide précieuse.

Je tiens également à exprimer toute ma grande gratitude aux membres de jury :

Pr. Djamel Eddine Saidouni, président de jury

Dr. Mohamed Khireddine Kholladi

Dr. Djamel Meslati

Dr. Brahim Belattar

d'avoir accepté de juger ce travail.

Je tiens à remercier tous mes collègues et amis qui m'ont soutenue toutes ses années.

ET MA MÈRE pour tout ce qu'elle a fait et ce qu'elle fait encore pour moi.

E.Kerkouche

ملخـــــص

العمل المقدم في هذه المد كرة يخص المساهمة في مجال هندسة البرامج بواسطة النماذج على العموم و التحويل بين النماذج باستخدام قواعـد البيانـات على وجه الخصوص. الهدف الرئيسي لهذه الأعمال هو دراسة "التصميم المتعددة النماذج" بالإضـافة إلى اقتراح طرق تقوم على تحويل النماذج لتسهيل استخدام تقنيات تحليل في دورة تطوير النظم المعقدة. المساهمات الخاصة تتعلق باقتراح أدوات و برامج التصميم تمكن من: أولا، مساعدة مطوري النظم المعقدة، وثانيا لزيادة الجودة واتساق النماذج المختلفة للنظام حتى تلبي توقعات مستخدميها. على وجه التحديد، يتم تقديم ثلاث مساهمات. المساهمة الأولى تكمن في تطوير الأداة الذي تمكن من الاستنتاج الآلي للوصف الموافق في لغة Maude إنطلاقا من نماذج ECATNets. المساهمة الثانية هي اقتراح أداة رسمية و برنامج تصميم وتحليل نظم برامج حاسوبية معقدة باستخدام نماذج Petri . المساهمة الثالثة هي المسـاهمة في إضفاء الطابع الرسمي على نماذج UML من خلال نماذج Petri.

كلمـــات مفتـاح : هندسة البرامج بواسطة النماذج, التصميم المتعددة النماذج, تحويل البيانـات, قواعد البيانـات,الطرق التحليلية, نمادج Petri , $AToM^3$.

Résumé

Les travaux présentés dans ce manuscrit s'inscrivent dans domaine de l'Ingénierie Dirigée par les Modèles en général et dans la Transformation de Modèles à l'aide des Grammaires de Graphes en particulier. Ils ont pour objectif principal l'étude de la "Modélisation Multi-paradigmes" ainsi que la conception d'une approche basée sur la Transformation de Graphes destinée à faciliter l'utilisation des techniques d'analyse formelle au cours du cycle de développement des systèmes complexes. Les contributions consistent en la proposition d'environnements de modélisation capables: d'une part, à assister les développeurs des systèmes complexes, d'autre part à augmenter la qualité sémantique et la cohérence de différentes représentations graphiques du système de manière à mieux répondre aux attentes de leurs utilisateurs. Plus précisément, trois contributions sont présentées. La première contribution qui consiste en la proposition d'un outil basé sur la méta-modélisation et une grammaire de graphe qui assure la génération automatique des descriptions équivalentes en Maude à partir de modèles ECATNets. La deuxième contribution est la proposition d'une plate-forme formelle et un outil graphique pour la spécification et l'analyse des systèmes logiciels complexes en utilisant les G-Nets. L'approche proposée est basée sur la Méta-modélisation et les Grammaires de Graphes. L'outil proposé permet d'exploiter les différents langages et techniques formels dans un cadre uni dont le but est de cumuler leurs avantages. La troisième contribution est un effort dans la formalisation du langage UML par le biais des réseaux de Petri colorés. Plus précisément, un outil basé sur la Méta-modélisation et les Grammaires de Graphes pour transformer un digramme Etats-Transitions et un diagramme de collaboration en un modèle réseaux de Perti colorés afin de vérifier les propriétés comportementales des systèmes. L'analyse formelle des réseaux de Petri colorés est réalisée à l'aide de l'analyser INA.

Mots Clés: Ingénierie Dirigée par les Modèles, Modélisation Multi-paradigme, Méta-modélisation, Transformation de Graphes, Grammaires de Graphes, Méthodes Formelles, Réseaux de Petri, AToM^3.

Abstract

The results presented in this manuscript are a contribution in the field of Model Driven Engineering in general and Model Transformation using Graph Grammars in particular. Their main objective is the study of "Multi-paradigms Modeling" and the design of an approach based on Graph Transformation to facilitate the use of formal analysis techniques in the development cycle of systems complex. Own contributions consist of modeling environments can: firstly, to assist developers of complex systems, secondly to increase the semantic quality and consistency of different graphical representations of the system to better meet the expectations of their users. Specifically, three contributions are presented. The first contribution was the development of a tool based on Meta-modeling and Graph Grammar that provides automatic generation of the descriptions equivalent Maude from ECATNets models. The second contribution is the proposal of a formal framework and a graphical tool for specification and analysis of complex software systems using G-Nets. The proposed approach is based on Meta-modeling and Graph Grammars. The proposed tool allows to use different languages and formal techniques in a unified framework whose goal is to combine their advantages. The third contribution is an effort in formalizing the UML through colored Petri nets. More specifically, a tool based on Meta-modeling and Graph Grammars to transform a set of statechart diagrams and collaboration diagram into a Petri Nets model in order to verify the behavioral properties of systems. The formal analysis of colored Petri nets is performed using the INA analyzer.

Keywords: Model Driven Engineering, Multi-paradigms Modeling, Meta-modeling, Graph Transformation, Graph Grammar, Formal Methods, Petri Nets and AToM^3.

Table des Matières

Chapitre 02 : Intégration des Méthodes Formelles à l'IDM

Table des Figures

Introduction générale

Les travaux présentés dans ce manuscrit s'inscrivent dans le contexte de l'Ingénierie Dirigée par les Modèles (*IDM*). Ils ont pour objectif principal l'étude du domaine de recherche baptisé "Modélisation Multi-paradigme" ainsi que la conception d'une approche basée sur la transformation de graphes destinée à faciliter l'utilisation des techniques d'analyse formelle au cours du cycle de développement des systèmes complexes.

Problématique

Les systèmes complexes sont caractérisés, non seulement par un grand nombre de composants mais aussi par la diversité de ces composants. Ceci implique que ces composants sont modélisés dans différents langages de modélisation ou formalismes. Les diagrammes d'entité-association, les réseaux de Petri, les Automates et les diagrammes UML sont communément utilisés dans ce cadre. La vérification des propriétés de ces systèmes est reconnue comme un problème difficile et se heurte à plusieurs problèmes. Un premier obstacle apparaît au niveau du choix des techniques de modélisation ou langages de modélisation utilisés pour modéliser les différents composants du système. Si le langage de modélisation est trop expressif, alors on ne peut pas, mathématiquement, l'analyser de manière automatique. Un second obstacle est lié à la vérification du comportement global du système. Les modèles qui représentent un tel système sont modélisés avec des langages ou des formalismes différents, ce qui rend tout raisonnement global sur le système difficile.

Les recherches sur ce sujet ont donné naissance récemment à un domaine de recherche, "la Modélisation Multi-Paradigme", basé principalement sur les concepts du domaine de l'ingénierie dirigée par les modèles. L'approche de modélisation Multi-paradigme cherche à apporter de nouvelles fonctionnalités aux différents modèles qui sont utilisés pour concevoir et analyser ces systèmes. Un des points clés dans cette approche est la possibilité de transformer les modèles, d'un espace de modélisation ou d'un niveau d'abstraction vers un autre. Son objectif est de faciliter l'utilisation conjointe de différents modèles qui sont utilisés pour représenter et/ou analyser les différentes propriétés ou aspects du système pendant le cycle de développement.

En pratique, le compromis de trouver un modèle qui soit à la fois suffisamment expressif et raisonnablement analysable n'est pas facile à trouver. Les modèles de spécification utilisés dans l'industrie sont en général informels ou semi-formel et présentent trop d'ambiguïté pour être vérifiés automatiquement. Les modèles formels ne sont pas assez expressifs et/ou théoriquement difficiles à maîtriser par les non spécialistes. Des travaux existants ont essayé de réduire cet écart en proposant des associations de modèles informels ou semi-formels vers des modèles formels qui supportent la vérification formelle.

Il s'avère donc très intéressant de définir des outils pour la modélisation et la vérification efficace du comportement des systèmes complexes. Ils serviront comme support à l'élaboration de modèles et à la création de nouveaux espaces de modélisation plus adaptés aux besoins de l'analyse et de la vérification.

D'abord utilisée principalement dans le domaine des systèmes logiciels, l'IDM est maintenant appliquée à la conception des systèmes complexes. Elle est vue comme une réponse possible aux nombreux enjeux de l'Ingénierie des Systèmes complexes :

- Augmentation de la productivité,
- Diminution des coûts et du temps de développement,
- Augmentation de la fiabilité (analyse, vérification,),
- etc.

Cette application s'est accompagnée d'un développement notable des techniques de définition de langages de modélisation, entrainant ainsi la multiplication des langages de modélisation dit "domaine-spécifique" à coté des langages de modélisation intégrés à des approches existantes. Dans le contexte de l'IDM, la définition d'un langage de modélisation a pris naturellement la forme d'un modèle appelé Méta-Modèle. La Méta-modélisation apporte donc la flexibilité nécessaire à la fourniture de moyens adaptés à la diversité des aspects/composants des systèmes complexes. Par ailleurs, la notion de transformation de modèles joue un rôle fondamental afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.). Les langages de modélisation et les modèles sont décrits comme des graphes, les transformations entre modèles sont effectuées par réécriture des graphes et peuvent être décrites aussi comme des graphes sous forme de grammaire de graphes.

Objectif (Contribution)

Le présent projet de thèse vise à créer de nouvelles techniques afin de mettre en place les outils facilitant le travail interdisciplinaire qui est au cœur de la recherche sur les systèmes complexes. En particulier, Il vise à étudier l'utilisation de techniques de vérification formelles et de simulation de différents modèles représentants les composants/aspects des systèmes complexes pour assurer la cohérence et la sûreté de leur fonctionnement.

Notre objectif principal est de proposer des environnements de modélisation capable d'une part d'assister les développeurs dans leurs modélisations des systèmes complexes avec des langages conviviaux et graphiques; d'autre part augmenter la qualité sémantique et la cohérence de différentes représentations graphiques du système de manière à mieux répondre aux attentes de leurs utilisateurs.

Organisation (Structure du manuscrit)

Dans le premier chapitre, nous présentons les concepts essentiels ainsi que la terminologie que nous utiliserons tout au long de ce manuscrit. Nous abordons les principes clés de l'ingénierie IDM et les différentes variantes d'ingénierie centrées sur les modèles. Nous verrons que la maîtrise de la sûreté de fonctionnement joue une part prépondérante dans la conception et le développement des systèmes complexes.

Au deuxième chapitre, nous présentons un panorama sur les méthodes formelles ainsi que leur classification. Ensuite nous abordons leur utilisation et leur intégration à la méthodologie IDM afin d'augmenter la fiabilité et de garantir l'absence d'erreurs de conception. Dans ce contexte, les réseaux de Petri représentent une technique formelle largement utilisée.

Le troisième chapitre est consacré à la présentation du domaine de la Modélisation Multi-Paradigme. Nous passons alors en revue les différents axes de recherche de ce domaine. Nous présentons également l'outil AToM^3: un outil de méta-modélisation et transformation de modèles.

Le quatrième chapitre présente notre support outillé pour manipuler et simuler les ECATNets. Nous avons proposé un outil basé sur la Méta-modélisation et une Grammaire de Graphe qui assure la génération automatique des descriptions équivalentes en Maude. Les étapes du simulateur des ECATNets sont décrites dans ce chapitre.

Dans le cinquième chapitre, nous présentons une plate-forme formelle et un outil graphique pour la spécification et l'analyse des systèmes logiciels complexes en utilisant les G-Nets. Notre approche est basée sur la Méta-modélisation et les Grammaires de Graphes. L'outil proposé permet d'exploiter les différents langages et techniques formels dans un cadre unifié dont le but est de cumuler leurs avantages.

Enfin, le dernier chapitre est consacré à la présentation de notre approche de formalisation du langage UML à l'aide des réseaux de Petri colorés. Nous présentons un outil basé sur la Méta-modélisation et les Grammaires de graphes pour transformer les digrammes d'états-transitions et le diagramme de collaboration en un modèle équivalent en réseaux de Perti colorés afin de vérifier les propriétés comportementales des systèmes. L'analyse formelle des réseaux de Petri colorés est réalisée à l'aide de l'analyseur INA.

La conclusion résume les points essentiels de ce travail et présente les perspectives de recherches suggérées par ce travail.

Chapitre 01 :

Ingénierie Dirigée par les Modèles

1.1 Introduction

Au cours des dernières décennies, le développement de grands projets d'ingénierie, toujours plus complexes, a mis en évidence la nécessité de disposer d'outils, de méthodes et de processus permettant d'en assurer la maîtrise tout au long de leur cycle de vie.

L'ingénierie dirigée par les modèles (IDM), d'abord utilisée principalement dans le domaine des systèmes logiciel, a permis plusieurs améliorations significatives dans le processus de développement de systèmes complexes en se concentrant sur des préoccupations plus abstraites autour des modèles utilisés que sur la programmation classique (le code). Il s'agit donc d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Un modèle est une abstraction, une simplification d'un système qui est suffisante, non seulement pour comprendre le système modélisé, mais également pour garantir son bon fonctionnement.

Dans la suite de ce chapitre, nous présentons les concepts essentiels ainsi que la terminologie que nous utiliserons tout au long de ce manuscrit. Nous abordons les principes clés de l'ingénierie IDM et les différentes variantes d'ingénierie centrées sur les modèles. Nous verrons que la maîtrise de la sûreté de fonctionnement joue une part prépondérante dans la conception et le développement des systèmes complexes.

1.2 Ingénierie Système : Notions & Concepts

1.2.1 Système Complexe

Suivant la définition de l'AFIS [AFIS], un *Système Complexe* est décrit comme un ensemble organisé d'éléments en interaction permanente entre eux. Cet ensemble forme un tout cohérent et intégré pour assurer une ou plusieurs fonctions correspondant à la finalité du système.

Un système est caractérisé par les propriétés qui résultent de l'*interaction* de ses éléments. Ces propriétés peuvent être qualifiées d'émergentes [Holland98] car ce sont des propriétés nouvelles obtenues au niveau du système du fait des synergies existant entre ses constituants. De telles propriétés émergentes peuvent être souhaitées ou indésirables.

L'objectif de l'activité de conception est d'obtenir le comportement global émergent recherché du système en maintenant les comportements émergents non intentionnels dans des limites considérées comme acceptables. Dans ce contexte, l'étude du système d'un point de vue comportemental est donc fondamentale.

Il existe plusieurs types de systèmes. Dans leur classification, D. Harel et A. Pnueli [Harel85] distinguent notamment les systèmes transformationnels, les systèmes interactifs et les systèmes réactifs. Les systèmes dits transformationnels sont des systèmes qui acquièrent des données, les traitent, produisent des sorties puis terminent. Les systèmes interactifs sont des systèmes qui interagissent avec leur environnement, à une vitesse qui leur est propre. Quant aux systèmes dits réactifs, ce sont des systèmes qui interagissent en permanence avec leur environnement, mais à une vitesse imposée par l'environnement. Ce couplage avec l'environnement, qui est par essence non totalement maîtrisable et non totalement prévisible, rend les systèmes réactifs particulièrement difficiles à concevoir.

La difficulté à concevoir un système est liée notamment à sa complexité. Celle-ci tient à au moins trois facteurs :

- **Le nombre et la nature de ses éléments.** Le nombre d'éléments peut être élevé. Il peut même être éventuellement variable au cours du temps et la nature de ces éléments peut être variée.
- **La nature de son organisation interne,** qui est liée aux relations qui existent entre ses éléments. Ceux-ci peuvent former des réseaux, des hiérarchies, etc … . L'organisation du système peut également varier au cours du temps.
- **Le couplage avec l'environnement.** Plus le système est en interaction forte avec l'environnement plus il est exposé à l'incertitude et à l'imprévisibilité de celui-ci.

1.2.2 Ingénierie Système

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

1.2.3 Modélisation des Systèmes Complexes

Dans le contexte de l'ingénierie système, le rôle principal des modèles est de permettre d'appréhender des systèmes dont la complexité est importante et d'en étudier des aspects particuliers avant même la mise en œuvre concrète du système [Lavagno03]. Les modèles permettent alors par exemple d'identifier des incertitudes quant à la spécification du système ou à l'adéquation d'une solution pour sa réalisation lorsque celles-ci sont trop complexes pour permettre un raisonnement immédiat. Par ailleurs, les modèles constituent également des supports pour

communiquer des idées au sujet de la conception du système entre les différentes parties impliquées dans le projet. Enfin, les modèles servent également de guides lors des phases d'implémentation du système. Dans ce contexte, les modèles utilisés permettent donc d'anticiper sur le comportement qu'aura le système final. Ils sont appelés *modèles prédictifs* par opposition aux modèles dits *explicatifs* dont le rôle est de donner une représentation la plus fidèle possible du monde réel. Les modèles prédictifs peuvent être des modèles physiques, structurels ou comportementaux. Les modèles physiques donnent des informations sur les caractéristiques physiques du futur système (largeur, hauteur, matériaux, etc.). Les modèles structurels sont utilisés pour définir l'organisation des différents éléments du futur système. Enfin, les modèles comportementaux ont pour objectif de spécifier le comportement dynamique du futur système.

Dans la suite de ce manuscrit, nous appelons *modèle* une représentation prédictive du système qui met en valeur des propriétés que l'on considère intéressantes par rapport à un objectif de conception donné. En ce sens, l'activité de *modélisation* correspond à la construction d'un modèle en tant que représentation d'un aspect du système pour un objectif de conception donné (étude d'une partie du système, analyse de propriétés particulières, simulation du comportement, etc.). Nous nous intéressons tout particulièrement dans ce manuscrit aux *modèles comportementaux*.

1.3 Ingénierie dirigée par les modèles (IDM)

L'Ingénierie IDM est une approche de développement mettant à disposition des outils, des concepts et des langages afin de simplifier et de mieux maitriser le processus de développement de systèmes qui ne cessent de croitre en complexité. D'autre part, elle permet aussi d'augmenter la productivité, la qualité, la réutilisabilité et l'évolution de ces systèmes.

1.3.1 Une approche autour des modèles

L'IDM est un domaine de recherche en pleine expansion aussi bien dans le monde académique que dans le monde industriel qui considère les modèles comme les éléments de base tout au long du processus de développement [Bézivin 04]. L'IDM raisonne entièrement à un haut niveau d'abstraction et non plus à celui des langages de programmation classiques. Une application sera alors générée en tout ou en partie, automatiquement ou semi-automatiquement à partir de modèles, en utilisant notamment des transformations successives de ces modèles. Il s'agit donc d'une forme d'ingénierie *générative*, le code source de l'application n'est plus considéré comme l'élément central d'un logiciel, mais comme un élément dérivé d'éléments de modélisation. Le

processus de conception est vu comme un ensemble de transformation de modèles. Cette approche adopte le principe: *"tout est modèle"* en analogie avec *"tout est objet"* dans la vision de l'approche orienté objet [Marvin68].

Dans cette optique, des outils permettant de construire et d'exploiter ces modèles ont été développés. Ces outils sont construits autour du concept de *Méta-modèle* qui permet de définir un langage de modélisation particulier à un domaine ou d'intégrer plus facilement plusieurs types de modèles. Ils intègrent aussi le concept de *transformation de modèles* pour pouvoir, par exemple, relier un modèle à une plate forme technologique spécifique ou générer du code. Le terme IDM [Kent02, Favre06] recouvre l'ensemble de ces disciplines, dans lesquelles les modèles jouent un rôle primordial.

1.3.2 Modèle, Langage de modélisation et Méta-modèle

*« Pour un observateur **A**, **M** est un modèle de l'objet **O**, si **M** aide **A** à répondre aux questions qu'il se pose sur **O** » [Marvin68]*

Un modèle est une abstraction et une simplification d'un système qu'il représente. Il offre donc une vision schématique d'un certains nombre d'éléments que l'on décrire sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions que l'on se pose sur lui.

Il faut noter qu'il reste toutefois difficile de répondre à la question "Qu'est ce qu'un bon modèle?". Néanmoins un modèle doit être suffisant et nécessaire pour permettre de répondre à certaines questions du système qu'il représente, exactement de la même façon que le système aurait répondu lui-même. Le modèle doit se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [Marvin68].

La notion de modèle dans l'IDM fait explicitement référence à la notion de *formalisme* ou de *langage de modélisation* bien défini. Un langage de modélisation est défini par une syntaxe abstraite, une syntaxe concrète et une sémantique. La syntaxe abstraite définit les concepts de base du langage. La syntaxe concrète définit le type de notation qui sera utilisé pour chaque concept abstrait qui peut être graphique, textuelle ou mixte. Enfin, la sémantique définit comment les concepts du langage doivent être interprétés par les concepteurs mais surtout par les machines.

En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé *Méta-modèle*.

Un méta-modèle est un modèle qui définit le langage d'expression d'un modèle. Autrement dit, le méta-modèle représente (modélise) les entités d'un langage, leurs relations ainsi que leurs contraintes, c'est-à-dire une spécification de la syntaxe du langage.

Le Méta-modèle à son tour est exprimé dans un langage de méta-modélisation spécifié par le Méta-Méta-modèle. Le langage utilisé au niveau du méta-méta-modèle doit être suffisamment puissant pour spécifier sa propre syntaxe abstraite et ce niveau d'abstraction demeure largement suffisant (méta-circulaire). Chaque élément du modèle est une *instance* d'un élément du méta-modèle.

Un modèle est dit *conforme* à un méta-modèle et constitue une *représentation* d'un système existant ou imaginaire. La relation entre un méta-méta-modèle et un méta-modèle est analogue à la relation entre un méta-modèle et un modèle. Cette relation est illustrée dans la Figure 1.1.

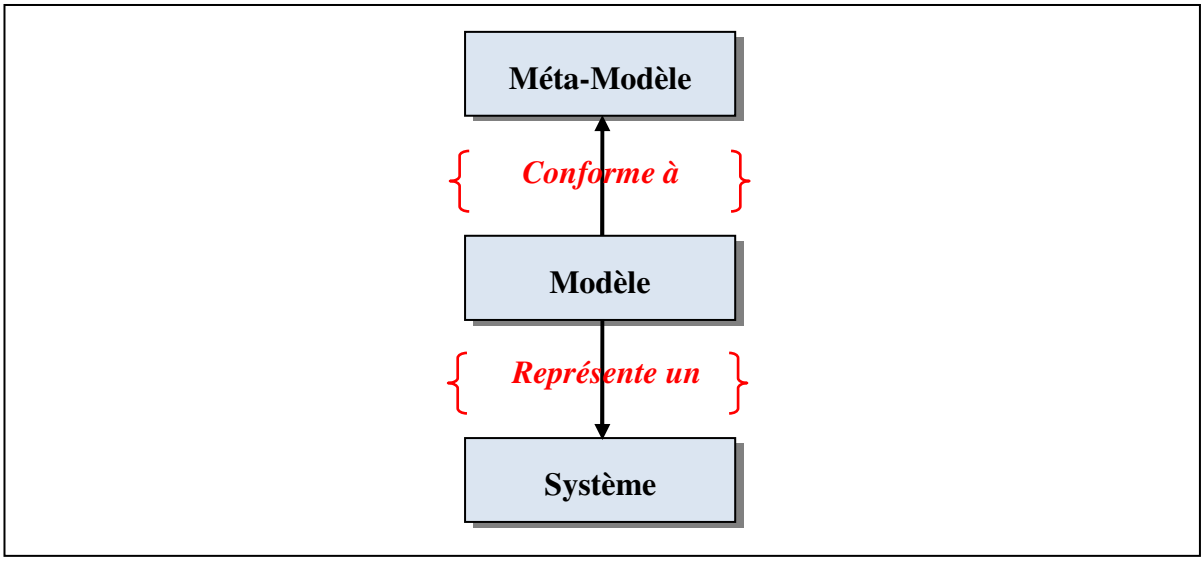


Figure 1.1 : Relation entre système, modèle et méta-modèle

1.3.3 Transformation de Modèles

La notion de transformation de modèles constitue l'élément central de la démarche IDM. En effet, cette notion porte sur l'automatisation de l'opération de transformation pendant le cycle de développement qui peut avoir des sémantiques différentes en fonction des utilisations: raffinement, optimisation, génération de code, etc.

La transformation de modèles est une opération qui consiste à générer un ou plusieurs modèles cibles conformément à leur méta-modèle à partir d'un ou de plusieurs modèles sources conformément à leur méta-modèle. Elle est qualifiée d'*endogène* si les modèles sources et cibles

sont conformes au même méta-modèle (source et cible sont dans le même espace technologique), sinon elle est dite *exogène* et elle se fait entre deux méta-modèles différents (source et cible sont dans deux espaces technologiques différents).

Dans la littérature, on peut distinguer trois types de transformations:

- ***Les transformations verticales***. La source et la cible d'une transformation verticale sont définies à différents niveaux d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Une transformation qui élève le niveau d'abstraction est appelée une abstraction.
- ***Les transformations horizontales***. Une transformation horizontale modifie la représentation source tout en conservant le même niveau d'abstraction. La modification peut être l'ajout, la modification, la suppression ou la restructuration d'informations.
- ***Les transformations obliques***. Une transformation oblique combine une transformation horizontale et une verticale. Ce type de transformation est notamment utilisé par les compilateurs, qui effectuent des optimisations du code source avant de générer le code exécutable.

De manière orthogonale à cette catégorisation, selon Czarnek [Czarnecki03], il existe deux grandes classes de transformation de modèles: les transformations de type Modèle vers code qui sont aujourd'hui relativement matures et les transformations de type modèle vers modèle qui sont moins maîtrisées. La deuxième classe de transformation fait l'objet de plusieurs recherches au cours de ces dernières années et plus particulièrement depuis l'apparition du MDA (Model Driven Architecture, voir la Section 4.1).

La transformation se fait par l'intermédiaire d'un ensemble de règles de transformations décrivant la correspondance entre les entités du modèle source et celles du modèle cible. La façon d'exprimer les règles de transformation peut être déclarative, impérative ou hybride. Il est à la charge de l'utilisateur de définir le langage de transformation qui répond le mieux à ses besoins et à ses compétences. Dans la spécification déclarative, les règles décrivent ce qu'on devrait avoir à l'issue d'un certains nombre d'éléments de modèle source. Par opposition à la spécification déclarative, la spécification impérative permet de décrire comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer. Enfin, la spécification hybride regroupe à la fois la spécification déclarative et la spécification impérative.

En réalité, la transformation se situe entre les méta-modèles source et cible. La transformation des entités du modèle source se fait en deux étapes:

- La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs méta-modèles, ce qui induit l'existence d'une fonction de transformation applicable à toutes les instances du méta-modèle source.
- La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé moteur de transformation ou d'exécution.

Une présentation générale des principaux concepts impliqués dans la transformation de modèles est illustrée dans la Figure 1.2.

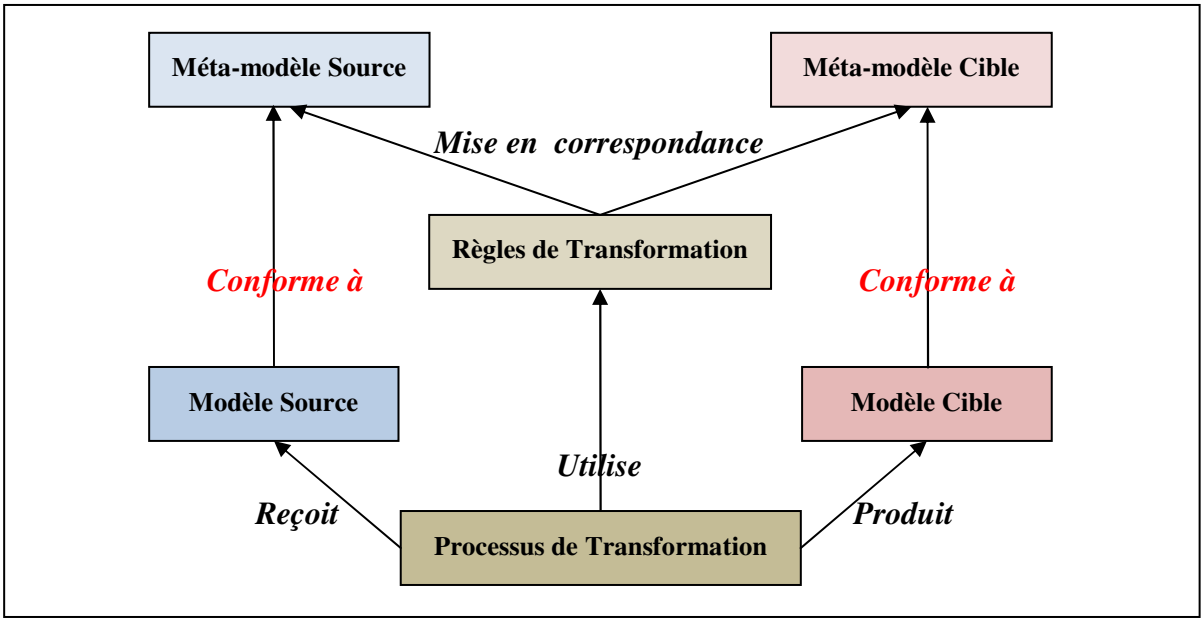


Figure 1.2: Concepts de base de la transformation de modèles

1.3.4 Techniques de transformation

On peut distinguer cinq techniques de transformation de modèles:

1.3.4.1 Manipulations directes

Ces approches se basent sur la représentation interne des modèles source et cible et sur une collection d'APIs pour les manipuler. L'implémentation des règles de transformation et leur ordonnancement restent à la charge du développeur en langage de programmation standard comme Java par exemple.

1.3.4.2 Approches relationnelles

Ces approches utilisent une logique déclarative reposant sur des relations d'ordre mathématique pour spécifier les relations entre les éléments des modèles source et cible par le biais

de contraintes. L'utilisation de la programmation logique est particulièrement adaptée à ce type d'approche.

1.3.4.3 Approches guidées par la structure

Dans ces approches, la transformation se fait en deux étapes. La première consiste à créer la structure hiérarchique du modèle cible, alors que la deuxième consiste à ajuster les attributs et les références dans le modèle cible.

1.3.4.4 Transformations de graphes

Les transformations de graphes, qui sont basées sur les grammaires de graphes [Gerber02], sont des techniques et des formalismes directement applicables à la transformation de modèles. Elles sont similaires aux approches relationnelles dans le sens où elles permettent l'expression des transformations sous une forme déclarative. Dans cette approche les modèles source et cible sont représentés sous forme de graphes. Cette notation visuelle permet aussi d'exprimer les règles de transformation sous forme graphique. Cette approche trouve son utilité dans le cas où les formalismes manipulés possèdent des syntaxes concrètes visuelles. Elle est formelle et bien fondée sur des bases mathématiques (comme la théorie des graphes et des grammaires formelles) ce qui permet de vérifier certaines propriétés de la transformation.

Cette approche vise à considérer l'opération de transformation comme un autre modèle conforme à son propre méta-modèle (lui-même défini à l'aide d'un langage de méta-modélisation). Une grammaire de graphe permet ainsi en tant que formalisme de modéliser une transformation [Vangheluwe02]. Une grande partie des outils récents adoptent cette approche avec certaines différences tels que : VIATRA, AToM^3, AGG et GReAT [Taentzer05].

1.3.4.5 Approches hybrides

Les approches hybrides sont une combinaison des différentes techniques. On peut notamment retrouver des approches utilisant à la fois des règles déclarative et impérative. ATL (ATLAS Transformation Language) [Jouault06] est un exemple de cette approche.

1.3.5 Qualité des modèles

Les qualités attendues d'un modèle dans le cadre de l'IDM sont nombreuses. Dans [Lavagno03], B.Selic liste quatre caractéristiques considérées comme essentielles. Selon lui, un modèle doit être:

1/ Abstrait : le modèle doit permettre d'omettre ou de cacher les détails que l'on ne souhaite pas considérer lors de l'étude d'une question, que ce soit pour des raisons de complexité ou de pertinence.

2/ Compréhensible : le modèle doit être compris par les personnes qui l'utilisent. Ce constat a des implications sur le formalisme à utiliser : d'une part, celui-ci doit être en adéquation avec l'objectif du modèle (notamment en termes de niveau d'abstraction, mais également en termes de type de représentation: graphique, textuelle, etc.) et, d'autre part, la sémantique du formalisme doit être communément admise.

3/ Fidèle et précis : le modèle doit représenter fidèlement les propriétés et les caractéristiques du système lorsqu'il est vu dans une optique spécifique.

4/ Prédictif : le modèle doit fournir les informations nécessaires et suffisantes pour permettre de faire des prédictions justes au sujet des propriétés du système.

1.3.6 Activités liées à l'IDM

Les activités liées à la manipulation des modèles dans le cadre de l'IDM sont nombreuses et apportent chacune un ensemble de problématiques spécifiques. Nous présentons ici une liste non exhaustive de ces activités et donnons un aperçu des problématiques qui leur sont liées.

1.3.6.1 Réalisation de modèles

La réalisation des modèles nécessite non seulement l'expertise technique pour comprendre ou concevoir la partie du système concerné mais aussi une bonne connaissance du langage de modélisation utilisé. Dans le cas de systèmes complexes, les modèles deviennent également complexes et surtout de taille importante. La qualité de l'outillage devient alors essentielle car ce sont les outils qui permettent de mieux visualiser le modèle, de s'affranchir de certains détails ou encore de vérifier automatiquement la syntaxe. Les problématiques liées à l'outillage sont variées: elles concernent la visualisation des modèles, les méthodes d'assistance, le support des langages de modélisation, etc … .

1.3.6.2 Stockage de modèles

L'informatisation des modèles pose le problème de la gestion de leur persistance puis de leur accès par les utilisateurs. Les problématiques liées à cette activité concernent, par exemple, les formats de stockage, l'organisation du stockage ainsi que la gestion des méta-données concernant les modèles.

1.3.6.3 Echange de modèles

L'échange de modèles entre différents acteurs d'un projet est une vraie nécessité car elle conditionne la bonne communication entre ces acteurs. Des problèmes de compréhension de modèles entre différents acteurs peuvent avoir des conséquences catastrophiques sur le système implémenté. L'échange de modèles pose notamment des problèmes de format (sérialisation, transport, etc.), de traduction et d'interprétation de la sémantique pour l'intéropérabilité (notamment entre les outils de modélisation).

1.3.6.4 Exécution de modèles

L'exécution de modèles comprend un éventail de tâches différentes allant de la simulation à l'exécution en temps réel en passant par l'exécution symbolique ou la génération de code. Dans ce cadre, les problèmes majeurs concernent l'exécutabilité de la sémantique des langages de modélisation utilisés. En effet, cette propriété d'exécutabilité conditionne la possibilité de pouvoir calculer, à partir du modèle, un comportement du système, ou même tous les comportements possibles de ce système.

1.3.6.5 Vérification de modèles

La vérification d'un modèle consiste à vérifier les propriétés propres de ce modèle par rapport à ce que l'on attend de lui (correction syntaxique, sémantique, etc.). La vérification de modèle recouvre différents aspects allant de la vérification de la syntaxe à la vérification de la sémantique. Les problématiques les plus complexes concernent bien sûr la vérification de la sémantique. Différentes techniques de vérification existent, avec leurs problématiques particulières: la preuve, le test ou encore le model-checking. Les techniques de preuve s'appuient sur l'utilisation de représentations formelles (à base de logique, d'automates, de Réseaux de Petri par exemple) du système. Dans ce contexte, on cherche à prouver des propriétés comme la consistance ou la complétude d'un modèle. Dans le cas de systèmes complexes, ces tâches deviennent impossibles à réaliser sur un modèle préexistant et un axe important de recherche vise à obtenir ces propriétés par construction de plusieurs modèles. Les techniques de model-checking visent à analyser le comportement spécifié par le modèle de manière à vérifier des propriétés comme la sûreté, l'atteignabilité ou la vivacité. Les problématiques dans ce contexte sont liées notamment à l'identification avec exhaustivité des états possibles du système (explosion des espaces d'état). Enfin, le test est utilisé en complément du model-checking, notamment dans le cas de systèmes pour lesquels le model-checking est particulièrement inefficace (lorsque les systèmes sont trop complexes par exemple). L'évaluation de la pertinence des tests est une des difficultés

principales dans ce domaine, et elle conditionne la façon dont les tests sont sélectionnés. Les enjeux liés à la vérification des modèles comptent aujourd'hui parmi les enjeux les plus importants de l'IDM.

1.3.6.6 Validation

La validation permet de vérifier que le système implémenté répond aux besoins initiaux qui ont amené à sa conception. Certaines techniques comme le test peuvent être utilisées à la fois pour la vérification et pour la validation. Dans ce cadre, les modèles permettent notamment de générer des scénarios et des vecteurs de test de façon automatique [Bernot91]. Par ailleurs, afin de minimiser les risques d'erreur de conception le plus en amont possible, les modèles peuvent être validés les uns par rapport aux autres au cours du cycle de développement. Il s'agit alors, par exemple, de vérifier que certaines propriétés sont préservées d'un modèle à un autre. Un ensemble de problématiques de ce domaine est lié au mécanisme de raffinement du modèle, par lequel on obtient un modèle plus détaillé à partir d'un autre nécessitant souvent un apport d'information.

1.3.6.7 Gestion de l'évolution des modèles

Les modèles évoluent au cours du cycle de développement du système. Ils peuvent être modifiés dans le cadre de correction d'erreurs ou d'ajout de fonctionnalités par exemple. Les problématiques dans ce contexte concernent en particulier la répercussion automatique des modifications sur les différents modèles impliqués.

1.4 les approches de l'Ingénierie dirigée par les modèles

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

1.4.1 L'Architecture Dirigée par les Modèles (MDA)

L'MDA est une approche de développement proposée et soutenue par l'OMG (*Object Management Group*) [OMG04]. L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des spécifications techniques de son implémentation sur une plate-forme donnée. Autrement dit, cette approche permet de réaliser le même modèle sur plusieurs

plates-formes grâce à des projections. La mise en œuvre du MDA est entièrement basée les modèles et leurs transformations.

Le cycle de développement de l'approche MDA est vu sous la forme d'un Y [Roques00] (voir la Figure 1.3) dont les branches représentent respectivement les spécifications fonctionnelles du système et les spécifications techniques de la plate-forme cible qui, une fois intégrées, mènent à l'implémentation. Par conséquent, le fossé entre le modèle et le système n'existe plus car le modèle est le système, ou au moins le système est sensé être généré directement et automatiquement à partir du modèle.

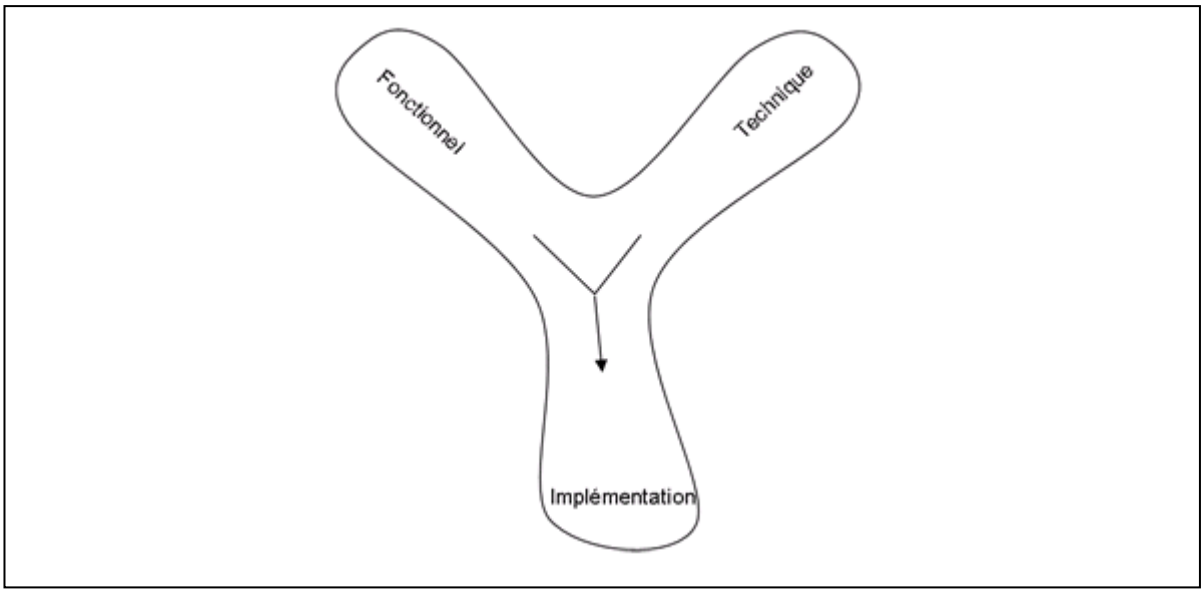


Figure 1.3 : Le cycle de développement en Y

1.4.1.1 Standards de l'OMG

L'approche MDA a le mérite d'être élevée au rang de standards de l'OMG (voir la Figure 1.4), dont notamment UML, OCL, MOF, XMI et CWM.

UML (*Unified Modeling Language*) [OMGa] Un langage visuel permettant de modéliser des systèmes à l'aide de diagrammes et de textes. Il permet aussi de décrire des architectures, des solutions ou des points de vue.

OCL (*Object Constraint Language*) [OMGd] Un ajout à UML, lui apportant la capacité de formaliser l'expression des contraintes. Il est désormais intégré à UML.

MOF *(Meta-Object Facility)* [OMGb] Un ensemble d'interfaces standards permettant de définir et de modifier des méta-modèles et leurs modèles correspondants. Le MOF est un standard de méta-modélisation pour définir la syntaxe et la sémantique d'un langage de modélisation, il a été créé par l'OMG afin de définir la notation UML.

XMI *(XML Metadata Interchange)* [OMGe] Un standard d'échange de métadonnées.

CWM *(Common Warehouse Metamodel)* [OMGc] Une interface servant à faciliter les échanges de métadonnées entre outils, plates-formes et bibliothèques de métadonnées dans un environnement hétérogène. Il est basé sur UML, MOF et XMI.

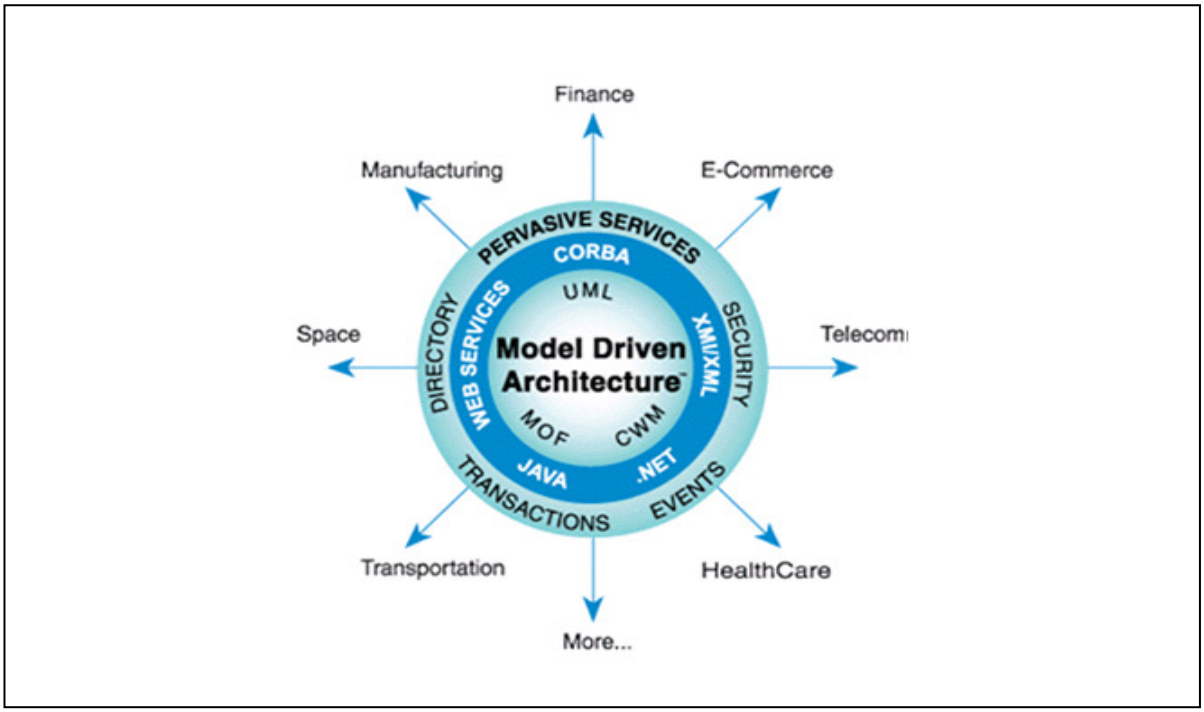


Figure 1.4 : Les standards de l'Architecture Dirigée par les Modèles

1.4.1.2 Transformations de modèles dans MDA

Dans l'approche MDA, l'OMG a défini plusieurs modèles qui vont servir dans un premier temps à modéliser l'application puis, par transformations successives, à générer le code de l'application. Les quatre principaux types de modèles définis dans l'approche MDA sont les suivants [Bézivin02]:

CIM *(Computation Independant Model)*: Appelé aussi modèle de domaine ou modèle métier, le CIM capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. Dans la pratique, l'appellation "CIM" est très peu utilisée.

PIM *(Platform Independant Model)*: Le PIM décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.) mais il doit rester technologiquement neutre.

PDM *(Platform Description Model)*: Le PDM est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit.

PSM *(Platform Specific Model)*: Le PSM est le résultat de la combinaison du PIM et du PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation.

La Figure 1.5 donne une vue générale des transformations possibles entre ces différents types de modèles.

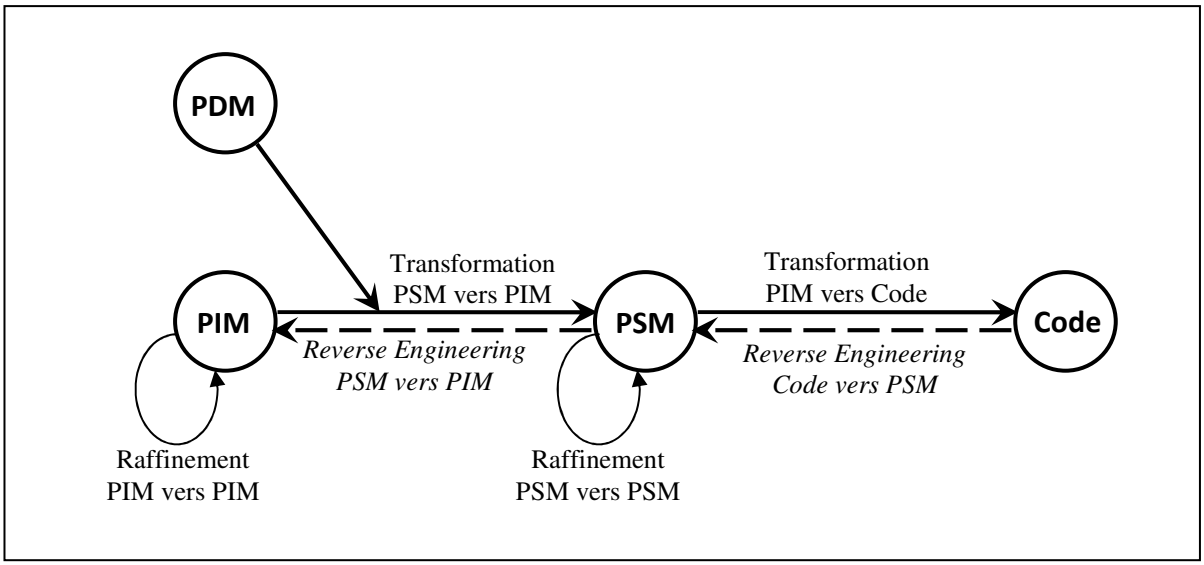


Figure 1.5 : Les modèles et les transformations dans l'approche MDA

Transformations PIM ⇨ PIM et PSM ⇨PSM: Les transformations de type PIM vers PIM ou PSM vers PSM visent à enrichir, filtrer ou spécialiser le modèle. Il s'agit de transformations de modèle à modèle [Czarnecki06].

Transformation PIM ⇨ PSM: La transformation de PIM vers PSM permet de spécialiser le PIM en fonction de la plate-forme cible choisie. Elle n'est effectuée qu'une fois le PIM suffisamment raffiné. Cette transformation de modèle à modèle est réalisée en s'appuyant sur les informations fournies par le PDM.

Transformation PSM ⇨ Code: La transformation de PSM vers l'implémentation (le code) est une transformation de type modèle à texte [Czarnecki06]. Le code est parfois assimilé à un PSM exécutable. Dans la pratique, il n'est généralement pas possible d'obtenir la totalité du code à partir du modèle et il est alors nécessaire de le compléter manuellement.

Transformations inverses PSM ⇨ PIM et code ⇨ PSM: Ces transformations sont des opérations de rétro-ingénierie (reverse engineering). Ce type de transformations pose de nombreuses difficultés mais il est essentiel pour la réutilisation de l'existant dans le cadre de l'approche MDA.

1.4.1.3 MOF et L'architecture à quatre niveaux

L'OMG, dans le cadre de ses travaux concernant la méta-modélisation, a défini la notion de méta-méta-modèle ainsi que la standardisation d'une architecture générale décrivant les liens entre modèles, méta-modèles et méta-méta-modèles [OMG04]. Cette architecture est hiérarchisée en quatre niveaux comme le montre la Figure 1.6.

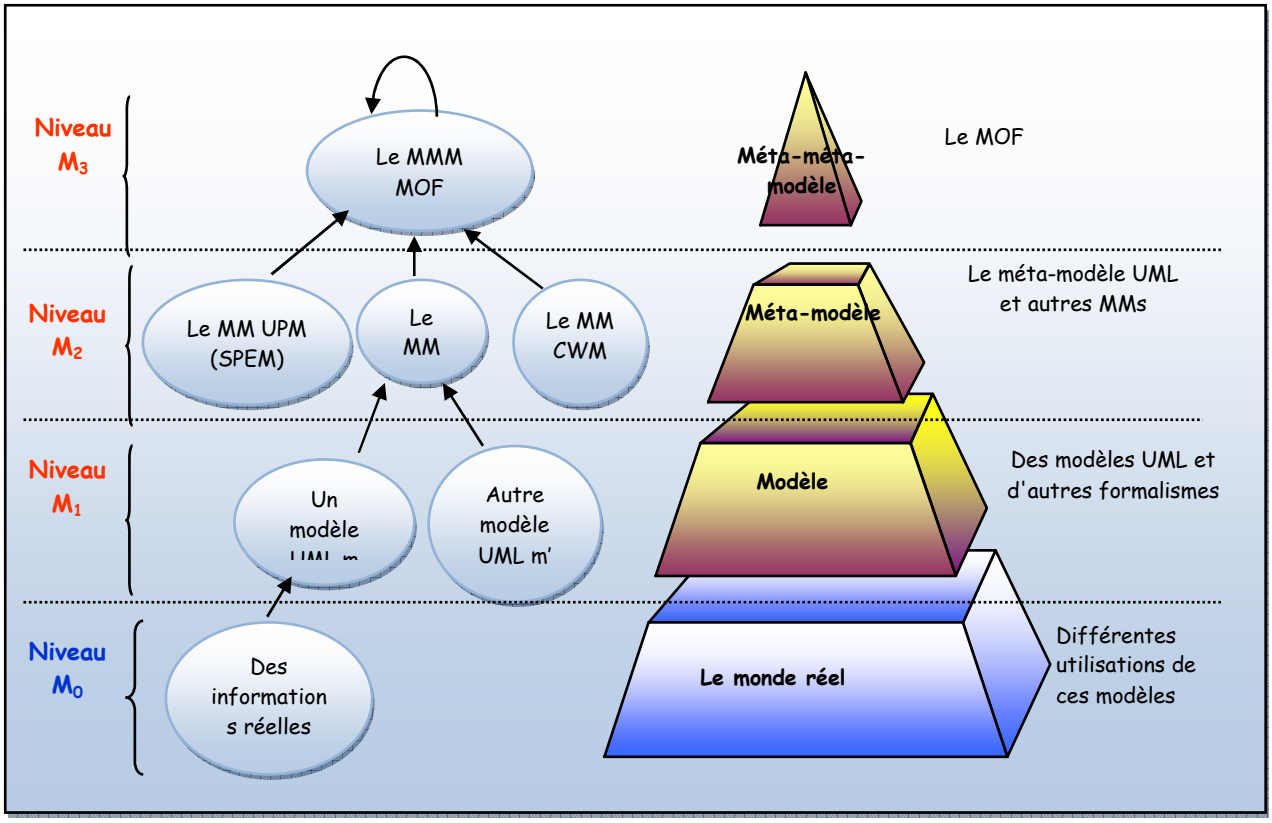


Figure 1.6: Architecture à quatre niveaux

Niveau M_3 (MMM): Le niveau M_3 (ou méta-méta-modèle) est composé d'une seule entité qui s'appelle le MOF (*Meta Object Facility [OMGb]*). Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif,

Niveau M_2 (MM): Le niveau M_2 (ou méta-modèle) définit le langage de modélisation et la grammaire de représentation des modèles M_1. Le méta-modèle UML, qui définit la structure interne des modèles UML suivant le standard UML, fait partie de ce niveau. Les profils UML, qui étendent le méta-modèle UML, appartiennent aussi à ce niveau. Les concepts définis par un méta-modèle sont des instances des concepts du MOF.

***Niveau M_1 (M)*:** Le niveau M_1 (ou modèle) est composé de modèles d'information. Il décrit les informations de M_0. Les modèles UML, les PIM et les PSM appartiennent à ce niveau. Les éléments d'un modèle (M_1) sont des instances des concepts décrits dans un méta-modèle (M_2).

***Niveau M_0*:** Le niveau M_0 (ou instance) correspond au monde réel. Il ne s'agit pas à proprement parler d'un niveau de modélisation. Ce niveau contient les informations réelles de l'utilisateur.

Il est important de noter que la proposition initiale d'OMG dans l'approche MDA était d'utiliser le langage UML comme unique langage de modélisation. Cependant, les limites d'UML ont rapidement été atteintes et il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profiles, afin d'exprimer de nouveaux concepts relatifs à des domaines spécifiques. Ces extensions devenant de plus en plus importantes et la communauté MDA a élargi son point de vue en considérant des langages de modélisation spécifiques à un domaine.

1.4.2 Modèle de Calcul Intégré (MIC)

Les travaux autour du Modèle de Calcul Intégré (Model Integrated Computing (MIC)) [Sztipanovits05], au milieu des années 90, ont proposé une approche de développement logiciel basée sur des modèles spécifiques au domaine d'application spécifique. La motivation de cette approche était d'apporter une méthodologie de développement ainsi que des outils logiciels comme support.

La méthodologie proposée est décomposée en deux phases. La première phase consiste à analyser le domaine d'application afin de trouver les paradigmes de modélisation les plus appropriés et de définir avec précision le langage de modélisation qui sera utilisé. Un outil automatique peut alors utiliser ces informations pour générer automatiquement un environnement de modélisation dédié au domaine. Cet environnement est utilisé directement dans la seconde phase qui consiste à modéliser l'application désirée. La plate-forme GME (*Generic Modeling Environment*) [Davis03] est une implémentation de la méthodologie définie par l'approche MIC.

1.4.3 Les usines logicielles (Software Factories)

Les usines logicielles "Software Factories" [Greenfield04] représentent la vision que Microsoft a proposée pour l'ingénierie dirigée par les modèles. Le terme d'usine logicielle fait référence à une industrialisation du développement logiciel similaire aux lignes d'assemblage dans l'industrie qui se base sur les idées suivantes:

- Les lignes d'assemblage ne fabriquent généralement qu'un seul type de produit avec de faibles points de variation. Ainsi les lignes de production dans l'automobile ne produisent

qu'un seul type de voiture, avec des variations possibles pour la couleur ou les combinaisons d'options.

- Les ouvriers sont souvent spécialisés. Si certains ont parfois des activités relativement variées, elles ne couvrent jamais la totalité des activités de la ligne d'assemblage.
- Les outils utilisés de la ligne d'assemblage sont très spécialisés et très automatisés. Ils ne peuvent généralement pas être réutilisés sur d'autres lignes de production que celle pour laquelle ils ont été conçus, ce qui permet d'atteindre des degrés d'automatisation importants.
- Les composants à assembler ou à usiner proviennent souvent de tierces parties (fournisseurs). Par exemple, les lignes de production automobile assemblent généralement des pièces qui sont produites dans d'autres usines.

L'approche de Microsoft propose d'adapter ces idées au développement de systèmes logiciels. Selon les deux premiers points, les produits logiciels ainsi que les développeurs au sein des équipes de développements devraient être hautement spécialisés. Le troisième point correspond également à l'utilisation d'outils de développement spécialisés au domaine d'application. Ce principe inclut les langages de modélisation, les logiciels d'assistance ainsi que les transformations. Le dernier point pousse à la réutilisation de composants déjà développés ailleurs.

En résumé, une "Software Factory" est un environnement de développement configuré pour produire rapidement un type spécifique d'applications. L'environnement de développement intégré "*Microsoft Visual Studio.NET" 2008* met en application ces idées et propose une plateforme de développement qui peut être configurée pour cibler un domaine particulier.

1.4.4 Synthèse

Les approches à base de modèles précédentes dans les sections précédentes ont permis l'émergence de l'IDM dont l'objectif est de mettre l'accent sur un certain nombre d'idées clef de chaque approche. Le MDA a mis en évidence l'intérêt de séparer les spécifications fonctionnelles d'une application de son implémentation sur une plate-forme donnée. Le MIC a fait émerger la notion de générateur d'éditeurs de modèles pour un domaine spécifique. Les usines logicielles ont mis en évidence la notion de chaîne de transformation d'artéfacts.

1.5 Processus de Vérification en IDM

Dans le contexte de l'ingénierie dirigée par les modèles, le développement des systèmes complexes fait appel à différentes techniques de modélisation qui dépendent:

- ✓ du domaine du système ou du sous système.
- ✓ Des différentes phases du cycle de développement.
- ✓ Des différents niveaux d'abstraction aux quels le système est étudié.
- ✓ Des différents aspects spécifiques à analyser et à vérifier lors de la conception.

Dans le but d'analyser le comportement modélisé pour assurer le bon fonctionnement du système et détecter d'éventuels problèmes, différentes techniques telles que la simulation, le test ou les méthodes formelles sont alors utilisées dès les premières étapes de sa conception. Dans ces techniques, les modèles peuvent être utilisés comme support pour ces activités de vérification et de validation. Certaines analyses peuvent être conduites directement en utilisant les modèles. Par exemple, des simulations ou des tests sur des modèles comportementaux sont souvent favorisés. En effet, il s'agit d'une méthode simple et relativement peu coûteuse, consistant à générer des cas de tests pertinents pour vérifier les scénarios attendus. En revanche, même pour un système d'une taille raisonnable, l'analyse des comportements possibles ne peut pas être exhaustive et l'on court le risque de ne pas identifier des situations potentiellement dangereuses pour le système.

D'autres analyses nécessitent la transformation des modèles réalisés dans des formalismes formels adaptés à un type de vérification formelle désirée. Les méthodes formelles, telles que la preuve de théorème, les réseaux de Petri ou le Model-Checking, s'appuient sur un cadre mathématique précis et non ambigu permettant de modéliser à la fois le système et les propriétés qu'il doit vérifier. Par exemple, la déduction d'un réseau de Petri global modélisant le comportement du système à partir des modèles comportementales permet de mettre en œuvre les outils de preuve formelle associés aux réseaux de Petri.

1.6 Conclusion

Dans ce chapitre nous avons défini le contexte dans lequel se place cette thèse et détaillé les principaux concepts du domaine de l'ingénierie des modèles. Nous avons mis en évidence le rôle central des modèles et de l'outillage sous-jacent dans le processus de conception des systèmes. Nous avons aussi identifié et mis en valeur les problématiques actuelles liées à la manipulation des modèles et au besoin de s'assurer de leur qualité tout en réduisant les coûts et le délai de production. Dans le cadre de l'approche IDM, nous avons présenté différentes techniques de traitement des modèles parmi lesquelles nous avons détaillé la méta-modélisation ainsi que les techniques de transformation de modèles. Ces techniques sont aujourd'hui des pivots majeurs pour l'innovation dans les domaines de recherche liés à la modélisation des systèmes complexes.

Chapitre 02 :

Intégration des Méthodes Formelles à l'IDM

2.1 Introduction

Face à la complexité croissante des systèmes complexes et les contraintes de sûreté et de bon fonctionnement, le développement de ces systèmes s'appuie de plus en plus sur les activités de modélisation, de vérification et de validation. La modélisation permet de séparer les différentes préoccupations qui interviennent dans le cycle de développement en procurant aux développeurs des langages de modélisation pour exprimer exactement les informations nécessaires à chacune d'entre elles. Pour s'assurer de la fiabilité et de la sûreté de fonctionnement du système, en particulier dans les premières étapes de sa conception, il est nécessaire d'associer à la modélisation des techniques d'analyses formelles pour vérifier les modèles par rapport aux propriétés attendues et valider les modèles par rapport aux exigences du client.

Une des difficultés à intégrer les méthodes formelles dans les activités de développement des systèmes est liée à la difficulté réelle de manipuler les concepts théoriques et les méthodes d'analyse associées, ainsi que la capacité des développeurs d'exprimer les propriétés du système de façon aisée.

L'IDM joue un rôle essentiel dans l'introduction des méthodes formelles dans les activités de développement des systèmes. Celle-ci repose, d'une part, sur la définition de langages par le biais de la méta-modélisation qui permet d'exprimer les différents aspects d'un langage. D'autre part, sur l'utilisation de transformations de modèles pour combiner ces aspects, échanger des informations entre eux afin de générer des modèles dans un langage formel pour permettre la validation et la vérification par différentes techniques formelles.

Dans ce chapitre, nous présentons un panorama sur les méthodes formelles et leur classification. Ensuite nous abordons l'utilisation et l'intégration de ces méthodes formelles à la méthodologie IDM afin d'augmenter la fiabilité et de garantir l'absence d'erreurs de conception. Dans ce contexte, les réseaux de Petri représentent une technique formelle largement utilisée. Un intérêt particulier sera porté sur eux.

2.2 Méthodes Formelles

Les méthodes formelles (*MFs*) sont des techniques basées sur les mathématiques pour décrire des propriétés de systèmes. Elles fournissent un cadre très rigoureux pour spécifier, développer et vérifier des systèmes d'une façon systématique, plutôt que d'une façon ad hoc, afin de démontrer leur validité par rapport à une certaine spécification [Wing90].

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

2.2.1 Les langages formels

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

Un langage formel est en effet un langage doté d'une sémantique mathématique adéquate basée sur des règles d'interprétation et des règles de déduction [Petit99, Benzaken91]. Les règles d'interprétation garantissent l'absence d'ambiguïté dans les descriptions produites, contrairement à des descriptions en langage informel ou semi-formel qui peuvent donner lieu à différentes interprétations. Alors que les règles de déduction permettent de raisonner sur les spécifications afin de découvrir de potentielles incomplétudes, inconsistances ou pour prouver des propriétés attendues.

2.2.2 Techniques d'analyse

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

2.2.2.1 Vérification

La vérification répond à la question "Construisons-nous correctement le modèle ?" ("*is the system being built right?*").

La vérification est l'ensemble des actions de revue, inspection, test, simulation, preuve automatique, ou autres techniques appropriées permettant d'établir et de documenter la conformité des artéfacts du développement vis-à-vis des critères préétablis. La vérification est définie comme étant "*la confirmation par examen et apport de preuves tangibles (informations dont la véracité peut être démontrée, fondée sur des faits obtenus par observation, mesures, essais ou autres moyens) que les exigences spécifiées ont été satisfaites*" [ISO 8402].

2.2.2.2 Validation

La validation cherche à répondre à la question "Construisons nous le bon modèle?" ("*is the right system being built?*"). La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs. Par définition la validation est la "*confirmation, par examen et apport de preuves tangibles, que les exigences particulières pour un usage spécifique prévu sont satisfaites. Plusieurs validations peuvent être effectuées s'il y a différents usages prévus*" [ISO 8402].

2.2.2.3 Qualification

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

2.2.2.4 Certification

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

2.2.3 Classification des Méthodes Formelles

Dans la littérature, il existe plusieurs classifications des méthodes formelles. Selon J.M. Wing [Wing90], on peut distinguer les méthodes:

- Orientées opérations pour décrire le fonctionnement du système et son comportement par des axiomes.
- Orientées données pour décrire les états du système
- Hybrides en combinant les deux orientations.

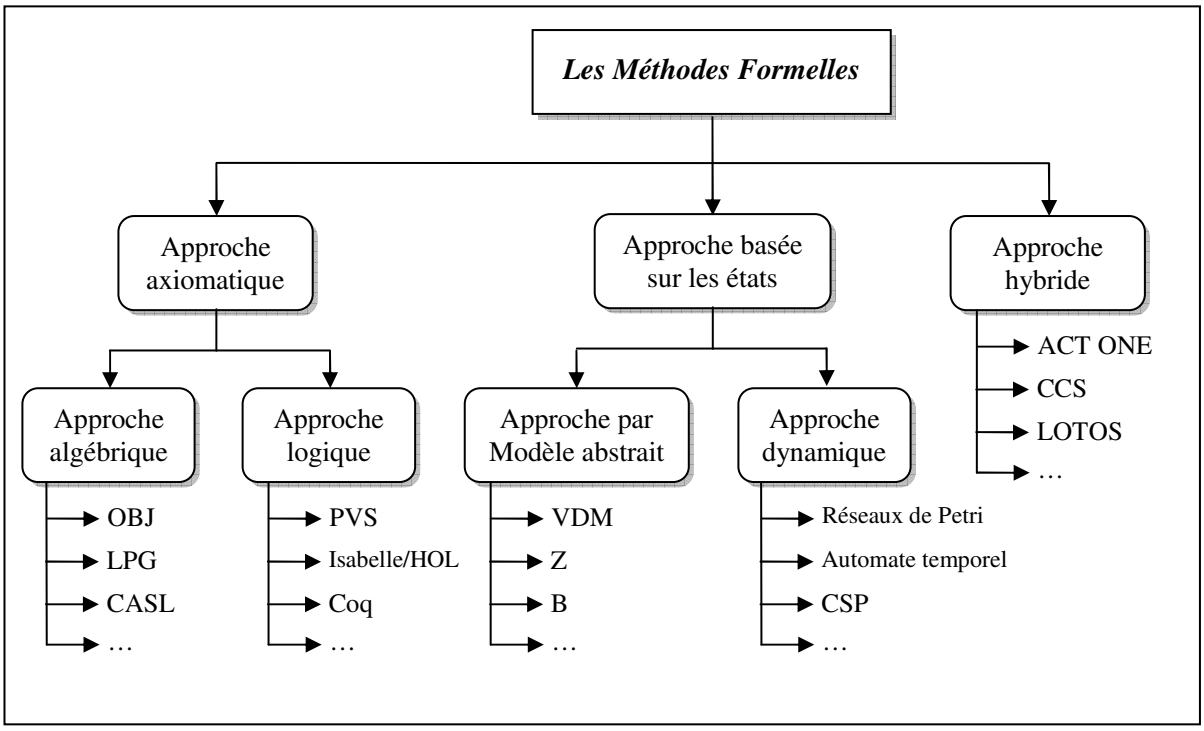


Figure 2.1 : La classification des méthodes formelles

2.2.3.1 L'approche axiomatique

Dans cette approche, on s'intéresse au comportement du système de manière indirecte en construisant une axiomatisation qui fournit les propriétés comportementales du système traité en utilisant une approche algébrique ou une approche logique [Attiogbé07].

L'approche algébrique permet de définir des types abstraits de données en spécifiant pour chaque opération les types de valeurs de ses paramètres et du résultat, en précisant une expression construite à l'aide des variables appelées termes, et en décrivant les propriétés des opérations sous forme d'équivalences entre termes (les axiomes). L'application des axiomes à des termes permet d'obtenir d'autres expressions d'équivalence [Truong06]. Parmi les langages de spécification algébrique connus nous pouvons citer : OBJ [Goguen96], LPG [Bert95] et Larch [Guttag85].

L'approche logique, quant à elle, permet d'exprimer des systèmes transformationnels en utilisant la logique temporelle pour exprimer des propriétés dynamiques de sûreté et vivacité des systèmes réactifs. Dans cette approche on s'intéresse à la démonstration de programmes en appliquant les théories de la démonstration automatique ou semi-automatique de théorèmes [Truong06]. Parmi les langages de spécification logiques connus nous pouvons citer : PVS [Owre93], Isabelle/HOL [Smith02] et Coq [Behrmann04].

2.2.3.2 L'approche basée sur les états

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

Les approches ensemblistes, appelées aussi approches par modèle abstrait, permettent de fournir une syntaxe et une sémantique du modèle abstrait en se basant sur la théorie des ensembles, logique du premier ordre, ou la théorie des types. Les approches ensemblistes différent des approches algébriques par l'utilisation des types abstraits prédéfinis pour modéliser l'état du système à construire. Chaque opération est spécifiée indépendamment en décrivant son effet sur l'état du système. Parmi les langages de spécification ensemblistes connus nous pouvons citer: VDM [Cliff86], Z [Attiogbé07] et B [Truong06].

Les approches dynamiques se basent sur la notion de processus pour spécifier les systèmes de transitions en utilisant les automates, les réseaux de Petri [Murata89] et les algèbres de processus comme CSP [Attiogbé07].

2.2.3.3 L'approche hybride

L'approche hybride complète une axiomatisation par un modèle de données ou bien l'inverse [Attiogbé07]. Par exemple, LOTOS [Turner07] est considéré comme un langage algébrique car ses expressions de contrôle sont caractérisées par une algèbre dont les termes sont des processus. Alors que les données sont caractérisées par une algèbre de types abstraits dont les termes sont des expressions fonctionnelles [Truong06]. Le langage CCS [Milner80] et le langage ACT ONE [Charles97] sont les prédécesseurs de LOTOS.

Généralement, l'approche basée sur les états est l'approche la plus utilisée pour vérifier les aspects dynamiques d'un système car cette approche permet un raisonnement sur le fonctionnement d'un système. Spécialement, les réseaux de Petri constituent le langage le plus utilisé parce qu'ils combinent les avantages de la représentation graphique avec la sémantique formelle attribuée au comportement des systèmes. Pour cette raison, nous allons détailler les réseaux de Petri dans la section 4.

2.2.4 Techniques de vérification formelle

De façon générale, on distingue deux grandes catégories de techniques pour vérifier formellement la correction d'un système. La première catégorie de techniques est appelée

vérification de modèles, ou *model checking*, qui consiste à construire un modèle à partir d'une description formelle d'un système. La seconde catégorie regroupe les techniques de preuve (*theorem proving)* qui sont des démonstrations mathématiques au sens classique du terme où la vérification des propriétés est effectuée par déduction à partir d'un ensemble d'axiomes et de règles d'inférences.

2.2.4.1 Vérification de Modèle (Model-Checking)

La vérification de modèles est une technique qui consiste à construire un modèle fini d'un système et vérifier qu'une propriété cherchée est vraie dans ce modèle. Il y a deux façons générales de vérification dans le Model-checking: vérifier qu'une propriété exprimée dans une logique temporelle est vraie dans le système, ou comparer (en utilisant une relation d'équivalence ou de préordre) le système avec une spécification pour vérifier si le système correspond à la spécification ou non. Au contraire du Theorem proving, le Model-checking est complètement automatique et rapide. Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage.

2.2.4.2 Preuve de théorèmes (Theorem proving)

La preuve de théorèmes est une technique où le système et les propriétés recherchées sont exprimés comme des formules dans une logique mathématique. Cette logique est décrite par un système formel qui défini un ensemble d'axiomes et de règles de déduction. La preuve de théorèmes est le processus de recherche de la preuve d'une propriété à partir des axiomes du système. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été éventuellement dérivés. Au contraire du Model-checking, le Theorem proving peut s'utiliser avec des espaces d'états infinis à l'aide de techniques comme l'induction structurelle. Son principal inconvénient est que le processus de vérification est normalement lent, sujet à l'erreur, demande beaucoup de travail et des utilisateurs très spécialisés avec beaucoup d'expertise.

2.3 Combinaison d'IDM avec les Méthodes Formelles

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

Bien sûr, chacune des deux approches a des points forts et des points faibles. Dans ce qui suit, nous examinons comment ces deux approches peuvent être combinées en montrant comment les inconvénients des méthodes formelles peuvent être surmontés grâce aux apports de l'IDM et réciproquement. La Figure 2.2 représente brièvement les avantages et les inconvénients de l'IDM et des MFs [Gargantini09].

	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 2.2: Les MFs & IDM

Avantages des MFs. L'utilisation des méthodes formelles dans l'ingénierie des systèmes devient indispensable, surtout dans les phases amont du développement. Un modèle abstrait du système peut servir de support pour s'assurer que le système en cours de développement répond aux exigences des clients par simulation ou tests, et de garantir certaines propriétés liées à son comportement par l'analyse formelle (validation et vérification).

Inconvénients des MFs. Bien qu'il existe plusieurs applications des méthodes formelles dans le monde industriel et avec de bons résultats, les développeurs hésitent encore à adopter les MFs. En plus de leur difficulté d'utilisation et d'apprentissage, ce scepticisme est principalement dû:

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

Avantages de l'IDM. L'approche IDM met davantage l'accent sur l'automatisation du processus de développement des systèmes. Cette approche est basée principalement sur des représentations du système à un haut niveau d'abstraction qui sont les modèles. Le processus

de développement, dans cette approche, revient à raffiner, maintenir, et éventuellement de transformer en d'autres modèles ou de générer le code exécutable. Il est important à noter que ces différentes activités se font d'une manière automatique. En plus, la méta-modélisation, qui est aussi un concept clé de l'approche IDM, permet de donner à un langage de modélisation une notation abstraite, ce qui permet de générer automatiquement son éditeur. La méta-modélisation de langages est de plus en plus adoptée pour des domaines spécifiques afin de réduire la complexité et d'exprimer efficacement les concepts du domaine.

Inconvénients de l'IDM. Malgré que la définition d'une syntaxe abstraite d'un langage par un méta-modèle est bien maîtrisée et supportée par de nombreux environnements de méta-modélisation, la définition de la sémantique de ces langages reste une question ouverte et cruciale. Actuellement, les environnements de méta-modélisation sont en mesure de faire face à la plupart des problèmes de définition syntaxique, mais ils manquent de tout support rigoureux permettant de fournir la sémantique des méta-modèles. La sémantique est généralement donnée en langage naturel, cela implique que les langages définis par méta-modélisation ne sont pas encore aptes à l'analyse formelle de leurs modèles.

L'absence de notations conviviale, d'intégration des déférentes techniques formelles et d'interopérables de leurs outils sont des défis importants pour les MFs. L'IDM permet, par le biais des notions de méta-modèle et de transformation de modèles, de concevoir des supports outillés pour les méthodes formelles tout en assurant leur interopérabilité. En se sens, l'IDM n'apporte rien de nouveau sur le plan théorique aux concepts d'analyse formelle, mais elle peut leur permettre une meilleure intégration afin de profiter pleinement de leurs avantages.

2.4 Réseaux de Petri

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

Le formalisme des réseaux de Petri (*RdP*) a été introduit par Carl Adam Petri en 1962 à l'université Darmstadt dans sa thèse intitulée : " *Communication avec des automates*" comme un outil de modélisation graphique et mathématique permettant la modélisation et l'analyse des systèmes dynamiques à événements discrets. En tant qu'outil graphique, les réseaux de Petri permettent la visualisation du comportement dynamique et les activités concurrentes du système.

En tant qu'outil mathématique, ils permettent l'analyse des propriétés du système concernant sa structure et son comportement. Ce formalisme bénéficie d'une riche panoplie de techniques d'analyse et d'outils. Les résultats de cette analyse sont utilisés pour évaluer le système et en permettre la modification ou l'amélioration. La Figure 2.3 montre la méthode générale basée sur le formalisme des réseaux de Petri pour la modélisation et l'analyse des systèmes.

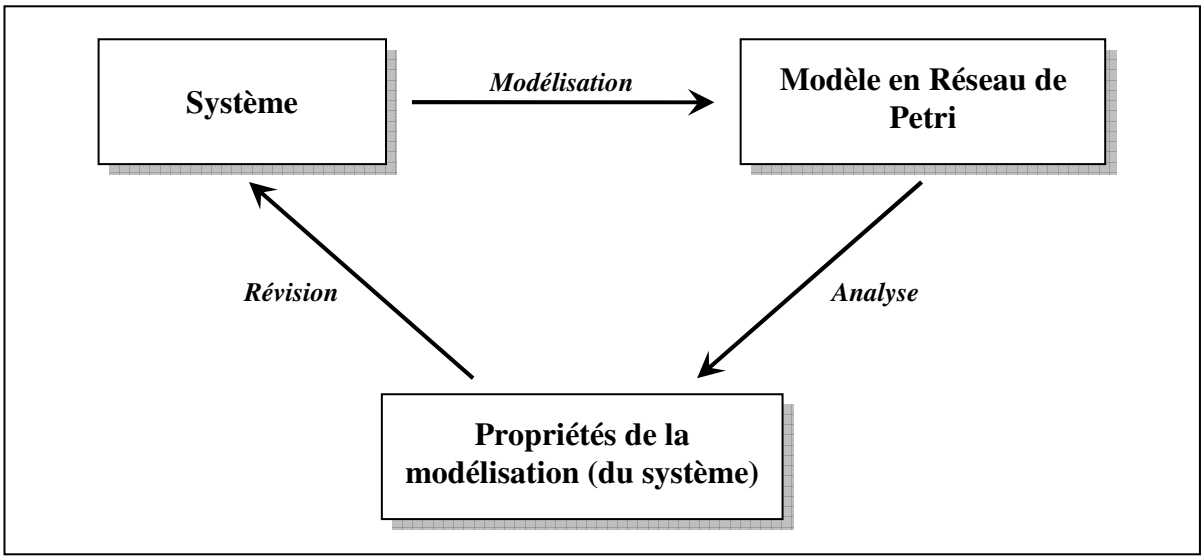


Figure 2.3 : Méthodes générale de modélisation et d'analyse basée sur les réseaux de Petri

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

2.4.1 Concepts de base & définition

2.4.1.1 Définitions informelles

Comme l'illustre le RdP de la Figure 2.4, un réseau de Petri est un graphe biparti orienté (ayant deux types de nœuds): des *places* représentées par des cercles et des *transitions* représentées par des rectangles. Les arcs du graphe ne peuvent relier que des places vers des transitions, ou des transitions vers des places.

Un réseau de Petri décrit un système dynamique à événements discrets. Les places permettent la description des états possibles du système et les transitions permettent la description des événements ou les actions qui causent le changement de l'état. Un réseau de Petri est un graphe muni d'une sémantique opérationnelle, c'est-à-dire qu'un comportement est associé au graphe, ce qui permet de décrire la dynamique du système représenté. Pour cela un troisième élément est ajouté aux places et aux transitions: les *jetons*.

Une répartition des jetons dans les places à un instant donné est appelée *marquage* du réseau de Petri. Un marquage donne l'état du système. Le nombre de jetons contenus dans une place est un entier positif ou nul. Pour un marquage donné, une transition est franchissable si chacune de ses places d'entrée contient au moins un jeton. L'ensemble des transitions franchissables pour un marquage donné défini l'ensemble des changements d'états possibles du système depuis l'état correspondant à ce marquage. C'est un moyen de définir l'ensemble des événements auxquels ce système est réceptif dans cet état.

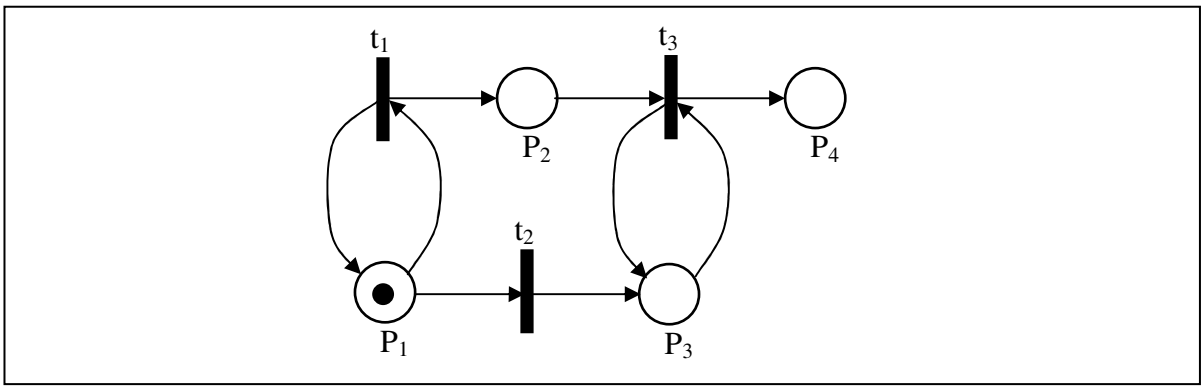


Figure 2.4: Exemple de réseau de Petri marqué

2.4.1.2 Définitions formelles

Un réseau de Petri (R) peut être représenté par un triplet $R = (P, T, W)$

Où :
- P est l'ensemble des places (les places représentent les conditions).
- T l'ensemble des transitions (les transitions représentent les actions) tel que : $P \cap T = \emptyset$.
- W est la fonction définissant le poids porté par les arcs tel que $W: \big((P \times T) \cup (T \times P)\big) \rightarrow N = \{0, 1, 2, \dots\}$.

Le réseau R est fini si l'ensemble des places et des transitions est fini: $|P \cup T| \in N$.

Un réseau $R = (P, T, W)$ est ordinaire si:

$$\forall (x, y) \in \big((P \times T) \cup (T \times P)\big): W\ (x, y)\ \leq 1.$$

Dans un réseau ordinaire la fonction W est remplacée par F où:

$$F \subseteq \big((P \times T) \cup (T \times P)\big)\ tel\ que\ (x, y) \in F\ \Leftrightarrow W\ (x, y) \neq 0\ .$$

Pour chaque $x \in (P \cup T)$:

${}^{*}x$ représente l'ensemble des entrées de x: ${}^{*}x = \{y\ \in P \cup T\ / W(y, x) \neq 0\}$.

x^* représente l'ensemble des sorties de x: $x^* = \{y \in P \cup T / W(x,y) \neq 0\}$.

Si ${}^*x = \emptyset$, x est dite source. Si $x^* = \emptyset$, x est dite puits.

Le *comportement* d'un réseau de Petri est déterminé par sa structure statique et par son état. L'état d'un réseau de Petri se modélise à l'aide d'un marquage que l'on fait évoluer en franchissant des transitions, ce qui correspond à exécuter les actions qui lui sont associées. Les règles suivantes permettent de représenter l'évolution des systèmes à modéliser:

I. Une transition t est dite franchissable pour le marquage M si pour toute place p telle que $W(p,t) \neq 0$ est marquée d'au moins $W(p,t)$ jetons, i.e. si $\forall p \in P, M(p) \geq W(p,t)$. On note ceci $M(t)$.

II. Une transition t est franchie en retirant $W(p,t)$ jetons de chaque place p telle que $W(p,t) \neq 0$, et en ajoutant $W(t,p')$ jetons à chaque place p' telle que $W(t,p') \neq 0$, i.e. si t est franchissable pour M, alors le franchissement de t fait passer le marquage M au marquage M' de la façon suivante $\forall p \in P, M'(p) = M(p) - W(p,t) + W(t,p)$. On note ceci $M(t\rangle M'$, ce qui veut dire que lorsque la transition t est franchie à partir d'un marquage M, il faut saisir $W(p,t)$ jetons à partir de chaque place d'entrée à la transition t et déposer $W(t,p)$ jetons dans chaque place de sortie de la transition t ce qui permet de produire un nouveau marquage M'.

La notion de franchissement de transitions peut alors être étendue aux séquences de transitions. Soit $s = t_1\, t_2 \ldots t_n$ une séquence de transitions. La séquence s est franchissable à partir de M et conduit au marquage M' ce qui sera note $M(s\rangle M'$ ssi Il existe des marquages $M_0 = M, M_1, \ldots\ldots, M_n = M' \; tels\; que \forall i : 0 \leq i \leq n-1 : M_i(t_{i+1}\rangle M_{i+1}$.

2.4.2 Représentation Matricielle

La représentation matricielle d'un RdP est offerte afin de simplifier les tâches d'analyse et de vérification effectuées sur un modèle RdP. Agir sur une représentation graphique d'un modèle RdP est une tâche délicate en comparant avec une représentation matricielle.

Définition

Soit Un réseau de Petri $R = (P,T,W)$ avec $P = \{p_1, p_2, \ldots, p_m\}\; et\; T = \{t_1, t_2, \ldots t_n\}$. On appelle matrice des pré-conditions *pré,* la matrice $m \times n$ à coefficients dans N tel que $pré\,(i,j) = W(p_i, t_j)$, qui indique le nombre de marques que doit contenir la place p_i pour que la transition t_j devienne franchissable. De la même manière on définit la matrice des post-conditions *post*, la matrice $n \times m\; tel\; que\; post\,(i,j) = W(t_j, p_i)$ contient le nombre de marques déposées dans p_i lors du franchissement de la transition t_j. La matrice *C= post – pré* est appelée matrice

d'incidence du réseau (m représente le nombre de places d'un réseau de Petri et n le nombre de transitions.)

Le marquage d'un réseau de Petri est représenté par un vecteur de dimension m à coefficients dans N. La règle de franchissement d'un réseau de Petri est définie par :

$$M'(p) = M(p) + C(p,t)$$

La représentation matricielle du réseau de Petri de la Figure 2.4 est comme suit :

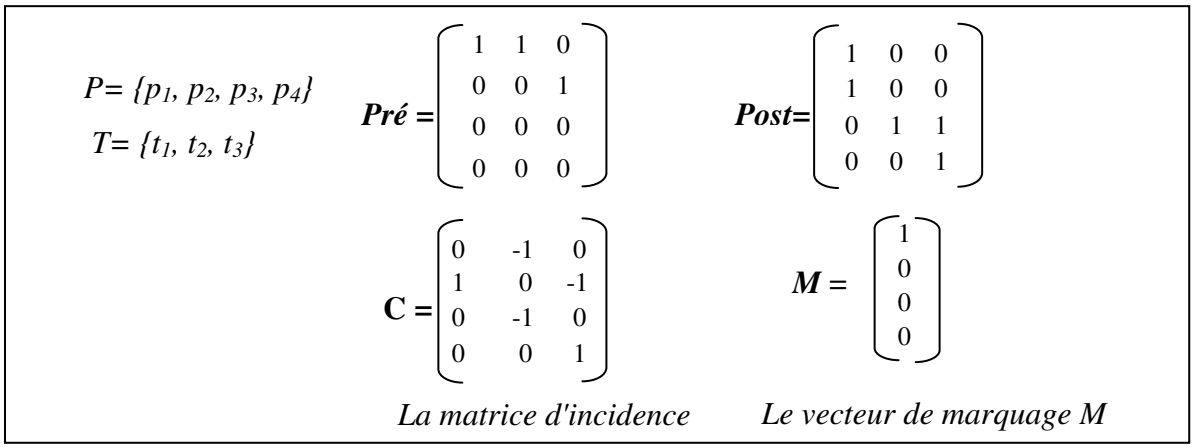


Figure 2.5: Matrice d'incidence et vecteur de marquage du RdP de la figure2.4

2.4.3 Propriétés des Réseaux de Petri

2.4.3.1 Les Propriétés Structurelles

Les propriétés structurelles dépendent uniquement de la topologie du réseau. Il s'agit de faire ressortir les propriétés statiques du système étudié. Ces différentes propriétés sont indépendantes du marquage. Ainsi, il est possible de faire apparaître, entre autres, les caractéristiques de synchronisation (*Figure 2.6* (a)) ou de précédence (*Figure 2.6* (b)).

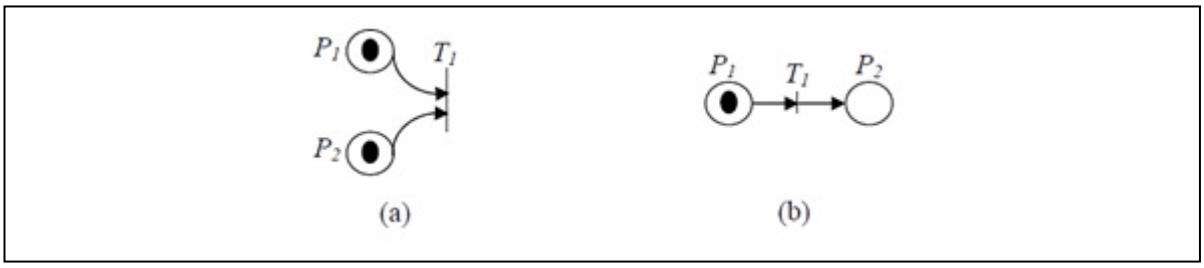


Figure 2.6 : Propriétés Structurelles des RdPs.

● ***Les conflits*** dans un RdP (*Figure 2.7*(a) et (b)): Si une place ce trouve en amont de plusieurs transitions. On note le conflit de la place P_i : $K = (P_i,\{T_1,T_2,\ldots\})$; Où T_1,T_2 ,... sont les transitions en concurrence. On parle de *conflit structurel* car cela ne dépend pas du marquage. Dans certains

cas, le franchissement de l'une des transitions peut empêcher le franchissement de l'autre (la *Figure 2.7* (a)). Le conflit devient *conflit effectif* quand il y a effectivement conflit, cela dépend du marquage (la *Figure 2.7* (b)).

● ***RdP à choix libre*** (*Figure 2.7* (c)) : Un RdP à choix libre est un réseau dans lequel pour tout conflit ($K = (P_i,\{T_1,T_2,\ldots, T_n\})$ aucune des transitions $T_1,T_2,\ldots,T_n$ ne possède aucune autre place d'entrée que P_i.

● ***RdP simple*** (*Figure 2.7* (d)) : Un RdP simple est un RdP dans lequel chaque transition ne peut être concernée que par un conflit au plus.

● ***RdP pur*** (*Figure 2.7* (e)): Un RdP pur est un réseau dans lequel il n'existe pas de transition ayant une place d'entrée qui soit à la fois place de sortie de cette transition.

● Un RdP est un ***graphe d'état*** (*Figure 2.7* (f)) si et seulement si toute transition a exactement une seule place d'entrée et une seule place de sortie.

● Un RdP est un ***graphe d'événement*** (*Figure 2.7* (g)) si et seulement si chaque place possède exactement une seule transition d'entrée et une seule transition de sortie.

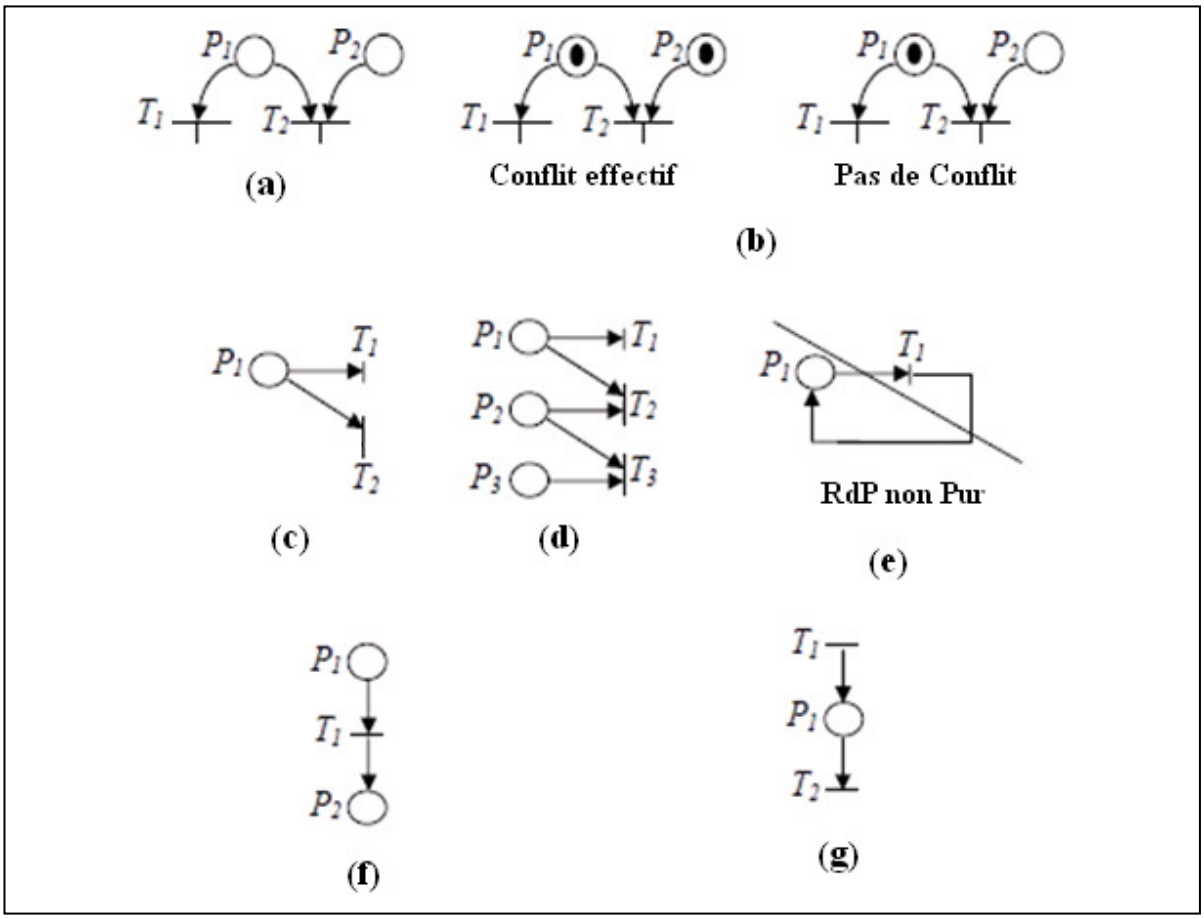


Figure 2.7: Détails des Propriétés Structurelles des RdPs

2.4.3.2 Les propriétés comportementales

Ces propriétés dépendent à la fois du marquage initial M_0 et de la structure du réseau. Il s'agit ici de faire ressortir les propriétés dynamiques du système étudié.

● ***Existence d'un marquage:*** Pour toute séquence de transitions "s", il existe un marquage M tel que celle-ci soit franchissable.

● ***Monotonie:*** L'augmentation de jetons dans les places d'un marquage préserve la possibilité de franchissement d'une séquence de transitions.

● ***Séquence répétitive:*** Une séquence de transitions est dite répétitive si pour tout marquage M tel que : M(s> alors M(s*>. La notion de séquence répétitive permet de définir une condition nécessaire et suffisante pour qu'un réseau marqué ait la possibilité d'être infiniment actif.

● ***Caractère borné:*** Cette propriété définit et caractérise la possibilité pour une place d'accumuler une quantité bornée ou pas de jetons au cours de l'évolution d'un réseau.

●***Place k-bornée, non bornée:*** Pour un réseau R et un marquage M_0, une place "p" du réseau marqué (R, M_0) est k-bornée si pour tout marquage M accessible depuis M_0, $M(p) \leq k$. Dans le cas contraire la place "p" est dite non-bornée.

● ***Réseau borné :*** Un réseau marqué est borné si toutes ses places sont bornées. Les réseaux 1-bornés sont appelés des réseaux "saufs".

● ***Activité d'un réseau :*** La notion d'activité d'un réseau recouvre deux classes de définitions. La première concerne l'activité individuelle des transitions, la seconde concerne l'activité globale d'un réseau (Indépendamment de transitions particulières).

● ***Pseudo-vivacité:*** Un réseau de Petri (R,M_o) est dit pseudo-vivant si pour tout marquage accessible depuis le marquage initial, il existe toujours une transition "t" qui puisse être franchie.

● ***Quasi-vivacité d'une transition:*** La quasi-vivacité d'une transition signifie que depuis le marquage initial, cette transition peut être franchie au moins une fois. Par conséquent, une transition qui n'est pas quasi-vivante est inutile.

● ***Quasi-vivacité d'un réseau:*** Un réseau est quasi-vivant si toutes ses transitions le sont.

● ***Monotonie et quasi-vivacité:*** La propriété de monotonie présentée plus haut, implique qu'une transition quasi-vivante pour (R, M) le reste pour (R, M').

● ***Vivacité :*** Les deux propriétés précédentes assurent une certaine correction du système mais elles ne permettent pas d'affirmer que, dans n'importe quel état atteint, le système dispose encore de

toutes ses fonctionnalités. Autrement dit, si toute transition peut toujours être ultérieurement franchie à partir d'un état quelconque du système. Par exemple, dans un réseau quasi-vivant une transition pourra être franchie une seule fois.

● ***Vivacité d'une transition:*** La vivacité d'une transition exprime le fait que quelque soit l'évolution du réseau à partir du marquage initial, le franchissement à terme de cette transition est toujours possible.

● ***Vivacité d'un réseau:*** Un réseau est vivant si toutes ses transitions le sont.

● ***État d'accueil:*** Un RdP possède un état d'accueil M_a pour un marquage initial M_0 si pour tout marquage accessible M_i il existe une séquence "s" telle que M_i (s>M_a.

● ***RdP réversible:*** Un RdP est réversible pour un marquage initial M_0 si M_0 est un état d'accueil.

● ***Absence de blocage:*** Cette propriété est plus faible que celle de vivacité. Elle implique seulement que le réseau a toujours la possibilité d'évoluer.

● ***Marquage puit:*** Un marquage puit est un marquage à partir duquel aucune transition n'est tirable. Un réseau marqué est sans blocage si aucun de ses marquages accessibles n'est un marquage puit.

2.4.4 Modélisation des systèmes complexes

Les RdPs permettent de modéliser un certain nombre de comportements importants dans les systèmes complexes tels que le parallélisme, la synchronisation, le partage de ressources, la mémorisation et la lecture d'information, la limitation d'une capacité de stockage, etc.

2.4.4.1 Parallélisme

Le parallélisme représente la possibilité que plusieurs processus évoluent simultanément au sein du même système. On peut provoquer le départ simultané de l'évolution de deux processus à l'aide d'une transition ayant plusieurs places de sortie.

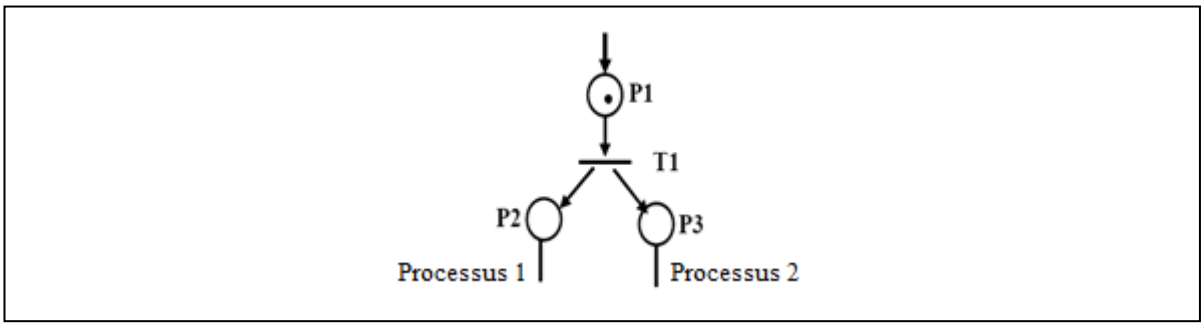


Figure 2.8: Structure du parallélisme

Le franchissement de la transition T_1 met un jeton dans la place P_2 (ce qui marque le déclenchement du processus 1) et un jeton dans la place P_3 (ce qui marque le déclenchement du processus 2).

2.4.4.2 Synchronisation

La synchronisation est modélisée sous deux formes :

- **Synchronisation mutuelle (Par rendez vous) :** La synchronisation mutuelle ou par rendez-vous permet de synchroniser les opérations de deux processus. La Figure 2.9 montre un exemple de deux processus. Le franchissement de la transition T_7 ne peut se faire que si la place P_{12} du processus1 et la place P_6 du processus 2 contiennent chacune au moins un jeton. Si ce n'est pas le cas, par exemple la place P_{12} ne contient pas de jetons, le processus 2 est bloqué sur la place P_6; il attend que l'évolution du processus 1 soit telle qu'au moins un jeton apparaisse dans la place P_{12}.

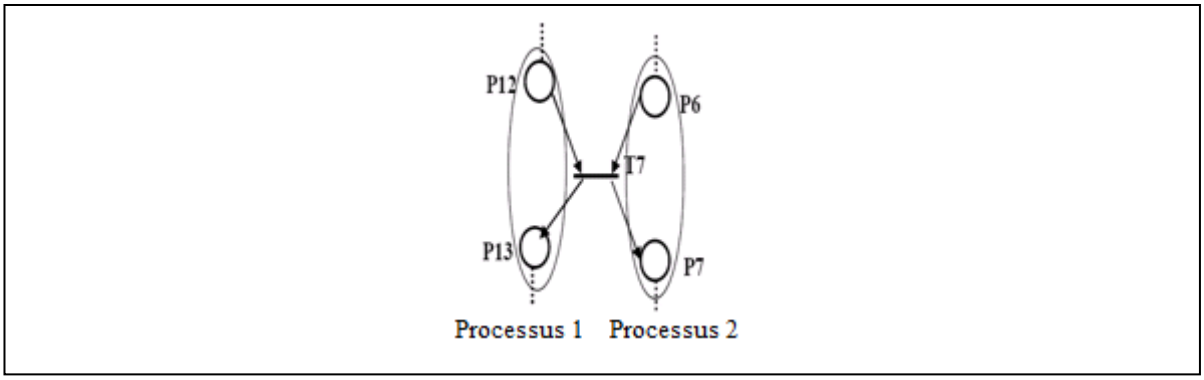


Figure 2.9: Synchronisation mutuelle

- **Synchronisation par signal (sémaphore):** Dans la synchronisation par signal, les opérations du processus 2 ne peuvent se poursuivre que si le processus 1 a atteint un certain niveau dans la suite de ses opérations. Par contre, L'avancement des opérations du processus 1 ne dépend pas de l'avancement des opérations du processus 2.

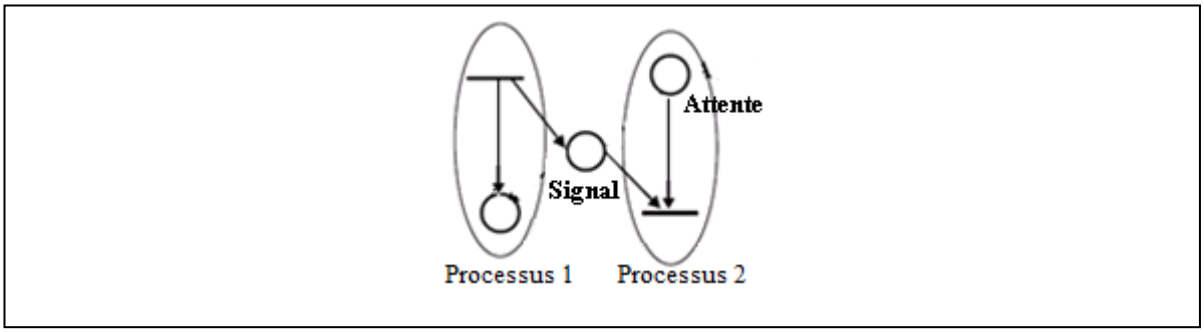


Figure 2.10: Synchronisation par sémaphore

Si la place "*Signal*" est marquée et la place "*Attente*" ne l'est pas, cela signifie que le processus 1 a envoyé le signal mais le processus 2 ne l'a pas encore reçu. Si, par contre, la place "*Signal*" n'est pas marquée et que la place "*Attente*" est marquée, cela signifie que le processus 2 est en attente du signal.

2.4.4.3 Partage de ressources

Cette structure va modéliser le fait qu'au sein du même système plusieurs processus partagent une même ressource en utilisant le principe de l'exclusion mutuelle. Dans la Figure 2.11, Le jeton dans la place P_0 présente une ressource mise en commun entre le processus 1 et le processus 2. Le franchissement de la transition T_{17} lors de l'évolution du processus 1 entraîne la consommation du jeton présenté dans la place P_0. La ressource que constitue ce jeton n'est alors plus disponible pour l´évolution du processus 2. Lorsque la transition T_{18} est franchie, un jeton est alors est placé dans la place P_0 : la ressource devient alors disponible pour l'évolution des deux processus.

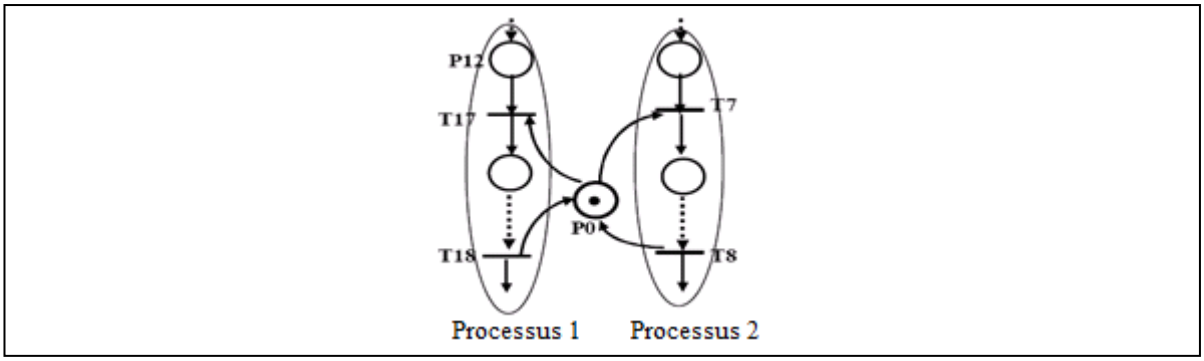


Figure 2.11: Synchronisation par Partage de ressources

2.4.4.4 Mémorisation

Dans tous les modèles, on peut compter le nombre de tirs d'une transition en utilisant une place sans sortie. Dans La Figure 2.12 les places "*Attente*" et "*Compteur*" peuvent servir à indiquer combien d'instances d'un processus sont en attente.

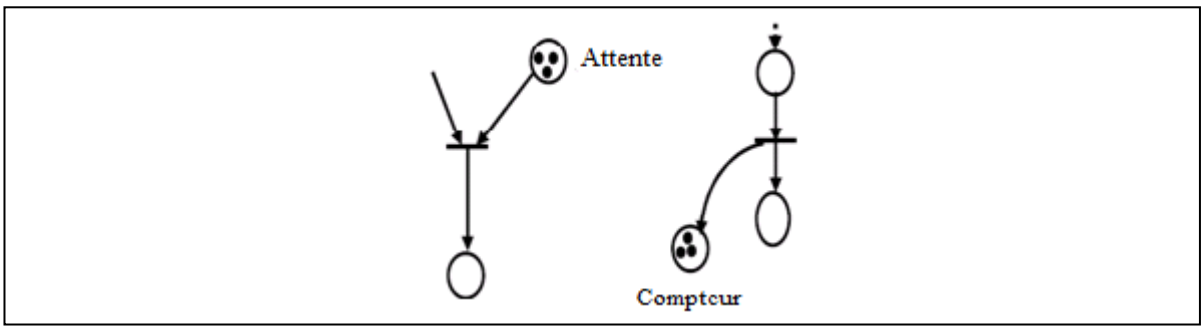


Figure 2.12 : Synchronisation par Mémorisation

2.4.5 Méthodes d'analyse des Réseaux de Petri

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 complexes. Parmi ces techniques nous pouvons citer l'analyse grâce aux graphes des marquages, l'analyse par algèbre linéaire, l'analyse structurelle, et l'analyse par réduction [Murata89].

2.4.5.1 Analyse par graphes des marquages

L'idée la plus naturelle pour étudier les propriétés d'un RdP est de construire le graphe de tous ses marquages accessibles. Le graphe des marquages accessibles est un graphe dont chaque sommet correspond à un marquage accessible et dont chaque arc correspond au franchissement d'une transition permettant de passer d'un marquage à l'autre. Deux situations peuvent alors 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 ce graphe.
2. ***Le graphe est infini.*** Dans ce cas, on construit un autre graphe appelé "graphe de couverture" permettant de déduire certaines propriétés.

2.4.5.2 Analyse par algèbre linéaire

L'analyse par algèbre linéaire permet d'étudier des propriétés d'un réseau de Petri 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.

2.4.5.3 Analyse par réduction

Pour l'analyse des propriétés d'un RdP, des difficultés peuvent apparaître dans l'utilisation du graphe de marquages, et même dans l'algèbre linéaire dans le cas d'un RdP de taille significative. L'objectif de la réduction est de présenter des règles permettant d'obtenir à partir d'un RdP marqué, un RdP marqué plus simple, c'est-à-dire avec un nombre réduit de places et un nombre réduit de transitions (en appliquant les règles de réduction). Ce dernier doit être :

1. Equivalent 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 [Murata89].

2.4.6 Extensions des Réseaux de Petri

Malgré ces qualités, Les RdPs classiques d'Adam Petri souffrent de certains reproches en termes d'expressivité tels que la distinction entre les jetons, l'expression de dimension temporelle ou le manque de structuration. Par conséquent, l'utilisation des RdPs produirait des modèles dont la taille croît rapidement avec la complexité du système, et qui deviennent rapidement difficiles à manipuler et/ou à analyser. Pour augmenter le niveau d'expressivité, les RdPs ont connu plusieurs extensions au fils des années. Les extensions les plus connues sont les réseaux de Petri colorés [Jensen 97, Jensen98], Les ECATNets [Bettaz92, Bettaz93], les réseaux de Petri algébriques, les réseaux de Petri Predicate/Transition [Genrich81], les G-Nets [Deng93], etc. Dans les chapitres suivants, nous donnerons un aperçu sur chaque type de réseau de Petri utilisé dans le cadre de cette thèse.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux des méthodes formelles, leurs techniques d'analyse et leur classification. Nous avons présenté aussi l'intérêt de leur intégration à l'IDM dans le but d'apporter la rigueur au processus de développement des systèmes complexes. Pour cela plusieurs techniques de vérification sont proposées. La technique de vérification par RdPs est la plus utilisée est cela est dû a leur représentation graphique avec l'aspect sémantique attribué au comportement des systèmes représentés.

Chapitre 03 :

La Modélisation Multi-Paradigme

3.1 Introduction

Dans les chapitres précédents, nous avons vu que l'IDM est une approche de développement des systèmes complexes dans la quelle les modèles et les transformations de modèles sont au cœur. Nous avons vu aussi que l'utilisation de multiples modèles du même système est à la fois inévitable et essentielle dans les différentes activités du cycle de développement, et que ces modèles sont décrits dans différents paradigmes.

Dans ce chapitre, nous nous intéressons donc à la problématique suivante: "Comment permettre aux concepteurs d'utiliser conjointement des multiples paradigmes de modélisation, tout en leur permettant de raisonner globalement sur le système et de vérifier certains aspects comportementaux?".

En complément à l'IDM, la modélisation multi-paradigme (*MMP*) [Vangheluwe02] propose une approche indépendante de tout domaine d'application qui permet d'adresser cette problématique. En plus des concepts liés à l'IDM, les deux concepts clés de la modélisation multi-paradigme sont la Modélisation Multi-Abstraction et la Modélisation Multi-Formalisme. La Modélisation Multi-Abstraction permet d'exprimer des modèles à des niveaux d'abstraction différents. Alors que la Modélisation Multi-Formalisme permet le couplage et la transformation entre des modèles décrits en utilisant des formalismes différents. L'approche MMP s'articule autour de plusieurs axes de recherche notamment la Modélisation Multi-Abstraction, la Modélisation Multi-Formalisme et la modélisation explicite de la Transformation de Modèles.

Dans ce chapitre, nous présentons le domaine de la Modélisation Multi-Paradigme. Nous passons alors en revue les différents axes de recherche de ce domaine. Nous présentons également l'outil AToM^3 : un outil de méta-modélisation et transformation de modèles.

3.2 Pourquoi la Modélisation Multi-Paradigme

"Modélisation Multi-formalisme & Modélisation Multi-abstraction"

La modélisation des systèmes complexes n'est pas une tâche facile, car ces systèmes ont souvent des composants et des aspects dont la structure et le comportement ne peuvent pas être décrits par un modèle unique dans un seul formalisme. Ceci implique que ces composants et aspects sont modélisés dans différents formalismes. Les diagrammes d'entité-association, les réseaux de Petri, les Automates et les diagrammes UML sont les plus utilisés dans ce cadre. Plusieurs approches sont possibles pour traiter cette diversité de formalismes [Vangheluwe02]:

L'approche *Super formalisme*: Elle consiste à utiliser un seul formalisme qui subsume l'ensemble des langages de modélisation et des formalismes nécessaires à la description d'un système. Bien qu'il existe quelques exemples de formalismes qui couvrent plusieurs domaines (par exemple, "Bond Graphs" pour les domaines de mécanique, hydraulique et électrique.), cette approche n'est ni possible, ni utile pour la plupart des cas des systèmes complexes.

L'approche *Co-simulation*: Dans cette approche, chaque composant ou aspect du système est modélisé en utilisant le formalisme et l'outil d'analyse approprié. L'évaluation et la vérification du comportement global du système sont effectuées généralement par une technique de Co-simulation. La Co-simulation est une simulation simultanée de différents composants d'un système et leurs interactions. Chaque modèle de composant est simulé avec un simulateur spécifique et l'interconnexion est assurée via des fonctions d'interface (ou d'entrée/sortie). L'échange de données d'entrée/sortie s'effectue à travers l'environnement de Co-simulation. Dans cette approche, l'évaluation des propriétés globale d'un système ne peut se faire qu'au niveau d'entrées/sorties entre ses composants. Par conséquent, il est impossible d'analyser le comportement global du système en utilisant les techniques des méthodes formelles qui pourrait être effectuées pour chaque composant séparément.

L'approche *Multi-Formalisme*: comme la Co-simulation, chaque composant est modélisé dans le formalise adapté mais le comportement global du système est évalué en transformant les modèles des composants vers un formalisme unique. Evidemment, le choix du formalisme cible dépend des propriétés du système qu'on souhaite étudier, et que ces propriétés doivent être préservées par les transformations. Il est facile de constater que la modélisation multi-formalisme englobe les deux autres approches: *super-formalisme* et de *Co-simulation*.

Pour rendre l'approche de Modélisation Multi-Formalisme applicable, il faudrait résoudre le problème de l'interconnexion et de l'interopérabilité des différents outils qui sont conçus pour les formalismes utilisés dans le processus de conception. En outre, dans certains systèmes, il est souhaitable d'avoir des formalismes très spécifiques au domaine avec leurs outils qui permettent la modélisation et l'analyse des comportements dans les termes du domaine. Malheureusement, la réalisation de ces outils est relativement complexe et coûteuse en termes de temps et d'argent [De Lara04].

Ce constat (de la réalisation coûteuse des outils) a amené une partie de chercheurs à travailler sur le principe de la *Méta-modélisation.* Selon ce principe, les formalismes de modélisation font eux-mêmes l'objet d'une modélisation. Pour concrétiser ce principe, des outils d'une nouvelle génération ont été développés: les *méta-outils*. Ces outils offrent la possibilité de

générer des outils personnalisés pour les formalismes en se basant sur des informations contenues dans leurs méta-modèles [De Lara02]. Donc, l'effort nécessaire pour créer un outil qui supporte un formalisme revient à définir son méta-modèle. D'autre part, les outils générés utilisent la même structure de données pour représenter les éléments des modèles. Par conséquent, la transformation entre les modèles est réduite à la transformation entre les structures de données qui représentent les modèles, ce qui assure l'interopérabilité entre les différents outils.

Le principal objectif de la Modélisation Multi-paradigme est de faciliter et d'automatiser l'utilisation conjointe de modèles pendant le cycle de développement de manière à rendre possible le raisonnement global et holistique sur le système [Vangheluwe02]. Plus précisément, cette approche cherche à apporter de nouvelles fonctionnalités aux différents modèles utilisés afin de faciliter la création de nouveaux espaces de modélisation plus adaptés à chaque activité dans le cycle de développement. La Modélisation Multi-paradigme repose sur trois directions de recherche orthogonales (voir la Figure 3.1):

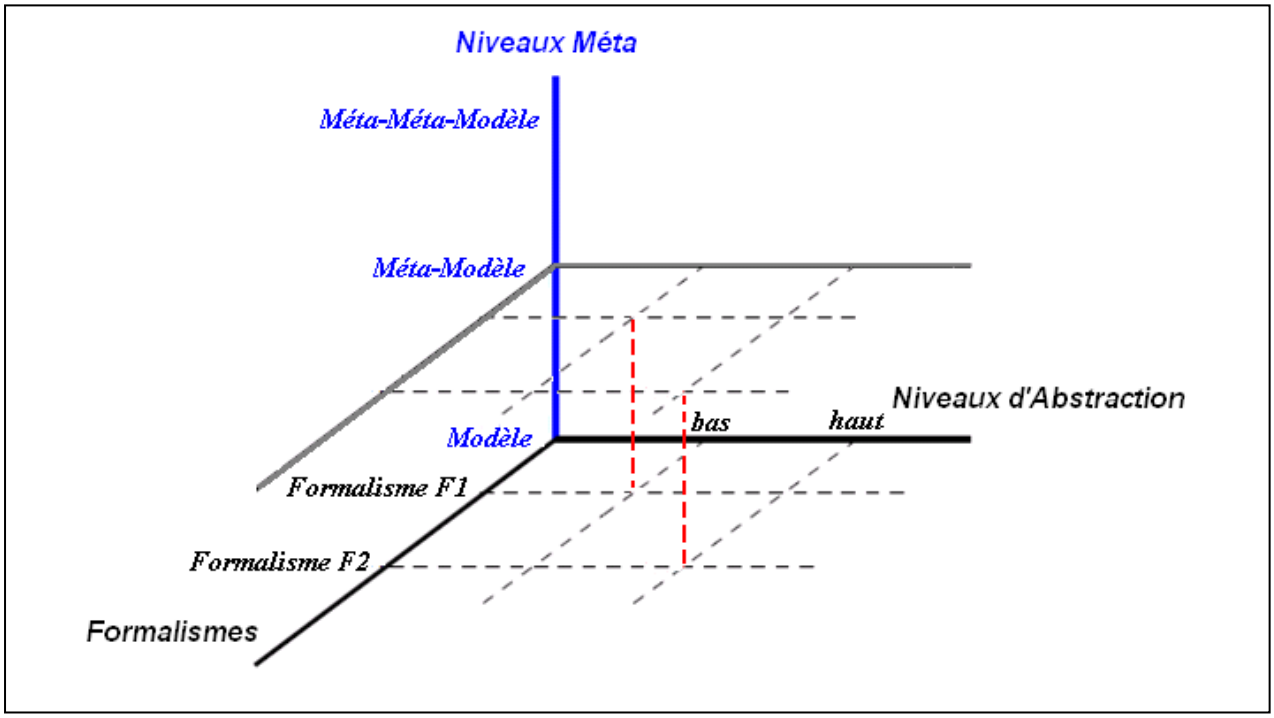


Figure 3.1: Modélisation Multi-paradigme: les trois directions

- *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.

3.3 Modélisation Multi-Paradigme: Les trois directions

La modélisation multi-paradigme est un champ de recherche multi-disciplinaire par essence. Il est donc naturel qu'elle implique diverses spécialistes dans des disciplines variées telles que l'automatique, le traitement du signal, l'ingénierie des langages de modélisation, la vérification formelle ou le développement de systèmes sur puce. Dans cette section, nous passons en revue les trois directions de recherche qui constituent la Modélisation Multi-Paradigme ainsi que le lien entre ces directions.

3.3.1 Modélisation Multi-Abstraction

Un modèle est une abstraction d'un système construite dans une intention particulière. Il doit pouvoir être utilisé pour aider à répondre à une question particulière sur le système. La réponse à cette question détermine la qualité du modèle. Par conséquent, un système peut être décrit par une infinité de modèles, chaque modèle est adapté à une tâche particulière. Ce mode de résonnement encourage les concepteurs à représenter les systèmes sous plusieurs aspects et à différents niveaux d'abstraction ou de détail ainsi que par le biais de différents formalismes.

Le niveau d'abstraction choisi, qui peut être différent pour chaque composants ou aspect du système, dépond principalement des objectifs des concepteurs. Ce niveau d'abstraction est déterminé par les connaissances disponibles sur le système, les questions qu'on se pose sur le système, le degré de la précision requise pour les réponses et les compétences des concepteurs eux-mêmes.

L'abstraction ne fait pas disparaître la complexité du système, elle permet simplement de l'appréhender selon des préoccupations particulières. La dérivation automatique de modèles à différent niveaux d'abstraction augmente considérablement la productivité ainsi que la qualité des modèles pour mieux maitriser et comprendre tous les détails du système. En plus, elle permet la convergence d'efforts de différentes modélisations. Il est à noter que les changements des niveaux d'abstraction peuvent impliquer éventuellement l'utilisation de différents formalismes. En général, le processus de changement d'abstraction peut être considéré comme un type de transformation qui préserve certaines propriétés (en général du comportement) du système pour une finalité spécifique.

Puisque les concepteurs basculent souvent entre les différents niveaux d'abstraction, l'outillage de ces opérations de transformation est indispensable dans les projets de conception des systèmes complexes. Le défi consiste alors à modéliser ces transformations, puis à utiliser ces modèles de transformation pour automatiser les opérations d'abstraction ou de raffinement. La

Figure 3.2 illustre l'opération d'abstraction et de raffinement dans le contexte de la modélisation multi-paradigme.

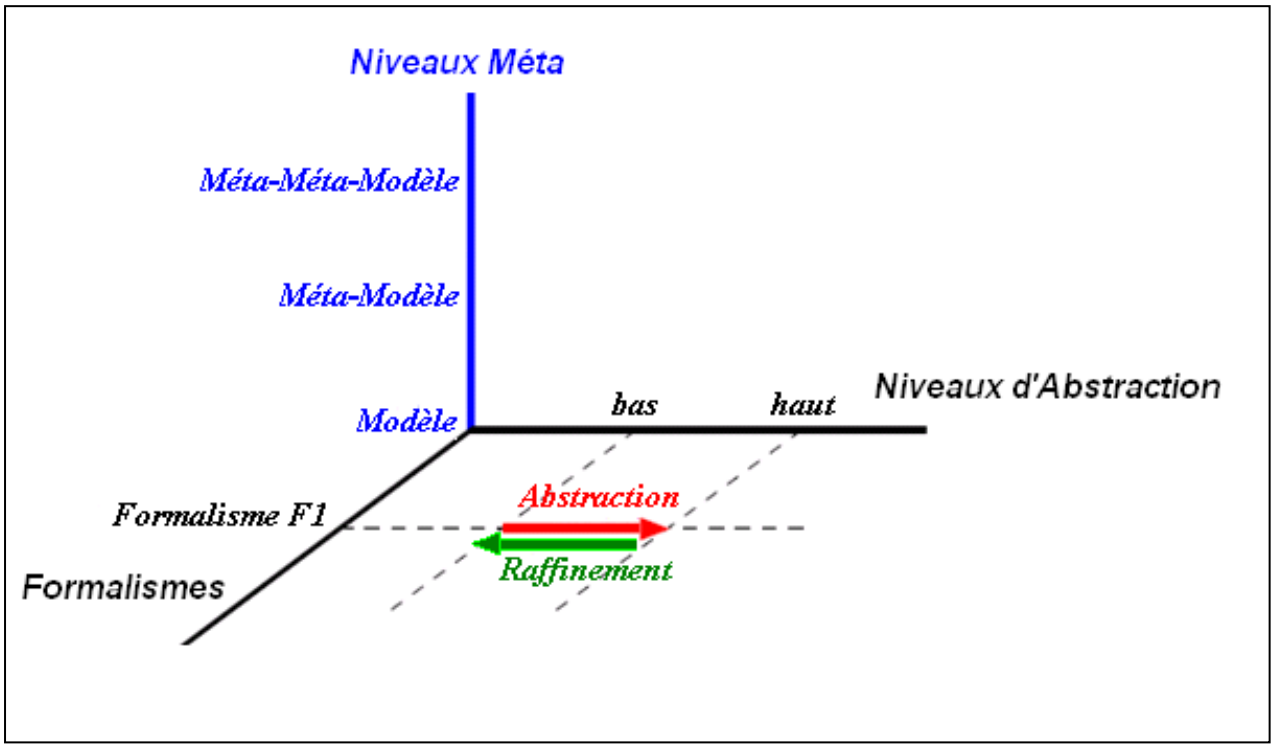


Figure 3.2 : Modélisation Multi-paradigme: l'Abstraction

3.3.2 Modélisation Multi-Formalisme

La complexité croissante des systèmes étudiés et les exigences des concepteurs ont amené à introduire de nombreuses catégories de formalismes dans le processus de développement. Le choix d'un formalisme adapté à un objectif spécifique est orthogonal à la sélection du niveau d'abstraction dans lequel le modèle sera décrit. Il est à noter que, de façon réciproque aux changements des niveaux d'abstraction, les changements de formalismes peuvent induire un éventuel changement dans le niveau d'abstraction. Le choix du formalisme est lié au niveau d'abstraction, la disponibilité des données utilisées pour construire le modèle, la disponibilité aussi d'analyseurs/simulateurs associés au formalisme ainsi qu'aux objectifs de modélisation considérés.

Selon R. Milner, dans [Milner93], il n'existe pas un modèle unique, ou un formalisme unique, permettant de modéliser tous les aspects ou tous les composants du système. Au contraire, les différents niveaux d'explication, les différentes théories et les différents formalismes ou langages dédiés sont nécessaires pour bien décrire les différents aspects ou les différents composants du système. L'utilisation de multiples formalismes n'est pas seulement de façon isolée, mais également de les combiner et de les intégrer afin de générer d'autres modèles dans des nouveaux espaces de modélisation adapté à des activités spécifique.

L'objectif de la modélisation multi-formalisme est de permettre l'utilisation de plusieurs formalismes et de gérer simultanément plusieurs modèles décrits dans ces formalismes qui, éventuellement, peuvent être mis en relation par des supports outillés.

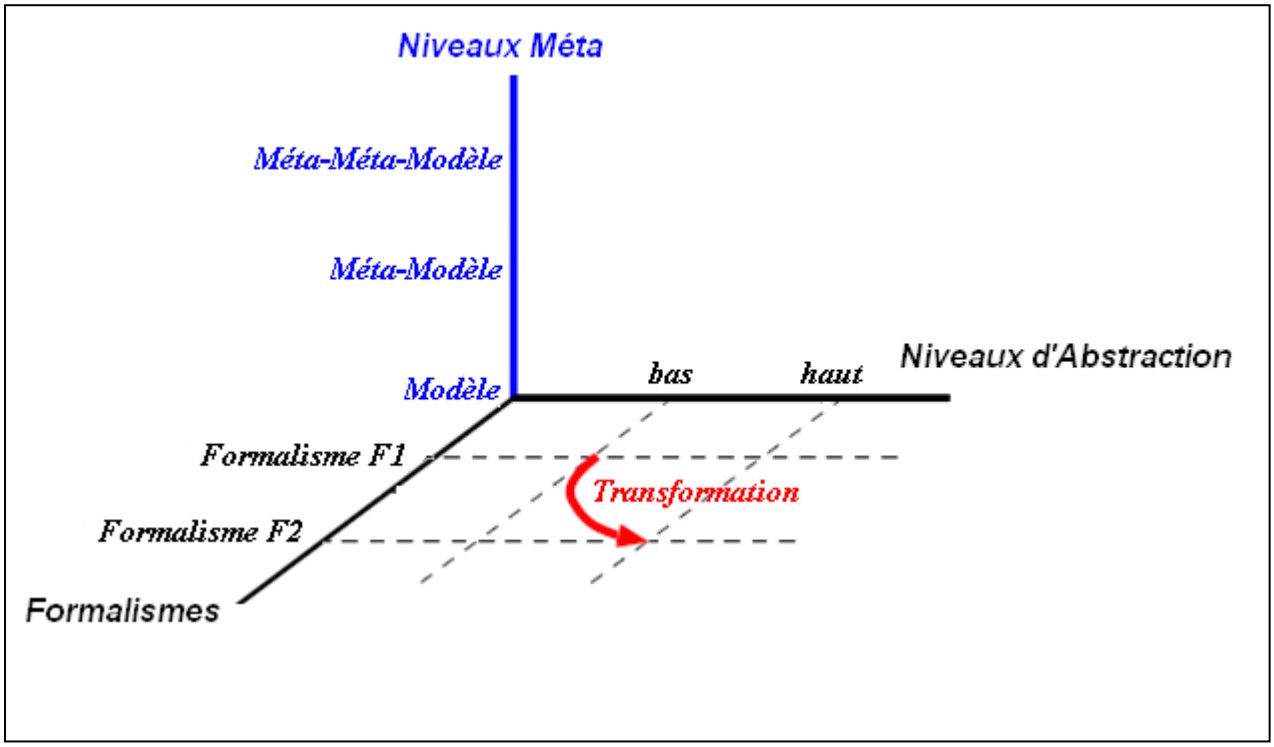


Figure 3.3 : Modélisation Multi-paradigme: Transformation de Modèle

3.3.3 Méta-Modélisation

La Méta-modélisation définit les constructeurs qui sont nécessaires à l'élaboration de modèles pour un domaine d'application particulier. Le méta-modèle est donc une représentation, dans le niveau méta, de différents concepts et contraintes utilisés dans les différents formalismes ou les langages de modélisation. Les environnements de méta-modélisation qui sont proposés (que ce soit dans l'industrie ou dans le monde académique) offrent un moyen efficace qui permet de définir des méta-modèles pour des formalismes ou des langages de modélisation, et ensuite d'exploiter ces méta-modèles pour générer automatiquement des outils de modélisation pour leurs modèles. La modélisation explicite des formalismes ou des langages est l'approche la plus efficace pour synthétiser des éditeurs de modélisation. La seule information à fournir est le méta-modèle du langage sans se préoccuper des détails d'implémentation de l'éditeur.

Les avantages de la méta-modélisation sont nombreux. Tout d'abord, un modèle explicite d'un langage de modélisation peut servir de *documentation*, qui peut être utilisée comme base pour analyser les propriétés des modèles décrits par ce langage. Cette approche a aussi l'avantage de préserver les acquis et aussi d'apporter une certaine flexibilité en fournissant les moyens d'adapter

un méta-modèle aux nouveaux besoins de modélisation spécifiques d'un domaine. A partir d'un méta-modèle existant, de nouveaux environnements pour de nouveaux langages peuvent être conçus, soit en modifiant simplement des parties du méta-modèle (ajout, suppression, modification des éléments et leurs relations et contraintes), soit en rajoutant des éléments et/ou des contraintes, sans en enlever les autres, pour avoir des nouveaux langages spécifiques à un domaine comme par exemple les profils UML. En ce sens, la création d'un nouvel outil de modélisation visuelle est plus rapide et plus fiable que le développement de cet outil à partir de rien [De Lara02].

La Modélisation Multi-paradigme peut être vue comme une généralisation de la Modélisation Multi-formalisme, augmentée par la méta-modélisation. Son objectif est de faciliter l'utilisation conjointe de modèles du système décrits dans plusieurs formalismes ou langages. La réalisation de cet objectif impose d'automatiser les différentes actions sur les modèles telles que le couplage, la composition, la transformation, etc.

L'interopérabilité entre les différents éditeurs générés, chacun pour son formalisme, est assurée par leurs méta-modèles. En effet, tous les éditeurs utilisent la même représentation interne des modèles. L'échange d'informations, la transformation de modèles et la combinaison de modèles revient à manipuler ces représentations internes.

3.3.4 Mettre en relation les trois dimensions

Dans les sous-sections précédentes, nous avons présenté les trois dimensions de la modélisation multi-paradigme ainsi que leurs avantages respectifs. La transformation de modèles permet de mettre en relation ces trois dimensions pour cumuler leurs avantages [Vangheluwe03]. La modélisation multi-paradigme consiste à explorer toutes les combinaisons possibles de ces trois dimensions afin de:

- Combiner, transformer et mettre en relation les différents formalismes.
- Utiliser des formalismes et des outils spécifiques au domaine d'application.
- Vérifier la cohérence entre les différentes vues/aspects du système.

Les différentes manipulations de modèles dans le contexte de la modélisation multi-paradigme sont:

- ***La transformation de formalismes:*** elle consiste à convertir un modèle décrit dans un formalisme, vers un autre modèle décrit dans un autre formalisme. La transformation de formalismes a de nombreuses utilisations. La plus rependue est la transformation vers un formalisme adapté à une finalité particulière. Souvent, cette transformation implique une

certaine perte d'information. Cette perte peut être bénéfique dans le sens où elle conduit à la réduction de la complexité du modèle.

- ***L'optimisation de Modèle:*** ces transformations ne changent pas le formalisme dans lequel le modèle est décrit. Leur application est pour réduire la complexité du modèle.
- ***La génération de code:*** ces transformations produisent une représentation textuelle du modèle (conforme à des contraintes syntaxiques du langage textuel) adapté pour un analyseur ou un simulateur.
- ***La simulation:*** consiste à animer les modèles pour leur donner une sémantique opérationnelle.

Du fait que les modèles et les méta-modèles sont décrits comme des graphes, les transformations entre modèles peuvent être décrites et effectuées par transformations de Graphes. Les transformations de graphes sont basées sur les grammaires de graphes pour décrire la transformation, et sur la réécriture des graphes pour effectuer la transformation. Cette approche de transformations de modèles est formelle et bien fondée sur les bases mathématiques de la théorie des graphes, des grammaires formelles et la réécriture de termes.

3.4 Les Transformations de Graphes

Les graphes et les diagrammes sont un moyen pratique, intuitif et direct pour la modélisation des systèmes complexes. Citons comme exemples, les diagrammes UML, les réseaux de Petri ou encore les automates finis. Si les graphes servent à visualiser les structures complexes des modèles d'une manière simple et intuitive, les transformations de graphes peuvent être exploitées pour spécifier comment ces modèles peuvent évoluer.

Les transformations de graphes [Rozenberg99] ont évolué dans la répercussion à l'imperfection dans l'expressivité des approches de réécriture classique comme *les grammaires de Chomsky* et *la réécriture de termes* pour s'y prendre avec les structures non linéaires.

Une transformation de graphe [Karsai04, Andries99, Rozenberg99] consiste en l'application d'une règle à un graphe et itérer ce processus. Chaque application de règle transforme un graphe par le remplacement d'une de ses parties par un autre graphe. Autrement dit, la transformation de graphe est le processus de choisir une règle d'un ensemble indiqué, appliquer cette règle à un graphe et réitérer le processus jusqu'à ce qu'aucune règle ne puisse être appliquée.

La transformation de graphe est spécifiée sous forme d'un modèle de grammaires de graphes. Ces dernières sont une généralisation des grammaires de Chomsky pour les graphes. Elles sont composées de règles. Une règle est constituée de deux parties, le *Left Hand Side* (*LHS*) et le

Right Hand Side (*RHS*). 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 [Rozenberg99].

Il existe plusieurs formalismes pour représenter les règles de transformations. Dans ce manuscrit, nous utilisons des règles qui sont exprimées visuellement. La Figure 3.4 montre le principe général de l'application d'une règle sur un graphe.

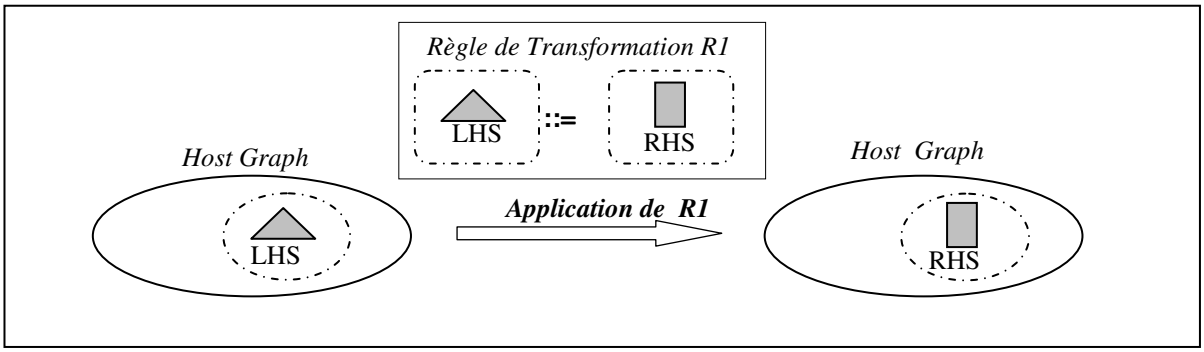


Figure 3.4 Principe de l'application d'une règle

Avant d'aborder les grammaires de graphes, nous allons rappeler quelques concepts de base relatifs à la théorie des graphes.

3.4.1 Notion de Graphe

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

3.4.1.1 Graphes non orientés

Un *graphe* est constitué de *sommets* qui sont reliés par des *arêtes*. Deux sommets reliés par une arête sont *adjacents*. Le nombre de sommets présents dans un graphe est appelé *l'ordre du graphe*. Le *degré* d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité.

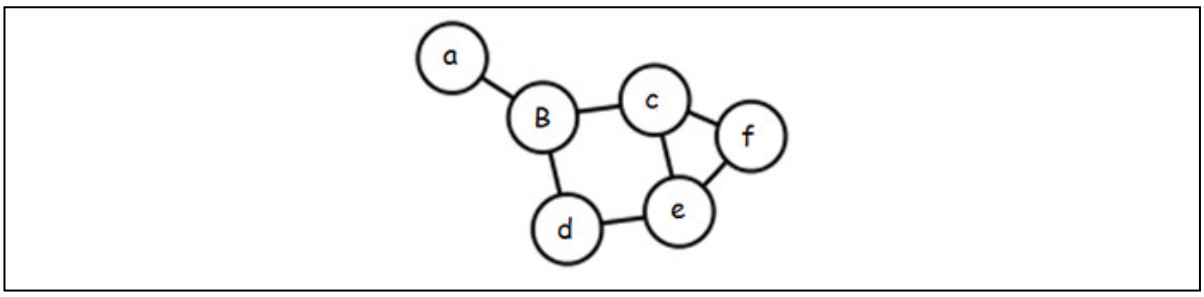


Figure3.5: Graphe non orienté

Un *sous-graphe* d'un graphe G est un graphe G' composé de certains sommets de G, ainsi que toutes les arêtes qui relient ces sommets.

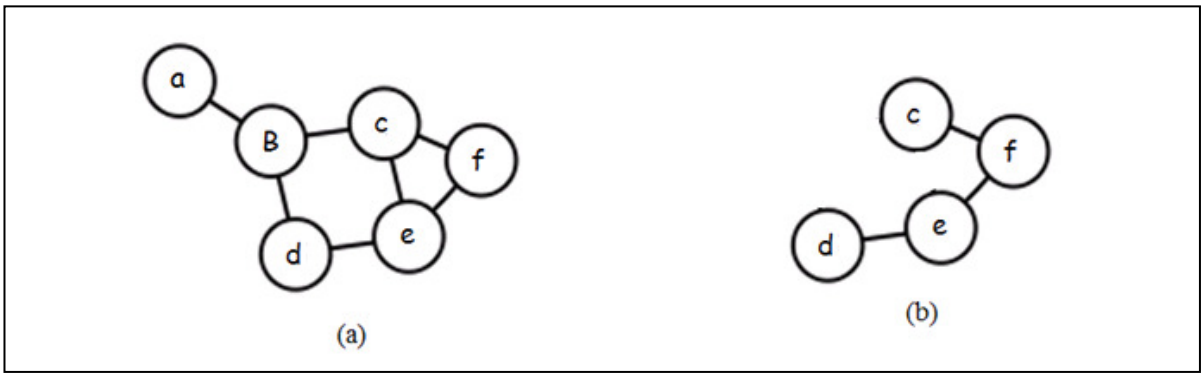


Figure3.6: (a) graphe, G (b) sous-graphe de G

3.4.1.2 Graphes orientés

Un *graphe orienté* est un graphe dont les arêtes sont orientées: on parle alors de l'origine et de l'extrémité d'une arête. Dans un graphe orienté une arête est dénommée *arc*.

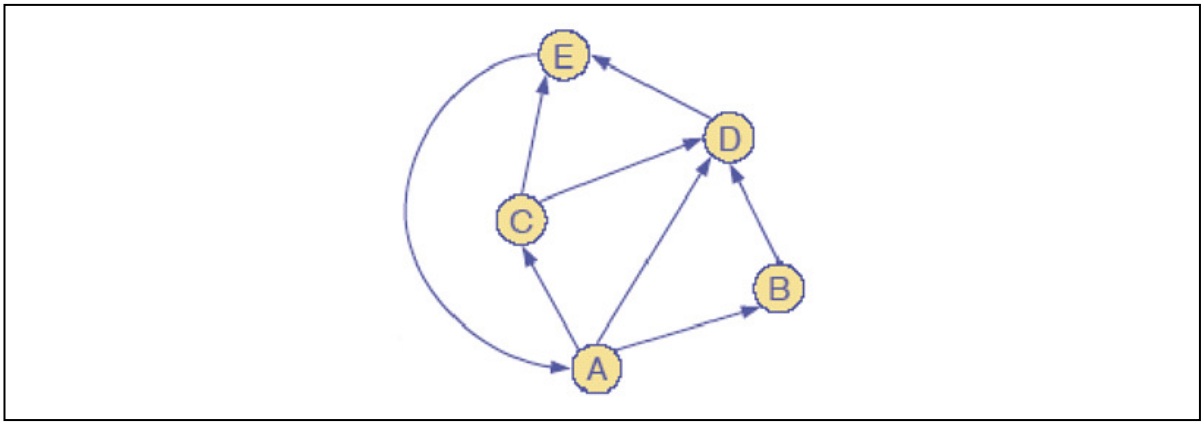


Figure3.7: Graphe orienté

Un *graphe étiqueté* est un graphe orienté, dont les arcs sont affectés d'étiquettes. Si toutes les étiquettes sont des nombres positifs, on parle de *graphe pondéré*.

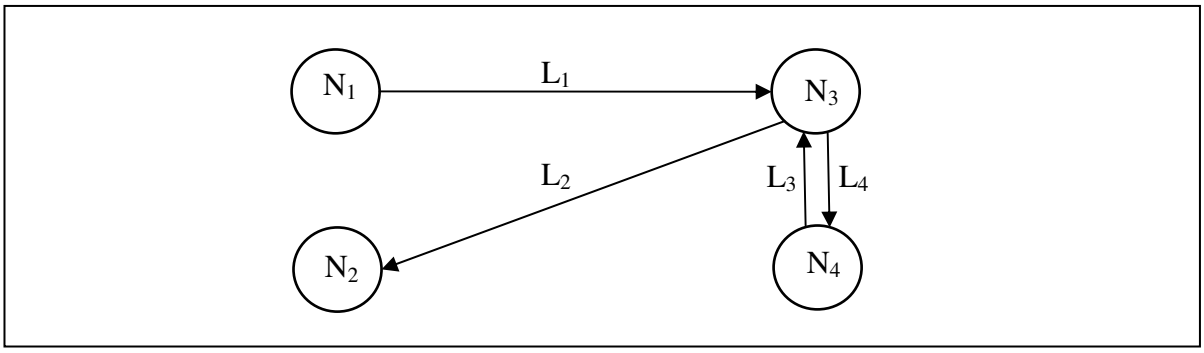


Figure 3.8: Graphe orienté étiqueté

Un *graphe attribué* est un graphe qui peut contenir un ensemble prédéfini d'attributs.

3.4.2 Grammaire de Graphe

Une grammaire de Graphe [Andries99] est généralement définie par un triplet:

$$GG = (P, S, T)$$

Où : $\begin{cases} P \text{ : ensemble de règles.} \\ S \text{ : un graphe initial.} \\ \text{T : ensemble de symboles.} \end{cases}$

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

3.4.2.1 Le principe de règles

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

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

3.4.2.2 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 produit peut être obtenu depuis le graphe d'origine G en passant par les cinq étapes suivantes :

1. Choisir une occurrence du graphe de coté 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'est-à-dire tout les arcs qui ont perdu leurs sources et/ou leurs destinations. Ce qui fourni 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 coté 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. Enfoncer le graphe de coté droit dans le graphe de contexte de L suivant la relation d'enfoncement *emb*. Pour chaque arcs retiré incident avec un sommet v dans D et avec un sommet v' dans l'occurrence de L dans G et pour chaque sommet v" dans R, un nouvel arc est établi (avec la même étiquette) incident avec l'image de v et le sommet v" à condition que (v', v") appartient a *emb*.

L'application de r à un graphe G pour produire un graphe H est appelée une dérivation directe depuis G vers H à travers r, elle est dénotée par G $\Rightarrow$ H.

En donnant les notions de règle et de dérivation directe comme étant les concepts élémentaires de la transformation de graphe, ont peut définir les systèmes de transformation de graphe et la notion de langages engendrés.

3.4.2.3 Système de transformation de graphe

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 [Rozenberg99].

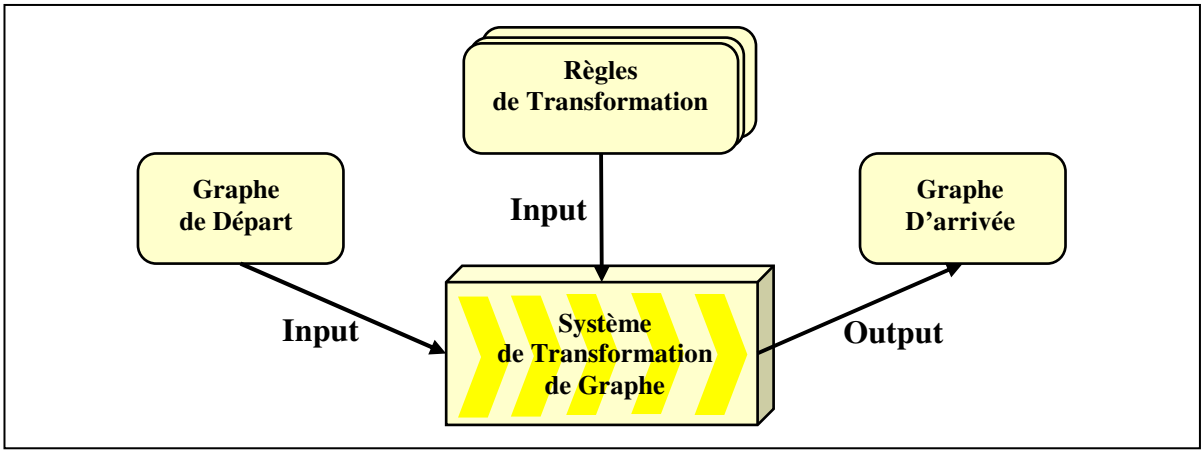


Figure 3.9: Système de réécriture de graphes

Cette approche de transformations de modèles a plusieurs avantages par rapport aux autres approches [Rozenberg99]:

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

3.4.2.4 Langage engendré

Supposons que nous avons un ensemble donné P de règles et un graphe G_0, une séquence de transformations de graphe successive : $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ est une dérivation à partir de G_0 vers G_n par les règles de P (à condition que toutes les règles utilisées appartiennent à P). G_0 est le graphe initial et G_n est le graphe dérivé de la séquence de transformation.

L'ensemble des graphes dérivés à partir d'un graphe initial S en appliquant les règles de P qui sont étiquetées par les symboles de T, est dit *langage engendré* par P, S et T et on écrit L(P,S,T).

3.5 AToM^3 : L'outil de Modélisation Multi-paradigme

AToM^3 (*A Tool for Multi-formalism and Meta-Modelling*) [De Lara02] est un outil de modélisation multi-paradigme développé par le laboratoire MSDL (Modelling, Simulation and Design Lab) à l'université de McGill Montréal, Canada. Comme il a été implémenté en Python [Python], il peut être exécuté, sans aucun changement, sur toutes les plateformes où un interpréteur de Python est disponible (Linux, Windows et MacOS).

AToM^3 est un outil visuel qui possède une couche de Méta-modélisation qui lui permet une modélisation graphique des différents formalismes. A partir de la méta-spécification, AToM^3 génère un outil pour la manipulation des différents modèles décrits dans le formalisme spécifié. Les modèles ont une représentation interne sous forme de graphes de syntaxe abstraite, et ils ont une autre représentation externe visuelle sous forme de graphes de syntaxe concrète.

D'autre part, les manipulations des modèles dans AToM^3 sont définies par des modèles de transformation dans le formalisme des Grammaires de Graphes. Les règles des grammaires de graphes sont spécifiées par la syntaxe concrète des modèles.

Nous expliquons à travers le formalisme des automates à états finis (*FSA*) comment utiliser AToM^3 pour générer un éditeur de modèles FSM, et comment éliminer l'indéterminisme de ces modèles par le biais d'une grammaire de graphes.

3.5.1 La Méta-modélisation avec AToM^3

Dans l'outil AToM^3, les formalismes sont décrits visuellement par des graphes en utilisant le formalisme des Entité-Relation (*ER*) ou le formalisme des diagrammes de classes d'UML. Ceci signifie que pour méta-modéliser de nouveaux formalismes on peut utiliser le formalisme ER ou les diagrammes de classes d'UML. Dans le cadre de cette thèse nous avons choisi d'utiliser le modèle des diagrammes de classes.

Afin de pouvoir spécifier entièrement les formalismes de modélisation, les méta-formalismes peuvent être étendus par l'expression de contraintes qui ne peuvent pas être exprimées visuellement. Les contraintes fournissent une vue sur la manière dont une entité du formalisme est liée à une autre pour qu'il ait un sens. Les contraintes sont exprimées sous une forme textuelle. Pour ce faire, certains systèmes (dont AToM^3) utilisent le langage OCL d'UML. Puisque AToM^3 est implémenté en Python, on peut également utiliser du code Python pour exprimer les contraintes.

Dans la modélisation au niveau méta-modèle, les entités qui doivent apparaître sur les modèles sont spécifiées avec leurs attributs, leurs relations, leurs cardinalités, leurs contraintes ainsi que leurs apparences graphiques.

A partir de la méta-spécification, AToM^3 génère un éditeur graphique pour la manipulation des différents modèles décrits dans le formalisme spécifié.

La Figure 3.10 montre le méta-modèle des automates à états finis, alors que la Figure 3.11 présente l'éditeur graphique généré automatique avec AToM^3 pour le formalisme des automates à états finis (FSA) [De Lara02].

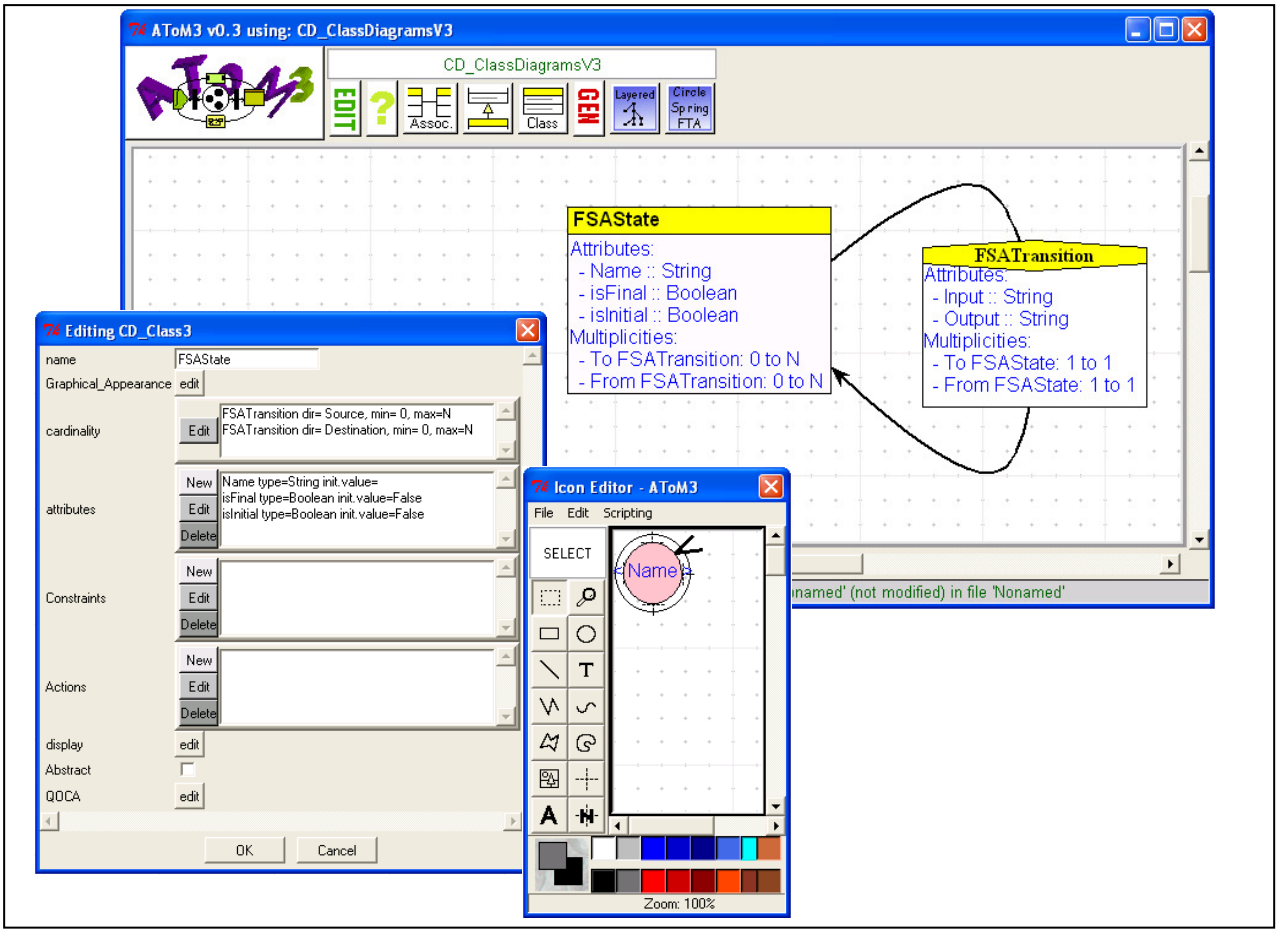


Figure 3.10: Méta-Modèle des automates à états finis

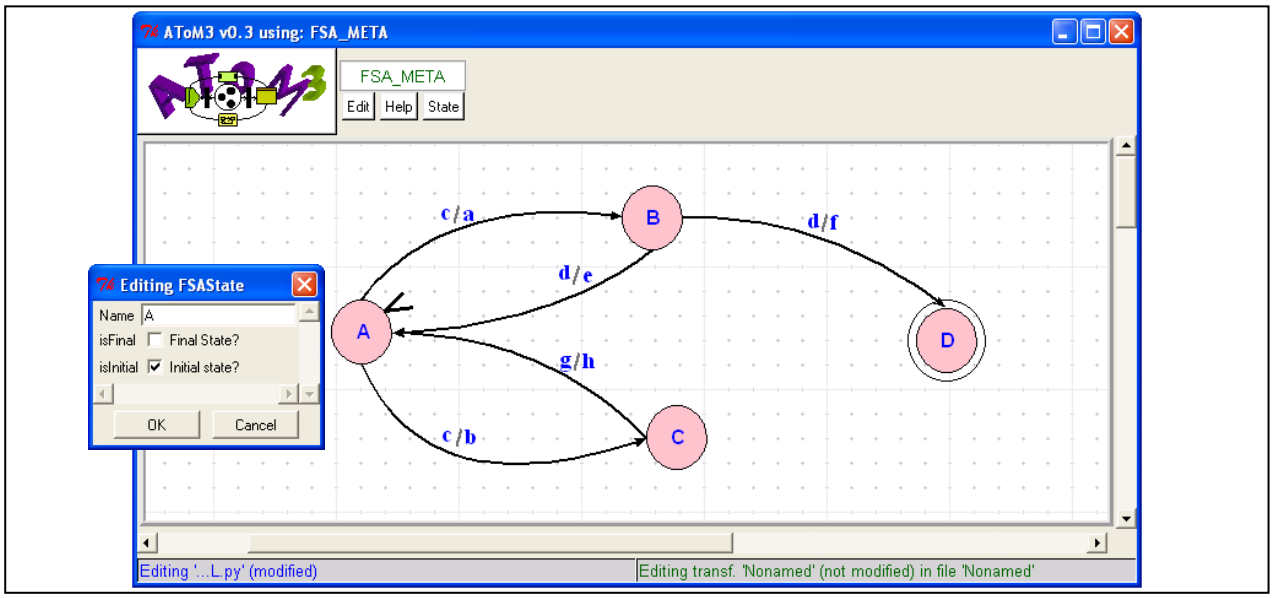


Figure 3.11: Editeur graphique généré pour les automates à états finis

3.5.2 La Transformation de Modèles.

Pour la transformation de modèles, AToM^3 possède un système de réécriture de graphes qui applique itérativement les règles d'une grammaire de graphes pour guider la procédure de transformation. Les règles sont spécifiées et classées selon des priorités attribuées par l'utilisateur pour diriger le choix de la règle à appliquer. A chaque itération, toutes les règles sont testées suivant l'ordre ascendant de leurs priorités.

Dans les règles, les attributs des éléments de la partie LHS doivent avoir des valeurs qui seront comparées avec les attributs des éléments du modèle pendant le processus de correspondance (matching). Un attribut peut prendre une valeur spécifique ou n'importe quelle valeur (<ANY>). En plus, les liens de traçabilité entre les éléments source et cibles sont représentés par des étiquettes (des numéros). Si une étiquette d'un élément apparaît dans la partie LHS mais pas dans la partie RHS, alors l'élément sera supprimé lors de l'application de la règle. Inversement, si une étiquette d'un élément n'apparaît que dans la partie RHS, alors l'élément sera crée par cette règle. Enfin, si une étiquette d'un élément apparaît à la fois dans la partie LHS et la partie RHS dans une règle, le nœud sera maintenu.

Après l'application d'une règle, les valeurs des attributs des éléments maintenus ou nouvellement crées par la règle seront définis. Dans l'outil AToM^3 il y a plusieurs possibilités. Si l'élément est déjà présent dans la partie LHS, les valeurs de ses attributs sont copiées (<COPIED>). On a également la possibilité de leurs donner des valeurs spécifiques ou d'assigner un programme Python pour calculer ses valeurs (<SPECIFIED>), éventuellement, en utilisant les valeurs des autres attributs.

Chaque règle peut aussi avoir des conditions supplémentaires d'application et des actions à effectuer.

En plus de règles de transformations, la grammaire peut utiliser aussi une action initiale et une autre finale. L'action initiale (finale, respectivement) spécifie les actions à exécuter avant (après, respectivement) l'application des règles.

AToM^3 offre la possibilité à l'utilisateur de créer une grammaire, de charger une grammaire, de modifier une grammaire et d'exécuter une grammaire. L'exécution de la grammaire sur un modèle en entrée produit un modèle en sortie.

La Figure 3.12 représente la grammaire de graphes qui permet d'éliminer l'indéterminisme dans un modèle des automates à états finis.

L'application de cette grammaire de graphe sur le modèle de la Figure 3.11 est illustrée dans la Figure 3.13.

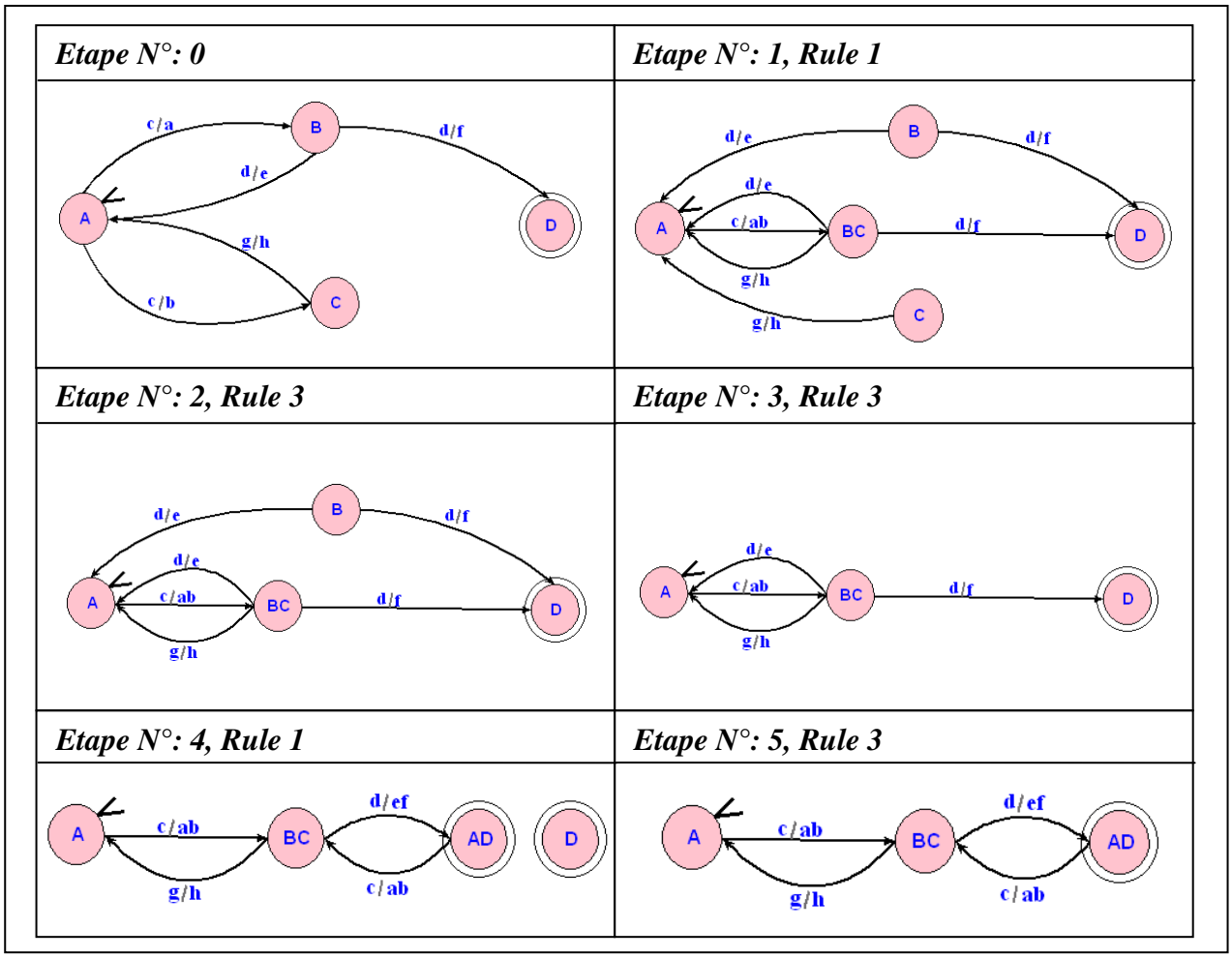
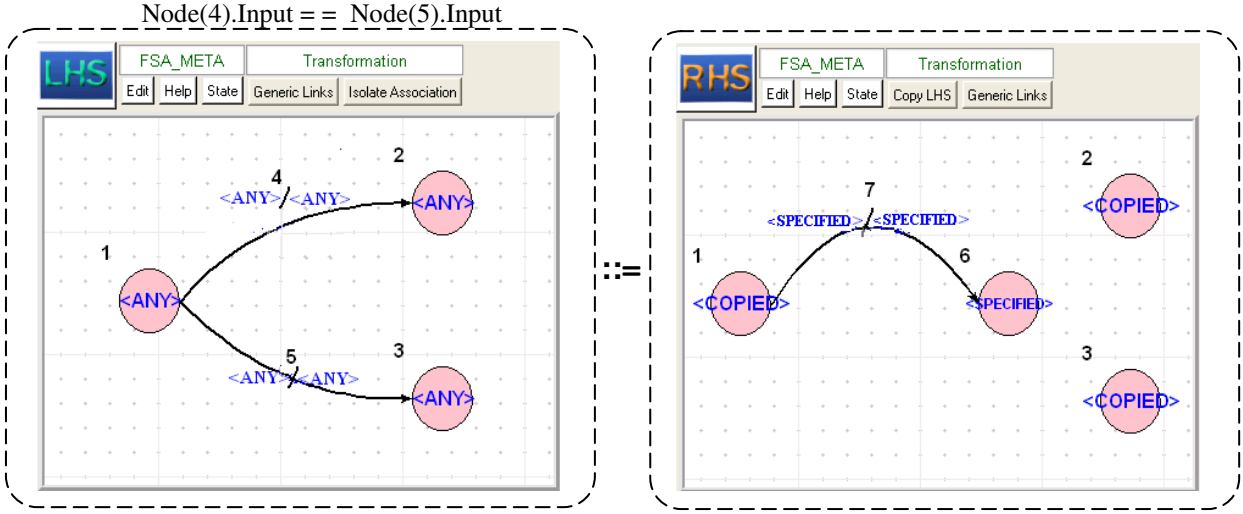


Figure 3.13: Application de la Grammaire

1- Rule 01. Priority:1

Condition

Node(4).Input = = Node(5).Input



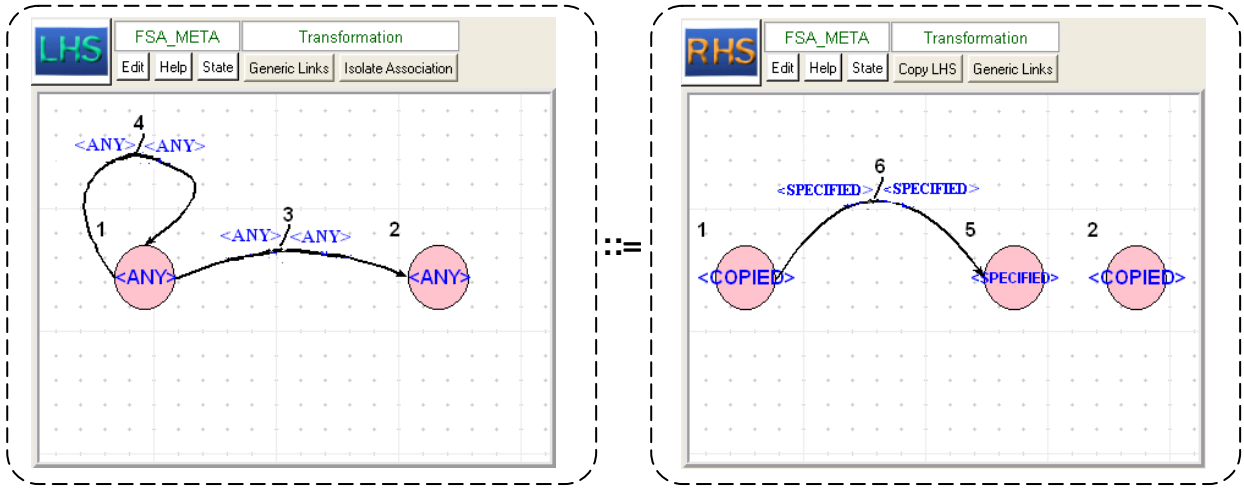
Action

Node(6).In_ connections.Append (Node(2).In_ connections)
Node(6).In_connections.Append(Node(3).In_ connections)
If (Node(2).isFinal or Node(3).isFinal): Node(6).isFinal = True

2- Rule 02. Priority:2

Condition

Node(3).Input = = Node(4).Input



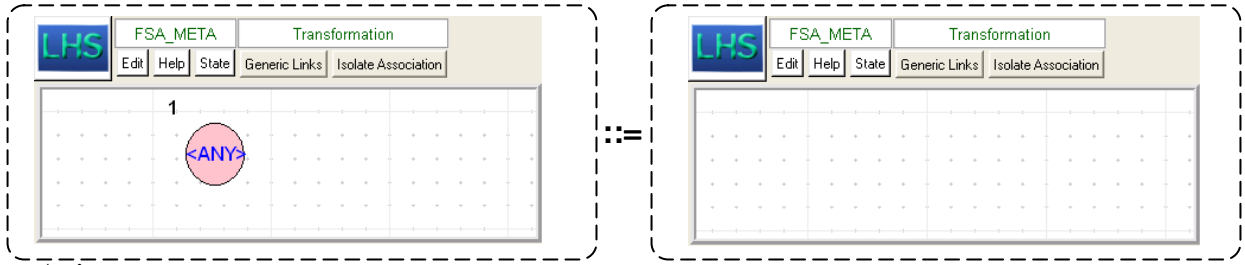
Action

Node(5).In_ connections.Append (Node(2).In_ connections)
If (Node(2).isFinal) : Node(6).isFinal = True

3- Rule 03. Priority:3

Condition

Node(4). In_ connections = = []



Action

Figure 3.12: Grammaire de Graphe: élimination de l'indéterminisme

3.6 Autres approches basées sur la transformation de graphes

Mis à part l'outil AToM^3, Il existe plusieurs systèmes implantant des systèmes de réécriture de graphes. Dans cette section, nous présentons quelques outils actuels de transformation de graphes. Il est important de noter qu'aucun de ces outils ne possède sa propre couche de Méta-modélisation.

3.6.1 PROGRES

PROGRES [Schürr99, Ranger08] (*Programmed Graph Rewriting Systems*) est l'un des premiers systèmes permettant de faire des transformations de graphes. Il a été développé en Allemagne à l'université d'Aachen. Cet outil repose sur l'approche orientée logique des grammaires de graphes. Les règles sont décrites de manière visuelle par une partie gauche et une partie droite.

3.6.2 Fujaba

FUJABA [Burmester04], (*From UML to Java and Back Again*) a pour but de fournir un environnement de génération de code Java et de rétro-conception. Il utilise UML comme langage de modélisation visuel. Durant la dernière décennie, l'environnement de FUJABA est devenu une base pour beaucoup d'activités de recherche notamment dans le domaine des applications distribuées, les systèmes de bases de données ainsi que dans le domaine de la modélisation et de la simulation des systèmes mécaniques et électriques.

3.6.3 AGG

AGG (*Attributed Graph Grammar system*) [AGG06, Taentzer03] est un environnement de développement basé sur les transformations des graphes. Cet outil, réalisé à l'Université Technique de Berlin (Technische Universität Berlin) et développé depuis 1997, est un des outils de transformation de graphes parmi les plus aboutis et les plus cités dans la littérature.

AGG est développé en utilisant le langage de programmation Java. Il offre une interface graphique permettant de construire les graphes et les règles de transformation et de réaliser des simulations pas à pas et quelques vérifications élémentaires. Les graphes considérés sont orientés et possèdent des nœuds et des arcs étiquetés par des objets Java.

3.6.4 GReAT

GReAT (*Graph Rewriting and Transformation*) [Balasubramanian06] permet la définition de transformations unidirectionnelles de plusieurs modèles source vers plusieurs modèles cible. GReAT utilise principalement une notation graphique pour définir les règles de transformations.

Cependant, certaines parties sont spécifiées textuellement comme les expressions d'initialisation des attributs et les conditions d'application. Ces modèles sont conformes à des méta-modèles spécifiés dans une notation et peuvent être créés avec l'outil de méta-modélisation GME [Lédeczi01].

3.6.5 VIATRA2

VIATRA2 (*Visual Automated model TRAnsformations*) [Varró06] est un plugin Eclipse développé depuis 2005 à l'université de Budapest. Le logiciel utilise un langage de modélisation particulier pour représenter les modèles appelé VPM [Varró03]. Pour définir les règles de transformations, l'utilisateur peut utiliser des motifs récursifs et des motifs de négations représentant les conditions d'application négatives. Un des avantages du système est la possibilité d'ordonnancement dans l'application des règles à l'aide d'une machine à états abstraite.

3.7 Conclusion

Dans ce chapitre, nous avons présenté le concept de la modélisation multi-paradigme qui représente une nouvelle approche de conception des systèmes complexes. Nous avons vu les différentes directions de recherche de ce domaine. Une brève introduction aux transformations de graphes a été faite. Elles représentent une approche de transformation de modèles qui permet de mettre en relation les dimensions de la modélisation multi-paradigme. Nous avons également présenté ATOM^3, l'outil de transformation utilisé dans notre travail. Les concepts présentés dans ce chapitre constituent un background nécessaire pour la compréhension de nos contributions dans le cadre de cette thèse et qui seront présentées dans les chapitres suivants.

Chapitre 04 :

Un Support Outillé de Manipulation et de Simulation des ECATNets

4.1 Introduction

Les ECATNets (*Extended Concurrent Algebraic Term Nets*) [Bettaz93] sont une catégorie des réseaux de Petri algébriques basés sur une combinaison saine des réseaux de Petri de haut niveau et des types abstraits algébriques. Les réseaux de Petri sont utilisés pour leur pouvoir d'exprimer graphiquement la concurrence et l'aspect dynamique. Les types abstraits algébriques sont utilisés pour leur puissance de décrire abstraitement les données. Leur association dans un seul cadre est motivée par le besoin de spécifier explicitement à la fois le comportement et les structures des données des processus.

La sémantique des ECATNets est définie en termes de la logique de réécriture [Meseguer96]. La logique de réécriture offre une base solide et rigoureuse pour toute démarche d'analyse des propriétés des systèmes étudiés, ce qui permet la simulation et la vérification formelle en utilisant le système Maude [Meseguer00] qui est à la fois un langage de spécification et un environnement d'exécution basé sur la logique de réécriture.

Dans ce chapitre, nous présentons un support outillé basé sur la méta-modélisation et les grammaires de graphes pour la manipulation et la simulation des ECATNets en utilisant AToM^3. L'outil proposé permet d'exploiter les avantages des ECATNets tels que l'aspect graphique, la simplicité et la lisibilité avec ceux du système Maude. Le principe de notre travail est comme suit: l'outil permet à l'utilisateur de faire une édition graphique d'un modèle ECATNet et de générer automatiquement sa description équivalente dans Maude. Enfin, le système Maude est appelé pour exécuter le modèle ECATNet et retourner le résultat de la simulation sous forme de marquage du modèle décrit en langage Maude.

Le reste de ce chapitre est organisé comme suit: La section 2 est une présentation générale des ECATNets, de leur description dans la logique de réécriture et de leur représentation dans Maude. Le méta-modèle des ECATNets ainsi que leur éditeur graphique sont décrits dans la section 3. La section 4 présente la grammaire de graphes qui assure la génération automatique des descriptions équivalentes en Maude. Les étapes du simulateur des ECATNets sont décrites dans la section 5. Finalement, la section 6 conclut le chapitre.

4.2 ECATNets

Les ECATNets [Bettaz93] sont une catégorie de modèle de réseau/données combinant les forces des réseaux de Petri avec ceux des types de données abstraits. Les places sont identifiées par des multi-ensembles des termes algébriques. Les arcs entrants de chaque transition t, c.-à-d. (p, t),

sont marqués par deux inscriptions IC(p, t) (*Input Conditions*) et DT(p, t) (*Destroyed Tokens*). Les arcs sortants de chaque transition t, c.-à-d. (t, p'), sont marqués par CT(t, p') (*Created Tokens*). Finalement, chaque transition t est marquée par une inscription TC(t) (*Transition Condition*) (voir la Figure 4.1).

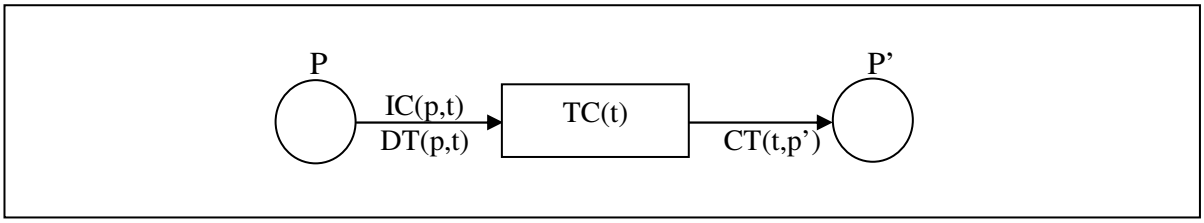


Figure 4.1 : ECATNets générique

IC(p, t) indique une condition qui valide une transition t, DT(p, t) indique le marquage (un multi-ensemble) qui doit être enlevé de p lors du franchissement de t, CT(t, p') indique le marquage (un multi-ensemble) qui doit être ajouté à p' lors du franchissement de t et TC(t) représente un terme booléen qui indique une condition additionnelle pour le franchissement de la transition t.

L'état courant d'un ECATNet est donné par l'union des termes ayant la forme (p, M(p)). Par exemple, soit un réseau contenant une transition t ayant une place d'entrée p marquée par le multi-ensemble a ⊕ b ⊕ c et une place de sortie vide p', l'état distribué s de ce réseau est donnée par le multi-ensemble : s = (p, a ⊕ b ⊕ c).

Une transition t est franchissable quand les trois conditions suivantes sont simultanément vraies :

- ✓ *1ère condition:* chaque IC(p, t) pour chaque place d'entrée p de t est tirable.
- ✓ *2ème condition:* TC(t) est vraie.
- ✓ *3ème condition:* l'addition de CT(t, p') à chaque place de sortie p' ne doit pas avoir comme conséquence d'excéder sa capacité quand cette capacité est finie.

Le franchissement de t va mener à:

- ↳ DT(p, t) est complètement enlevé (cas positif) de la place d'entrée p. Notons que dans le cas non positif, on enlève les éléments en commun entre DT(p, t) et M(p).
- ↳ CT(t, p') est ajouté à la place de sortie p'.

D'un point de vue syntaxique, la différence entre un ECATNet simplifié et un réseau de Petri ordinaire est que les jetons dans les premiers ne sont pas imperceptibles car ils sont exprimés en matière de termes algébriques possédant une identité, une syntaxe et une sémantique. D'ici les jetons peuvent porter des informations complexes grâce à leur nature en tant que termes

algébriques. Du point de vue sémantique, Le franchissement d'une transition et les conditions de ce franchissement sont formellement exprimés par des règles de la logique de réécriture ([Meseguer96], [Meseguer00]) où:

- Les axiomes sont en réalité des règles de réécriture conditionnelles décrivant les effets des transitions comme des types élémentaires de changement.
- Les règles de déduction nous permettent d'avoir des conclusions valides des ECATNets à partir des changements.

Une règles de réécriture est de la forme "t: u → v if boolexp" :

Où : { t: est la transition liée à cette règle. ; u et v: sont respectivement les côtés gauche et droit de la règle. ; boolexp: est un terme booléen. }

Plus précisément, u et v sont des multi-ensembles de pairs de la forme $(p, [m]_{\oplus})$, telle que p est une place de réseau, $[m]_{\oplus}$ est un multi-ensemble des termes algébriques, et l'union multi-ensemble sur ces termes. Notons que les termes sont considérés comme singletons. L'union des multi-ensembles sur les pairs $((p, [m]_{\oplus})$ est dénotée par $\otimes$. Un état d'un ECATNet lui-même est représenté par un multi-ensemble de telles paires.

La logique de réécriture offre aux ECATNets une version textuelle simple et pratique pour leur analyse sans perdre leur sémantique formelle (rigueure mathématique et raisonnement formel). Plusieurs travaux ont été réalisés dans le contexte de l'implémentation des ECATNets dans un système basé sur cette logique. L'implémentation dans le système Maude en est un exemple [Bettaz93]. Maude fournit une plate-forme permettant un développement facile et une exécution efficace des outils des ECATNets.

4.2.1 Formes des Règles de Réécriture [Bettaz92]

4.2.1.1 Cas positif sans Arcs Inhibiteurs

IC(p,t) est de la forme $[m]_{\oplus}$

Cas 1. $[IC(p, t)]_{\oplus} = [DT(p, t)]_{\oplus}$

La forme de cette règle est : $t : (p, [IC(p ,t)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$

Cas 2. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$

Cette situation correspond à vérifier que IC(p, t) est inclus dans M(p) et, retirer DT(p,t) de M(p). La forme de cette règle est :

$$t : (p, [IC(p, t)]_{\oplus}) \otimes (p, [DT(p, t)]_{\oplus}) \rightarrow (p, [IC(p, t)]_{\oplus}) \otimes (p', [CT(t, p')]_{\oplus}).$$

Cas 3. $[IC(p, t)] \cap [DT(p, t)] \neq \phi_M$

Cette situation correspond au cas le plus général. Elle peut cependant être résolue d'une manière élégante en remarquant qu'elle pourrait être apportée aux deux cas précédents. Ceci est réalisé en remplaçant la transition t de ce cas par deux transitions t1 et t2. Ces transitions, une fois franchies concurremment, donnent le même effet global que la transition t. IC(p, t) est éclaté en deux multi-ensembles IC1(p, t1) et IC1(p, t2). De même, DT(p, t) sera éclaté en deux multi-ensembles DT1(p, t1) et DT1(p, t2) :

$$IC(p, t) = IC1(p, t1) \cup IC1(p, t2), DT(p, t) = DT1(p, t1) \cup DT1(p, t2).$$

Les quatre multi-ensembles obtenus doivent réaliser $IC1(p, t1) = DT1(p, t1)$ et $IC2(p, t2) \cap DT2(p, t2) = \phi_M$. Le franchissement de la transition t est identique au franchissement en parallèle des deux transitions t1 et t2.

4.2.1.2 Cas Général

IC(p,t) est de la forme $[m]_\oplus$

Cas 1. $[IC(p,t)]_\oplus = [DT(p,t)]_\oplus$

La forme de cette règle est : $t : (p, [IC(p, t)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus)$

Cas 2. $[IC(p, t)]_\oplus \cap [DT(p, t)]_\oplus = \phi_M$

Cette situation est équivalente à celle de vérifier que IC(p, t) est inclus dans M(p) et retirer DT(p, t) de M(p). La forme de cette règle est:

$$t : (p, [IC(p, t)]_\oplus) \otimes (p, [DT(p, t)]_\oplus) \rightarrow (p, [IC(p,t)]_\oplus) \otimes (p', [CT(t, p')]_\oplus)$$

Cas 3. $[IC(p, t)]_\oplus \cap [DT(p, t)]_\oplus \neq \phi_M$

Ce cas peut être résolu aussi en combinant les deux cas précédents. Ceci peut se faire en remplaçant la transition de ce cas par deux transitions, une fois franchies concurremment, donnent le même effet global. En réalité, ce remplacement montre comment spécifier une situation donnée en deux niveaux d'abstraction.

IC(p, t) est de la forme ~$[m]_\oplus$

La forme de cette règle est:

$$t : (p, [DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus) \text{ if } ([IC(p, t)]_\oplus \setminus ([IC(p, t)]_\oplus \cap [M(p)]_\oplus)) = \phi_M \rightarrow [\text{false}]$$

IC(p, t) = empty

Cette règle est de la forme **:**

$$t : (p, [DT(p, t)]_\oplus \cap [M(p)]_\oplus) \rightarrow (p', [CT(t, p')]_\oplus) \text{ if } [M(p)]_\oplus \rightarrow \phi_M$$

Telle que la capacité C(p) est finie, la partie conditionnelle de règle de réécriture va inclure le composant suivant :

$$[CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus} \cap [C(p)]_{\oplus} \rightarrow [CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus} \ (\textbf{\textit{Cap}})$$

Dans le cas où il y a une condition de transition TC(t), celle-ci est représentée dans la logique de réécriture par une règle non conditionnelle.

4.2.2 Représentation des ECATNets dans Maude

Considérons la version textuelle des ECATNets dans Maude. Le module générique qui décrit des opérations de base d'un ECATNet est le suivant :

```
fmod  GENERIC-ECATNET is
sorts  Place Marking GenericTerm .
op  mt : ->  Marking .
op <_;_> :  Place GenericTerm -> Marking .
op _._ :  Marking Marking -> Marking [assoc comm id: mt] .
endfm
```

Comme illustré dans ce code, mt dénote le marquage vide dans un ECATNet. L'opération "<_;_>" permet la construction des marquages élémentaires. Les sous-lignes dans cette définition indiquent les positions des paramètres. Le premier paramètre de cette opération est une place et le second est un terme algébrique quelconque qui peut être dans cette place. L'opération $\oplus$ n'est pas définie dans ce module, l'opération "_._" implémentant l'opération $\otimes$ est suffisante grâce au concept de décomposition. Si une place contient plusieurs termes, par exemple (p, a $\oplus$ b), alors on peut l'écrire < p ; a > $\otimes$ < p ; b >.

4.2.3 Méthodes et outils d'analyse des ECATNets

La puissance des ECATNets réside dans leur sémantique basée sur la logique de réécriture et de leur langage de spécification Maude. Du moment qu'un ECATNet peut être transformé en une spécification Maude, le système Maude peut être alors utilisé comme moyen pour l'analyse et la simulation des systèmes modélisés par des ECATNets.

En plus de l'utilisation de Maude comme simulateur des ECATNets, plusieurs outils d'analyse des ECATNets ont été proposés. Nous pouvons citer ici l'analyse structurelle basée sur les verrous et les trappes [Bettaz96], la technique d'analyse par graphe de couverture [Boudiaf04] et la technique de réduction des ECATNets [Boudiaf06].

4.3 Méta-Modèle des ECATNets

Pour construire un éditeur graphique pour les modèles en AToM^3, nous devons définir un méta-modèle pour eux. Le méta-formalisme utilisé dans ce travail est le modèle de diagrammes de classes d'UML et les contraintes sont exprimées en code Python.

Pour méta-modéliser les ECATNets dans l'outil AToM^3, nous avons proposé deux classes reliées par deux associations comme le montre la Figure 4.2 [Kerkouche08, Kerkouche09].

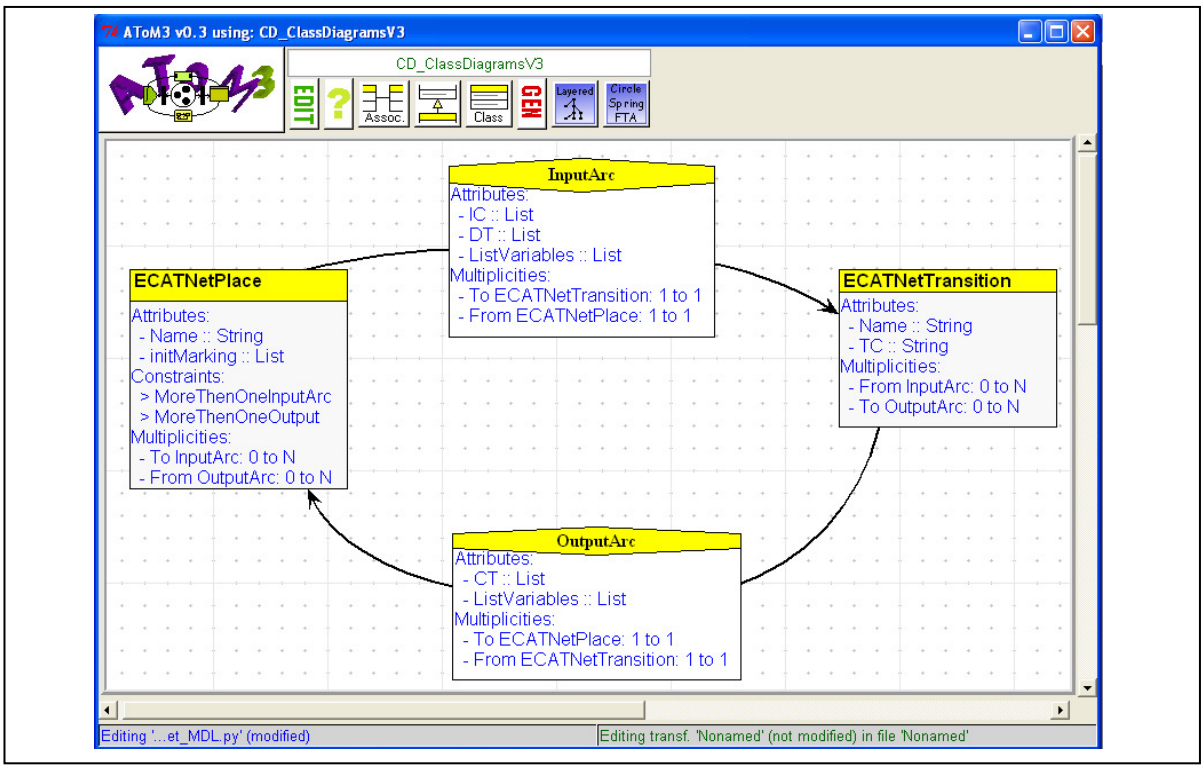


Figure 4.2: Méta-modèle des ECATNets

Classe "ECATNetPlace": représente une place visualisée à l'aide d'un cercle. Elle possède un attribut *name* qui est visuel et un autre attribut *initMarking* qui représente une liste de termes algébriques symbolisant le marquage initial de la place. Cet attribut est visualisé dans le cercle représentant la place. Pour chaque instance de cette classe, il y a deux contraintes qui interdisent l'existence de plus d'un arc d'entrée provenant de la même transition et de même pour l'arc de sortie.

Classe "ECATNetTransition": elle est visualisée par un rectangle. Elle possède un attribut *name* et un attribut *TC* (*Transition Condition*) qui sont visuels. Une instance de la Classe "ECATNetTransition" peut être reliée à une instance de la classe "ECATNetPlace" par un arc d'entrée ou de sortie.

Association "InputArc": elle relie la classe "ECATNetPlace" avec la classe "ECATNetTransition". Elle est visualisée par une flèche bleue portant les deux attributs *IC* (*Input Condition*) et *DT* (*Destroyed Tokens*) qui représentent respectivement la liste des conditions d'entrée et les jetons détruits à partir de la place d'entrée.

Association "OutputArc": cette association permet de déterminer l'arc sortant d'une transition vers une place. Elle est visualisée à travers une flèche rouge portant l'attribut *CT* (*Created Tokens*) où l'on trouve la liste des différents jetons créés dans la place destination de l'arc.

Grâce au méta-modèle des ECATNets, nous allons générer un éditeur graphique pour ces derniers. L'outil généré va nous permettre d'éditer et de manipuler les déférents modèles des ECATNets (voir la Figure 4.3).

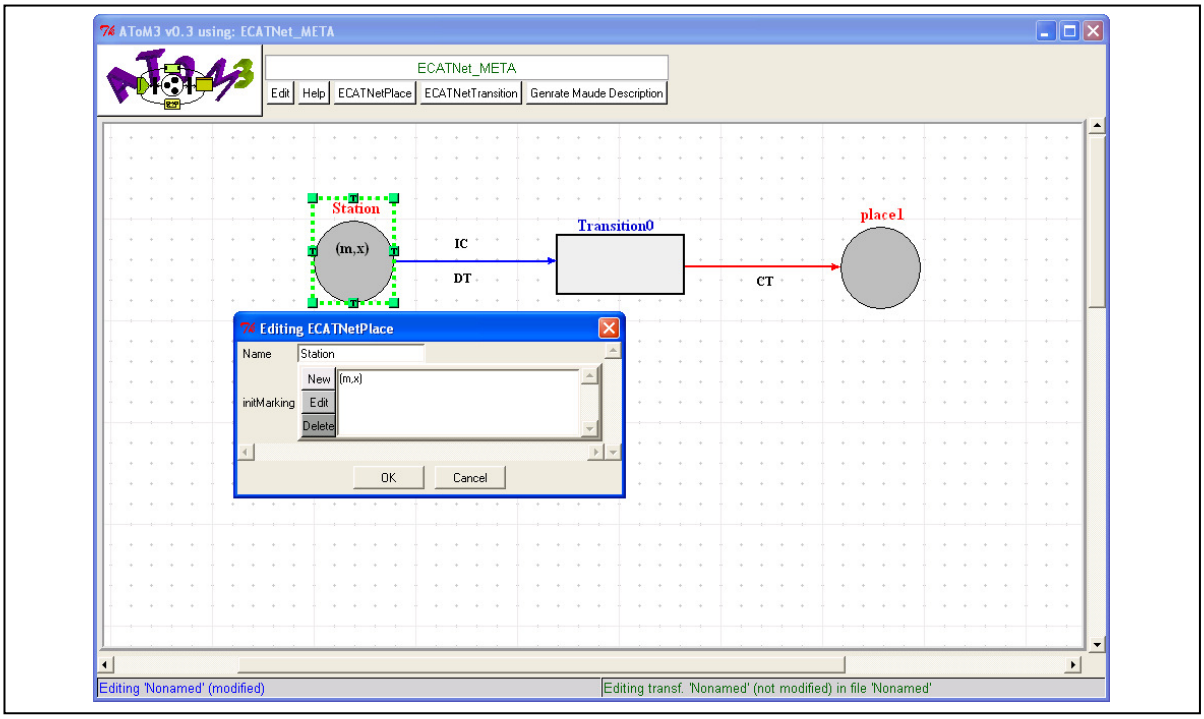


Figure 4.3: Outil de modélisation des ECATNets

Pour générer automatiquement les descriptions équivalentes des modèles ECATNets en langage Maude, nous allons définir une grammaire de graphes qui permet de guider la génération de code Maude dans l'outil AToM^3.

4.4 Génération de la description Maude équivalente

Afin de produire les spécifications Maude équivalentes aux modèles ECATNets créés par notre outil, nous avons proposé une grammaire de graphes avec six règles qui seront exécutées dans un ordre ascendant de leurs priorités [Kerkouche09a] par le système de réécriture de graphes

de l'outil AToM^3. L'application de cette grammaire à un modèle ECATNet conduit à la génération d'un fichier " *.maude"* qui contient sa description textuelle équivalente en Maude. Dans ce cas, nous sommes concernés par la génération automatique de code. Par conséquent, aucune règle ne va changer le modèle ECATNet. Autrement dit, dans toutes les règles de notre grammaire, la partie gauche est identique à la partie droite.

L'action initiale de notre grammaire de graphes consiste à créer un fichier " *.maude*" qui contiendra la description équivalente en Maude et à décorer toutes les places et les transitions du modèle par des variables temporaires utilisées pour spécifier les conditions des règles. Pour chaque transition, deux attributs sont ajoutés: "*current*" and "*visited*". L'attribut "*current*" est utilisé pour identifier la transition dans le modèle pour laquelle le code est en cours de génération, alors que l'attribut "*visited*" est utilisé pour indiquer que le code pour la transition est généré. Dans les éléments de type place, deux attributs sont également ajoutés : "*fromVisited*" et "*toVisited*". L'attribut "*fromVisited*" indique que cette place est traitée comme une place d'entrée pour la transition qui est en cours de traitement, alors que l'attribut *toVisited* indique que la place est traitée comme une place de sortie. Tous ces attributs temporaires sont initialisés à 0.

Les règles de notre grammaire sont présentées dans la Figure 4.4 et décrites comme suit :

Règle N°1: genLHS_rl (priorité 1). Cette règle est appliquée pour localiser une place d'entrée (non encore traitée) de la transition en cours de traitement (*current = = 1*) afin de générer le code Maude correspondant à cette place dans la partie gauche de la règle de réécriture Maude équivalente à la transition courante.

Règle N°2: betweenLHSandRHS (priorité 2). Cette règle génère en langage Maude les symboles " =>" qui sépare la partie gauche de la partie droite de la règle de réécriture Maude équivalente à la transition courante. Cette règle est appliquée lorsque toutes les places d'entrées de la transition courante sont traitées.

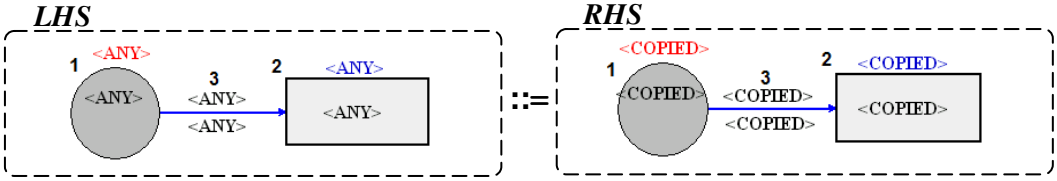
Règle N°3: genRHS_rl (Priorité 3). Cette règle est appliquée pour localiser une place de sortie (non encore traitée) de la transition en cours de traitement (*current = = 1*) afin de générer le code Maude correspondant à cette place dans la partie droite de la règle de réécriture Maude équivalente à la transition courante.

Règle N°4: genTC (priorité 4). Cette règle est appliquée pour générer le code Maude correspondant au contenu de l'attribut TC de la transition courante et de marquer cette transition comme "*visited == 1*".

Rule N°1- genLHS_rl. Priority:1

Condition : ((Node(1). fromVisited = = 0) and (Node(2).current = = 1) and (Node(2).visited = = 0))

LHS ***RHS***

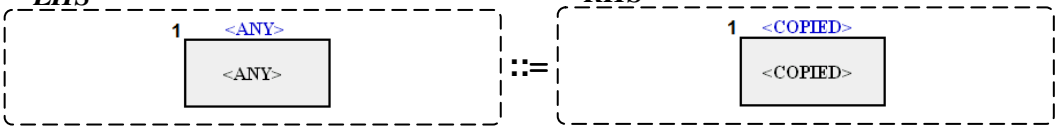


Action : Node(1). fromVisited = 1

Rule N°2- betweenLHSandRHS. Priority:2

Condition : ((Node(1).current = = 1) and (Node(1).visited = = 0))

LHS ***RHS***

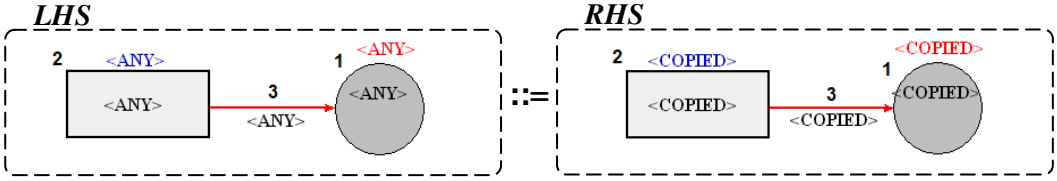


Action : Node(1).current =2

Rule N°3- genRHS_rl. Priority:3

Condition : ((Node(1). toVisited = = 0) and (Node(2).current = = 2) and (Node(2).visited = = 0))

LHS ***RHS***

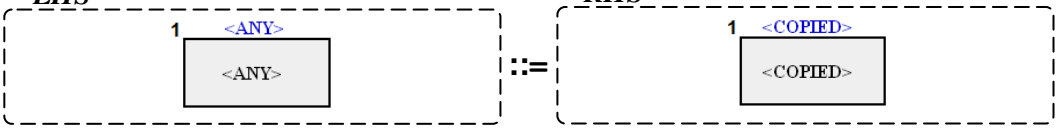


Action : Node(1). toVisited = 1

Rule N°4- genTC. Priority:4

Condition : ((Node(1).current = = 2) and (Node(1).visited = = 0))

LHS ***RHS***

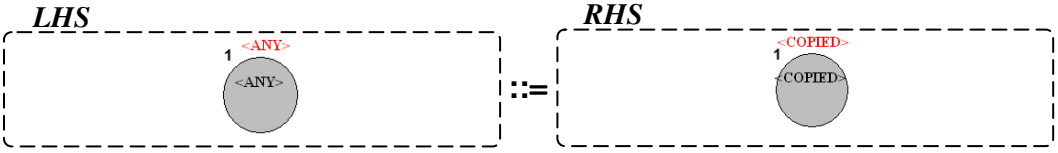


Action : Node(1).visited =1
Node(1).current = 0

Rule N°5- InitialisePlace. Priority:5

Condition : ((Node(1).fromVisited = = 1) or (Node(1).toVisited = = 1))

LHS ***RHS***

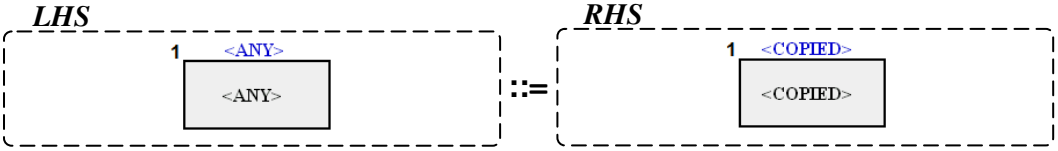


Action : Node(1). fromVisited = 0
Node(1). toVisited = 0

Rule N°6- SelectTransition. Priority:6

Condition : ((Node(1).current = = 0) and (Node(1).visited = = 0))

LHS ***RHS***



Action : Node(1).current =1

Figure 4.4: Grammaire de Graphes ECATNets2Maude

Règle N°5: InitialisePlace (priorité 5). Cette règle est appliquée pour localiser et initialiser les attributs temporaires dans les places pour traiter la prochaine transition.

Règle N°6: SelectTransition (priorité 6). Cette règle est appliquée afin de sélectionner une transition non encore traitée pour générer la règle de réécriture équivalente en Maude.

À la fin d'exécution des règles, l'action finale de notre grammaire détruit les attributs temporaires des éléments du modèle et ferme le fichier de sortie.

Pour générer la description Maude à partir de l'éditeur des ECATNets, nous avons assigné l'exécution de cette grammaire de graphes à un bouton étiqueté " *Generate Maude Description*".

4.5 Etapes du Simulateur des ECATNets: Problème de routage

Nous considérons un exemple dans [Bettaz93] à propos d'un réseau de communication qui relie des messages émetteurs aux messages récepteurs. D'abord, nous présentons le modèle ECATNet décrivant cet exemple dans l'outil généré. Puis, nous rapportons le résultat de l'application de notre grammaire de graphes de génération de code Maude sur cet exemple. Enfin, nous examinons l'application du système Maude sur la spécification générée pour simuler le modèle ECATNet.

5.1 Présentation de l'Exemple

Cet exemple concerne un réseau de communication qui relie des messages émetteurs M aux messages récepteurs N. Chaque émetteur (respectivement, récepteur) est connecté à un port de réseau. Chaque groupe d'émetteurs (ou/et de récepteurs) envoie des messages en parallèle. La Figure 4.5 montre la fenêtre principale sur laquelle l'exemple est créé avec les marquages des places et les inscriptions des arcs.

Les places, les transitions et les inscriptions d'arc sont comme suit :

Places: R1, R2, S1,..,Sm, Queue1,.., Queuen, Adr1,.., Adrn

Transitions: From-S1,.., From-S1, To-R1,.., To-Rn, Check-Adr1,.., Check-Adrn

Inscriptions des arcs: Nous utilisons la définition en termes de spécification algébrique de la file d'attente (queue) : q est une variable de type queue. front(q) est une fonction qui retourne le message m qui est en tête de la file q. addq(m, q) est une fonction qui ajoute le message à la fin de q. remove(q) est une fonction qui retourne le reste de la file q après avoir supprimé le premier message (en tête de q).

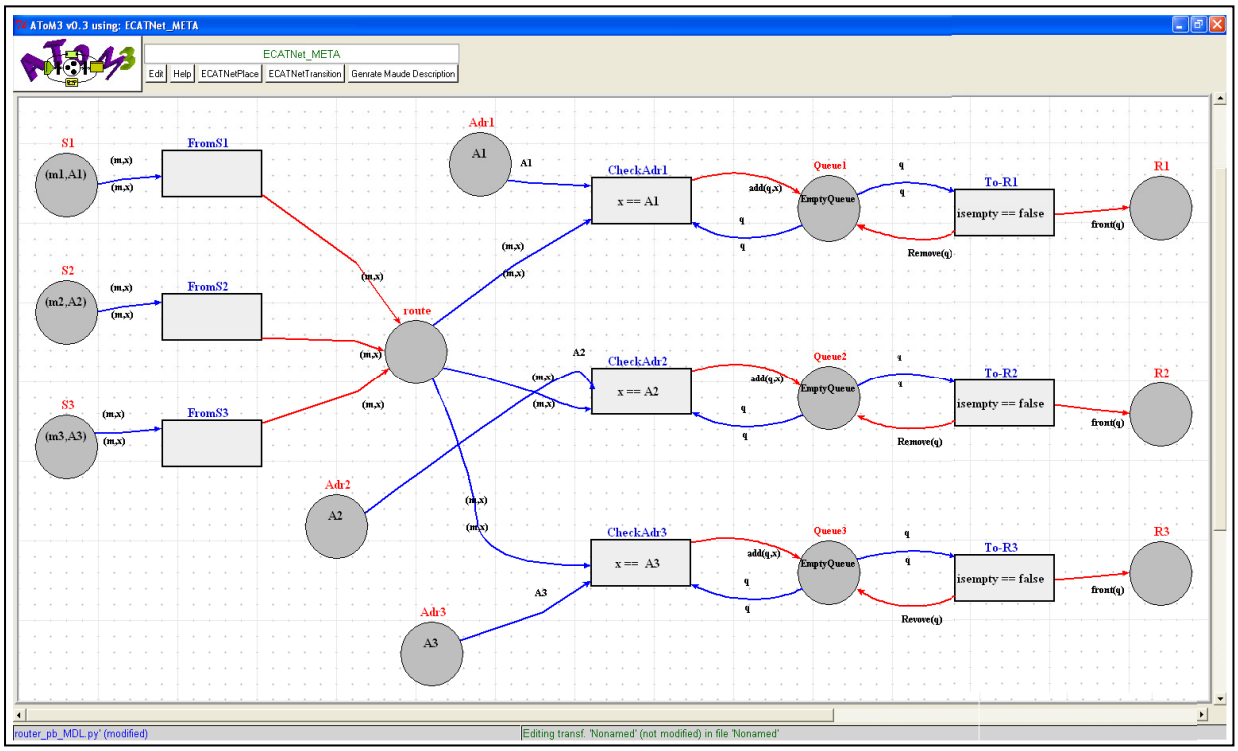


Figure 4.5: Fenêtre principale avec le modèle de l'exemple

5.2 Translation du Modèle ECATNet à une description Maude équivalente

Cette étape doit avoir une représentation graphique d'un modèle ECATNet comme entrée. L'étape consiste à translater cette représentation graphique à une description équivalente dans Maude. Cette représentation contient d'une part la structure de l'ECATNet, et d'autre part, l'état initial de cet ECATNet. Après l'exécution de notre grammaire, nous avons obtenu le fichier "*router_pb-system.maude*" de la Figure 4.6. Le fichier généré dans cette étape se compose de deux éléments: un code équivalent dans Maude pour la structure de l'ECATNet et l'état initial dans la syntaxe Maude.

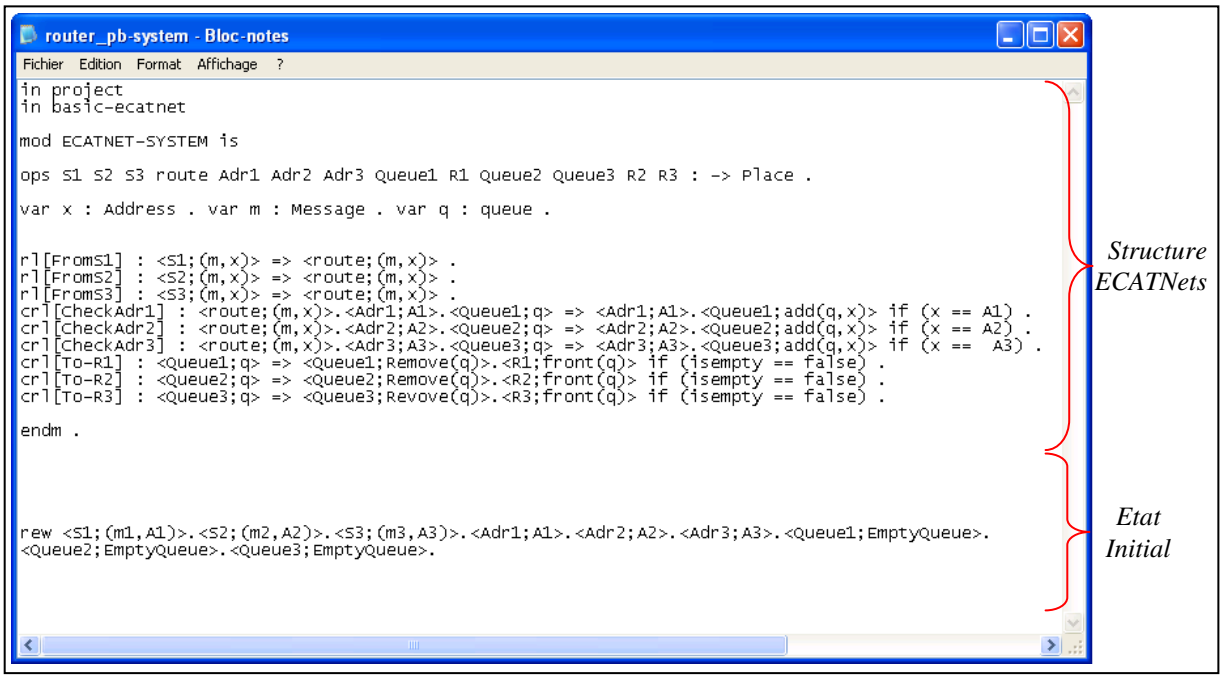


Figure 4.6: Spécification Maude de l'exemple

5.3 Simulation

Le résultat de l'étape précédente est l'entrée de celle-ci. Pour effectuer la simulation, nous avons besoin de la part de l'utilisateur l'état initial de l'ECATNet. La simulation consiste à transformer ce marquage initial à un autre en faisant une ou plusieurs actions de réécriture. Par conséquent, en plus de l'état initial, l'utilisateur peut donner au simulateur le nombre des étapes de réécriture s'il veut contrôler les états intermédiaires. Si ce nombre n'est pas donné, le simulateur continue le travail jusqu'à un état final. Dans la Figure 4.7, nous demandons à l'application de faire la simulation sur l'exemple précédent du marquage initial sans indiquer le nombre des étapes de réécriture. L'état final (résultat) de la simulation est donné de la même manière que l'état initial. Notons que le cas de l'infinité de calcul est possible.

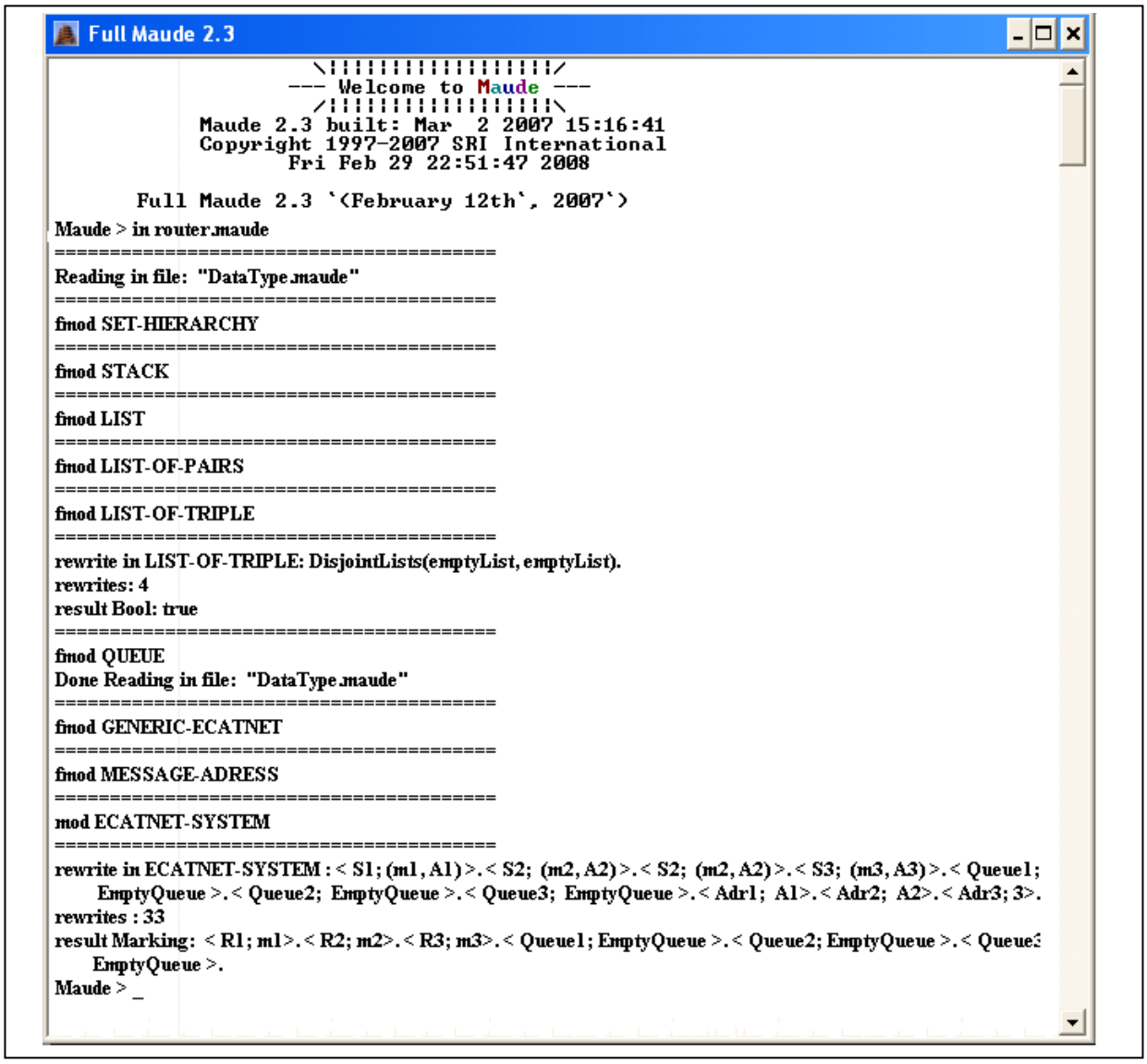


Figure 4.7: Simulation de l'exemple ECATNets sous le système Maude

4.6 Conclusion

Dans ce chapitre, nous avons décrit notre support outillé de manipulation et de simulation pour les ECATNets basé sur la méta-modélisation et les grammaires de graphes. L'éditeur permet à un utilisateur de dessiner un ECATNet graphiquement et de générer sa description équivalente dans Maude. Ce dernier est utilisé pour la simulation de la spécification. Nous avons montré à travers un exemple les étapes de simulation dans notre outil. Notre objectif à travers cet outil c'est de faciliter l'utilisation conjointe des modèles graphiques des ECATNets et les représentations textuelles dans le langage Maude. Les ECATNets disposent d'un côté graphique facilitant le développement d'un système au cours de sa spécification. D'autre part la version textuelle du langage Maude permet le développement des outils pour la simulation et l'analyse des ECATNets.

Chapitre 05 :

Une Plate-Forme Formelle pour la Spécification et l'Analyse des G-Nets

5.1 Introduction

G-Nets [Deng93] sont une catégorie des réseaux de Petri de haute niveau définis pour supporter la spécification et la conception modulaire des systèmes d'information distribués. Les G-Nets intègrent la théorie des réseaux de Petri avec l'approche orientée objet du génie logiciel. La motivation de cette intégration est de bénéficier des avantages qu'apporte le traitement formel des réseaux de Petri dans l'approche modulaire orientée objet pour la spécification et le prototypage de systèmes logiciels complexes.

Les notations graphiques proposées par les G-Nets fournissent un moyen intuitif pour spécifier les systèmes en termes de modules qui favorisent l'abstraction, l'encapsulation et le couplage faible entre ces modules. D'autre part, le comportement de chaque module (*G-Net*) est décrit dans les bases des réseaux de Petri.

Une fois la spécification G-Nets est achevée, elle peut être transformée manuellement [Deng93] en un modèle équivalent dans les réseaux de Petri à prédicats/transitions (*PrT-Nets*) [Genrich81]. Cette classe de réseaux de Petri de haut niveau aussi, permet de spécifier les communications inter-processus avec passage de valeurs. L'objectif de cette transformation est d'appliquer les techniques d'analyse formelle des PrT-Nets sur les spécifications G-Nets. L'outil d'analyse PROD [PROD] est l'un des analyseurs les plus utilisés pour les PrT-Nets.

PROD [PROD] est un outil textuel d'analyse d'atteignabilité des PrT-Nets développé par le laboratoire DSL (*Digital Systems Laboratory*) à l'université de "*Helsinki University of Technology*", Finlande. Le langage de description des modèles PrT-Nets en PROD est le langage C étendu par des directives permettant de décrire le réseau de Petri. Cette description est compilée par l'analyseur PROD afin d'obtenir un programme exécutable qui génère le graphe d'atteignabilité du modèle PrT-Net. A notre connaissance, aucun outil graphique pour la génération de telle description n'existe à ce jour.

Dans ce chapitre, nous présentons une plate-forme formelle et un outil graphique pour la spécification et l'analyse des systèmes logiciels complexes en utilisant les G-Nets. Notre approche est basée sur la méta-modélisation et les grammaires de graphes. L'outil proposé permet d'exploiter les différents langages et techniques formels dans un cadre unifié dont le but est de cumuler leurs avantages. Notre plate-forme permet à l'utilisateur d'élaborer un modèle G-Nets et de le transformer automatiquement vers un ensemble de modèles PrT-Nets équivalents. Afin d'effectuer l'analyse en utilisant l'analyseur PROD, notre plate-forme permet aussi de générer

automatiquement pour chaque modèle PrT-Nets sa description équivalente selon le langage d'entrée de l'analyseur PROD.

Pour ce faire, nous avons défini un méta-modèle pour le formalisme G-Nets et un autre pour le formalisme PrT-Nets. Ensuite, nous avons utilisé l'outil de méta-modélisation AToM^3 pour générer automatiquement un outil de modélisation visuelle pour chaque formalisme en fonction de son méta-modèle. Nous avons également proposé deux grammaires de graphes. La première effectue la transformation des modèles G-Nets vers des modèles PrT-Nets équivalents selon la technique de transformation proposée dans [Deng93]. La seconde génère pour les modèles PrT-Nets, obtenus par la première grammaire, leurs descriptions équivalentes dans le langage de l'analyseur PROD.

Le reste de ce chapitre est organisé comme suit: La section 2 présente les G-Nets ainsi que la technique de transformation vers les PrT-Nets. Les méta-modèles des G-Nets et des PrT-Nets ainsi que leurs éditeurs graphiques sont décrits dans la section 3. La section 4 présente la plate-forme formelle ainsi que les grammaires de graphes. La section 5 présente un exemple d'utilisation de notre approche. Finalement, la section 6 conclut le chapitre.

5.2 Les G-Nets

Le principe de base du génie logiciel est le découpage d'une application en plusieurs parties relativement indépendantes appelées modules, où chaque module masque leur détails internes de traitement et communique avec les autres modules à travers des interfaces bien définies [Fairley85]. Les G-Nets sont une classe de réseaux de Petri de haut niveau qui adoptent ce principe de modularité ([Deng93] et [Fairley85]). Une spécification basée sur les G-Nets (appelée spécification G-Nets) est constituée d'un certain nombre de G-Nets. Chaque G-Net représente un module indépendant. Une spécification G-Nets est organisée en fonction de la structure du système étudié.

Un G-Net se compose de deux parties: une place spéciale appelée *Generic Switch Place* (*GSP*) et une structure interne (*IS*: *Internal Structure*). La place GSP représente l'abstraction du module, c'est-à-dire sa signature. Elle sert d'interface entre le G-Net et les autres G-Nets de la spécification à travers des méthodes et des attributs (s'il y a lieu). Une méthode définit dans la structure interne du G-Net les paramètres d'entrées, la liste des places initiales avec leurs marquages et la liste des places finales qui représentent la fin d'exécution de la méthode. La structure interne (IS) contient les détails du comportement du G-Net décrits par les notations des réseaux de Petri augmentées par des directives qui permettent d'indiquer la communication entre

les G-Nets. Elle consiste en trois types de places (place ordinaire, place finale et place ISP), des transitions et des jetons. La place ISP (*Instantiated Switching Place*) est utilisée pour représenter la communication inter-G-Nets. L'inscription **isp(*G',mtd$_i$*)** dans la place ISP indique l'invocation du G-Nets "***G'*** "avec la méthode " ***mtd$_i$*** ". La Figure 5.1 montre les notations utilisées pour représenter les G-Nets.

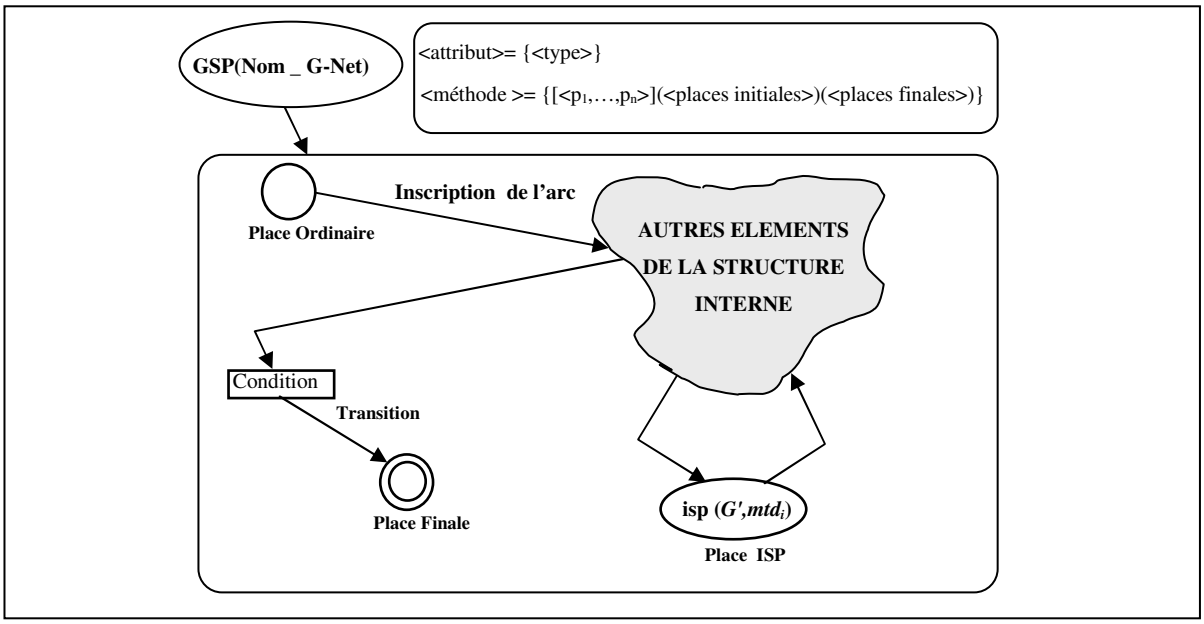


Figure 5.1 : Notations utilisées pour représenter un G-Net

Pour analyser une spécification G-Nets, les auteurs de [Deng93] ont proposé une technique de transformation qui permet de convertir une spécification G-Nets en un ensemble de modèles PrT-Nets. Chaque Modèle PrT-Net correspond à un couple (*G-Net, méthode*) dans la spécification G-Nets.

Les PrT-Nets sont une catégorie des réseaux de Petri de haut niveau permettant de spécifier des communications inter-systèmes avec passage de valeurs. Les jetons des places sont munis d'opérations et de relations et sont structurés par des n-uplets de valeurs. Dans les PrT-Nets les transitions sont remplacées par des règles dans une logique de premier ordre [Genrich81]. Les transitions sont sensibilisées par une famille d'événements au lieu d'un seul événement comme les réseaux de Petri standards et les arcs sont dénotés par une variable au lieu d'une valeur.

L'idée de base de la technique de transformation d'une spécification G-Nets en un modèle PrT-Net équivalant est schématisée dans la Figure 5.2. Pour plus de détails de la transformation voir [Deng93].

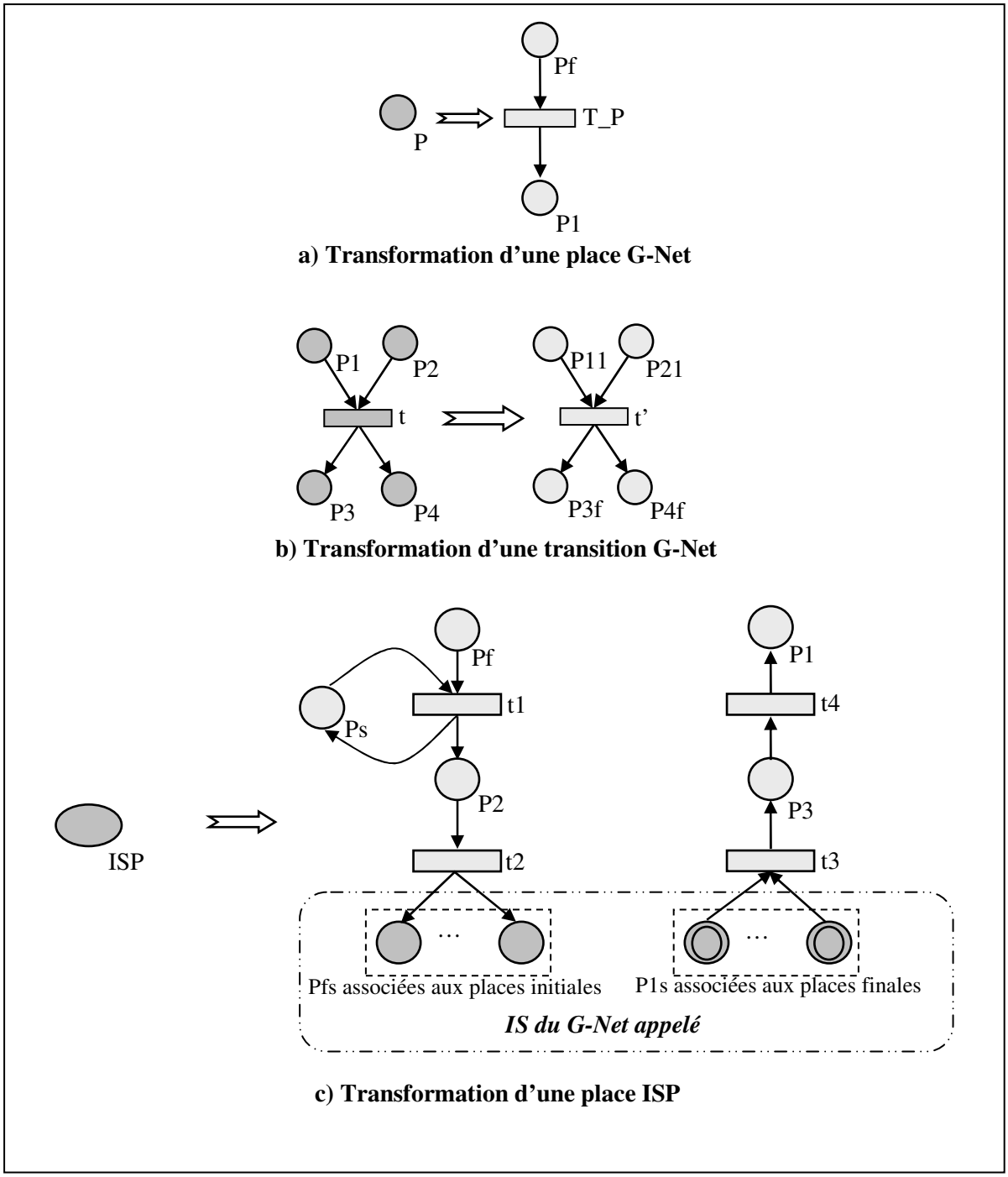


Figure 5.2: Technique de Transformation [Deng93]

Notons que cette approche de transformation se fait manuellement. Dans le cadre de ce chapitre, nous proposons une approche basée sur la transformation de graphes et grammaire de graphes qui permet d'automatiser cette approche.

5.3 Méta-Modélisation des G-Nets et des PrT-Nets

Pour construire un éditeur graphique pour les modèles en AToM^3, nous devons définir un méta-modèle pour eux. Le méta-formalisme utilisé dans ce travail est le modèle de diagrammes de classes d'UML et les contraintes sont exprimées en code Python.

Pour méta-modéliser les G-Nets dans l'outil AToM^3, nous avons proposé quatre classes reliées par cinq associations comme le montre la Figure 5.3 [Kerkouche09b]. Nous avons aussi associé à chaque élément du méta-modèle la représentation graphique qui lui convient selon la Figure 5.1.

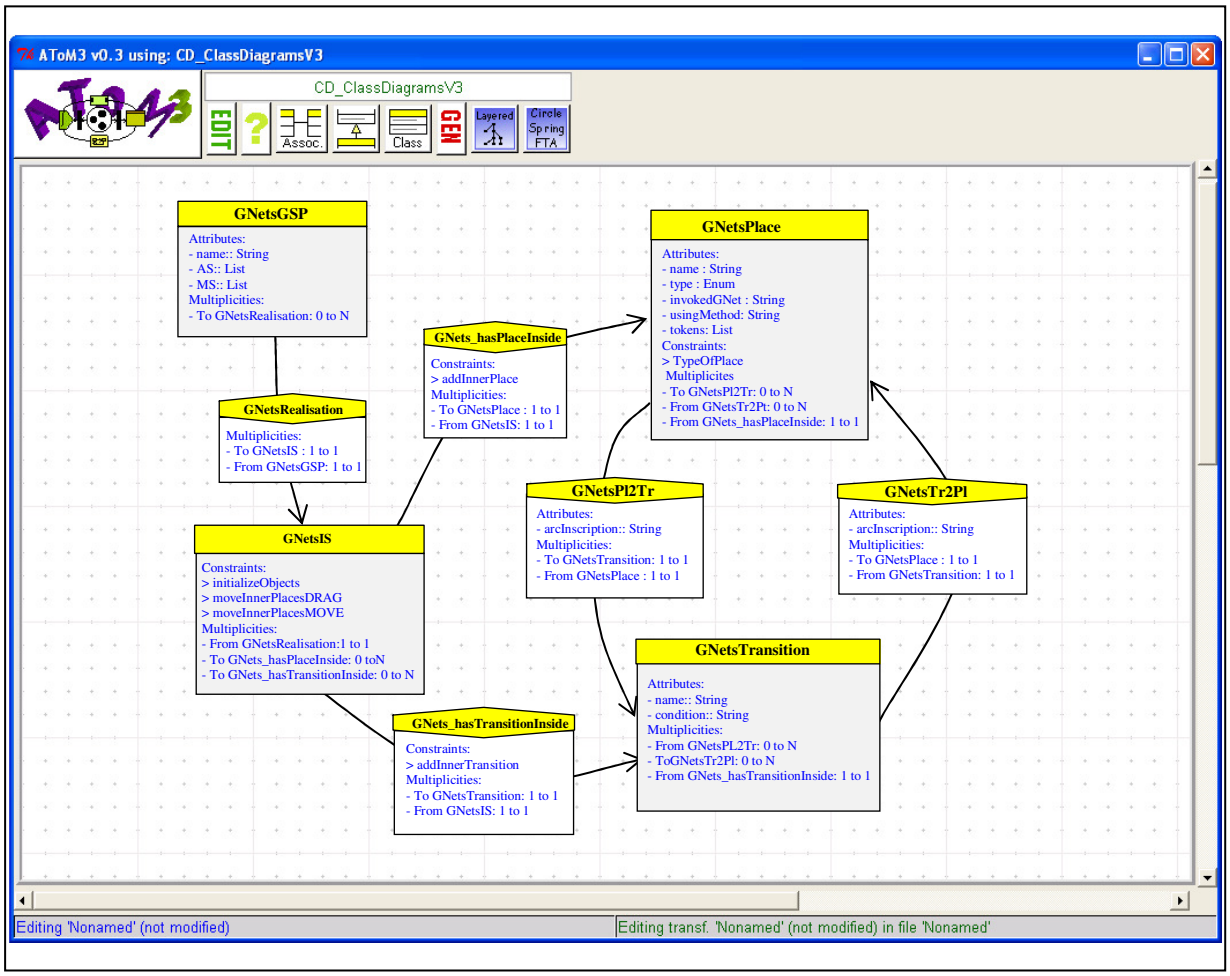


Figure 5.3: Méta-modèle des G-Nets

Classe "GNetsGSP": elle représente la place spéciale *Generic Switch Place* (*GSP*) du G-Net. Elle possède un attribut clé *Name* et deux listes. La liste *AS* contient les variables du G-Net, et la liste *MS* contient les méthodes du G-Net.

Classe "GNetsIS": elle représente la structure interne (*IS*) du G-Net. Pour chaque instance de cette classe, il y a trois contraintes qui permettent de garder ses éléments (les places et les transitions) à l'intérieur de son apparence graphique.

Classe "GNetsPlace": elle représente les places du G-Net. Elle possède cinq attributs: *name*, *type*, *tokens*, *invokedGNet* et *usingMethode.* L'attribut *type* indique le type de la place: place ordinaire, place finale ou place ISP. Les attributs *invokedGNet* et *usingMethode* sont utilisés seulement dans le cas où la place est de type ISP. La contrainte "*TypeOfPlace"* donne l'apparence graphique appropriée à chaque type de place.

Classe "GNetsTransition": elle représente les transitions du G-Net. Elle possède un attribut *name* et un attribut *condition*.

Association "GNetsRealisation": elle relie la classe "*GNetsGSP*" avec la classe "*GNetsIS*". Elle est visualisée par une flèche noire partant de la place *GSP* du G-Net vers son *IS*.

Association "GNetsPl2Tr": elle relie la classe "G-NetPlace" avec la classe "G-NetTransition". Elle est visualisée par une flèche portant l'attribut *arcInscription*.

Association "GNetsTr2Pl": elle relie la classe "G-NetTransition" avec la classe "G-NetPlace". Elle est visualisée par une flèche portant l'attribut *arcInscription*.

Associations "GNets_hasPlaceInside" et ***"GNets_hasTransitionInside":*** ces deux association sont utilisées pour représenter l'appartenance de places et de transitions de la structure interne (*IS)* d'un G-Net. Il est à noter que ces deux associations ne sont pas représentées graphiquement.

Grâce au méta-modèle des G-Nets, nous avons généré un éditeur graphique pour ces derniers en utilisant AToM^3. L'outil généré va nous permettre d'éditer et de manipuler les déférents modèles des G-Nets (voir la Figure 5.4).

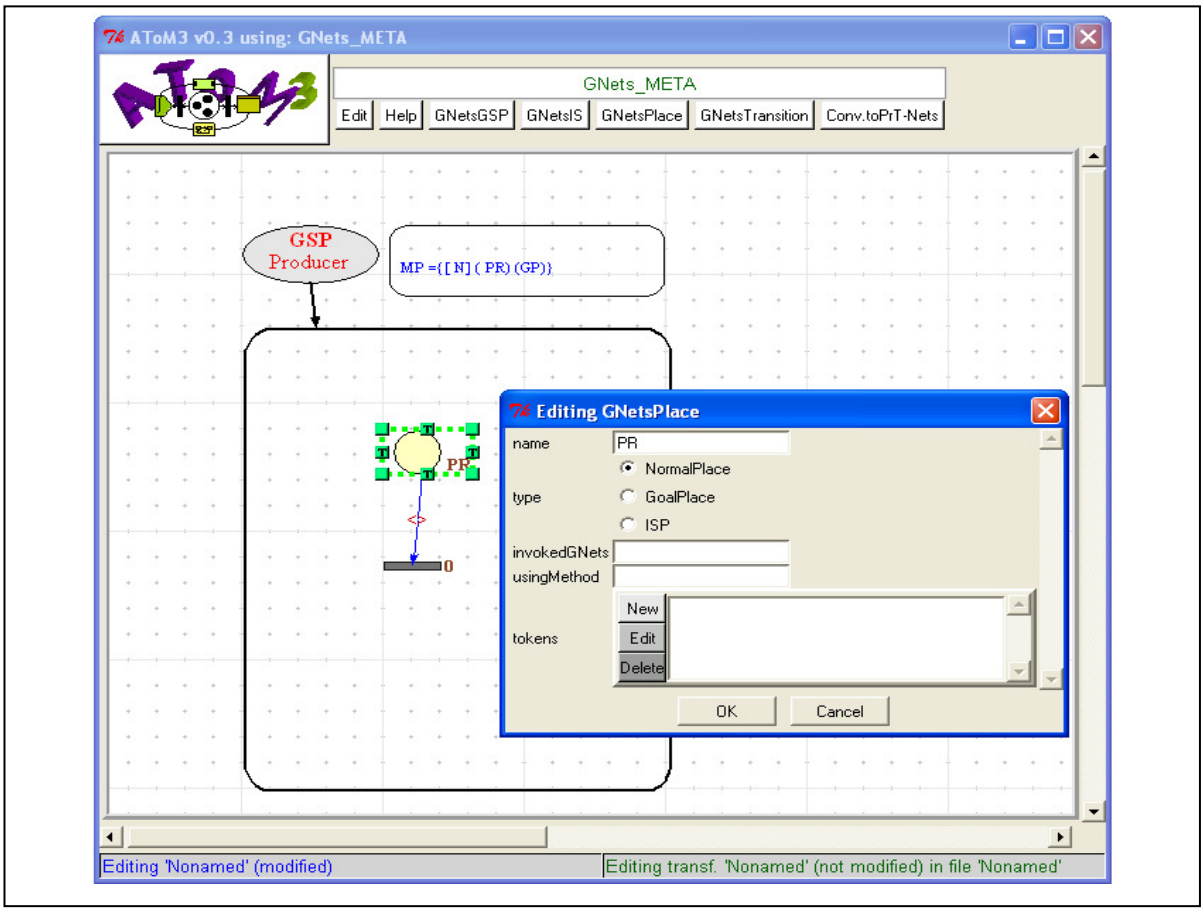


Figure 5.4: Outil de modélisation des G-Nets

Les PrT-Nets consistent en des places, des transitions, d'arcs des places vers des transitions et d'arcs des transitions vers des places. Nous avons proposé pour les méta-modéliser deux classes et deux associations [Kerkouche09b] comme l'illustre la Figure 5.5.

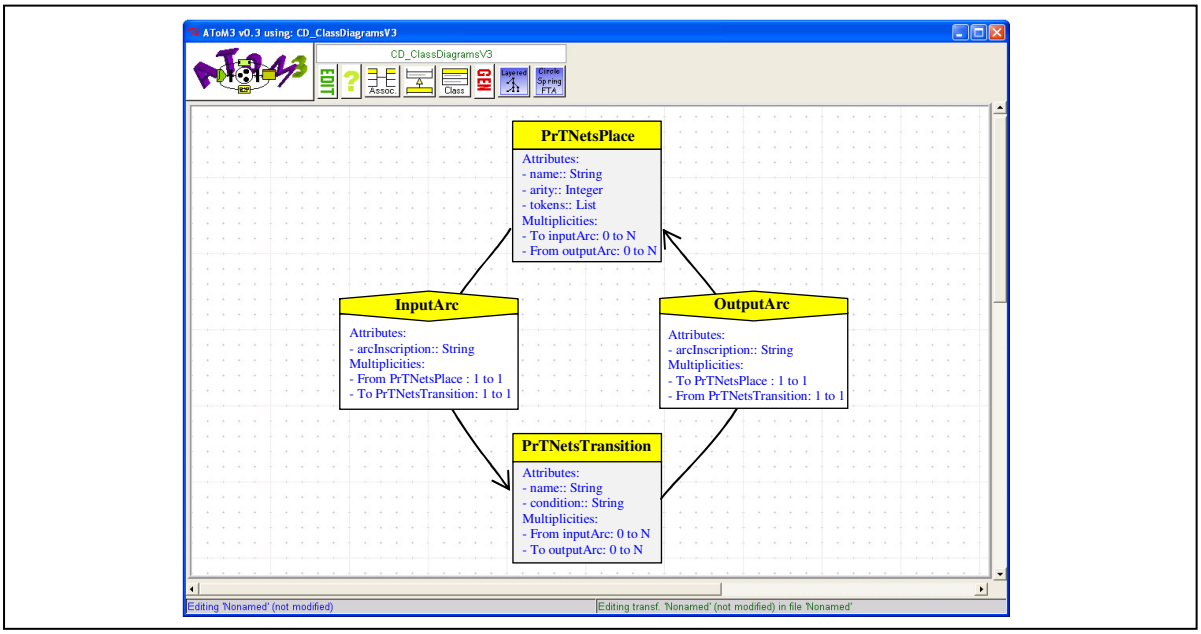


Figure 5.5: Méta-modèle des PrT-Nets

Etant donné notre Méta-modèle des PrT-Nets décrits dans la Fgure 5.5, nous avons généré un éditeur visuel de modélisation pour les PrT-Nets tel qu'il est présenté dans la Figure 5.6

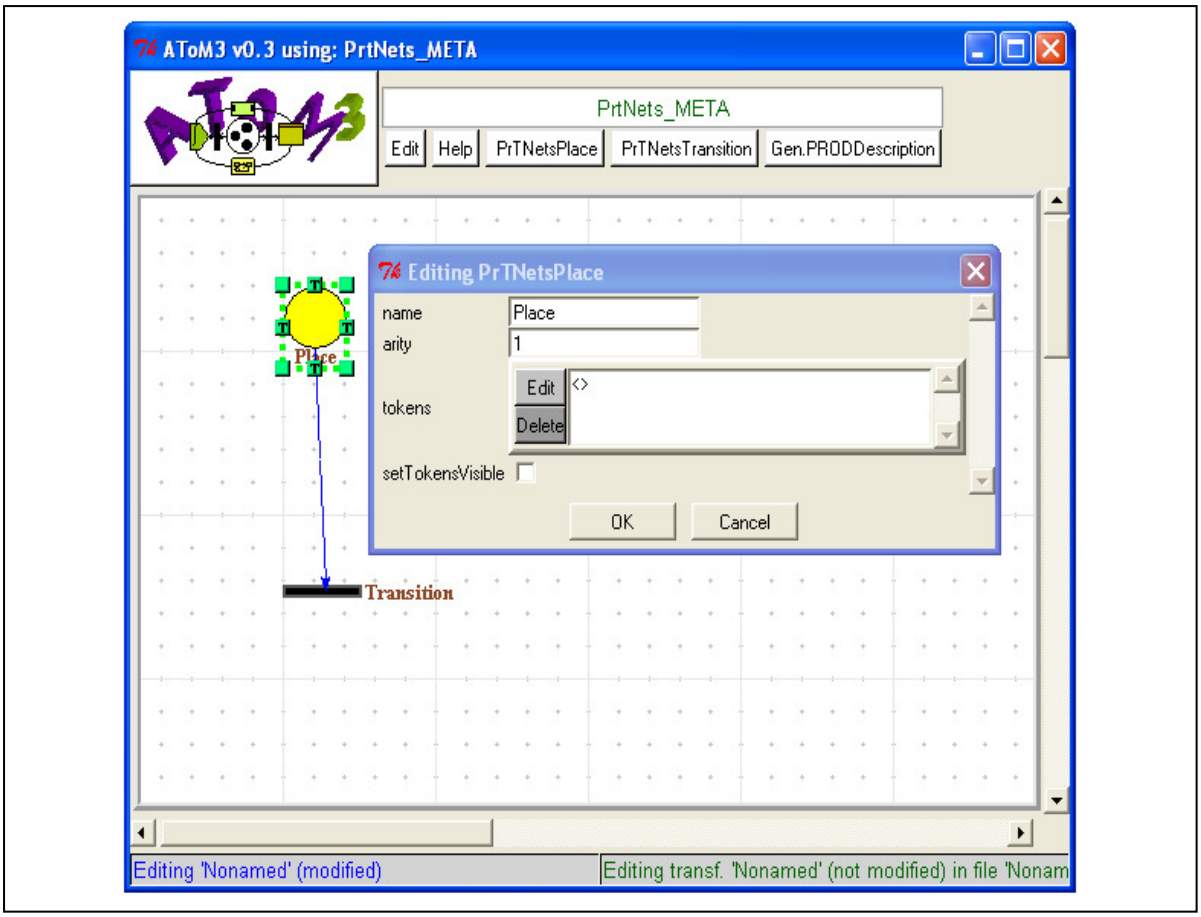


Figure 5.6: Outil de modélisation des PrT-Nets

Puisque AToM^3 est un outil visuel pour la modélisation multi-formalisme, il est possible d'utiliser les deux éditeurs simultanément. Plus précisément, les deux barres d'outils des deux formalismes générées peuvent être affichées dans l'interface utilisateur de l'outil AToM^3. Ceci permet de manipuler les deux formalismes en même temps et dans le même cadre (voir la Figure 5.11).

5.4 Plate-forme Formelle pour les G-Nets

Dans cette section nous présentons notre plate-forme formelle pour la spécification et l'analyse des G-Nets.

La plate-forme de modélisation que nous avons obtenu dans la section précédente par la méta-modélisation permet uniquement de créer des modèles et de vérifier qu'ils sont syntaxiquement corrects. Dans ce qui suit, nous améliorons ses capacités au moyen des grammaires de graphes. Plus précisément, nous allons définir deux grammaires de graphes. La première

grammaire transforme un couple (*G-Net, Méthode*) dans la spécification G-Nets en un modèle PrT-Nets équivalent. La deuxième grammaire génère la description PROD équivalente au modèle résultat de la première grammaire. Ensuite, l'analyseur PROD peut être utilisé pour analyser le couple (*G-Net, Méthode*). Le schéma global de la plate-forme est illustré dans la Figure 5.7.

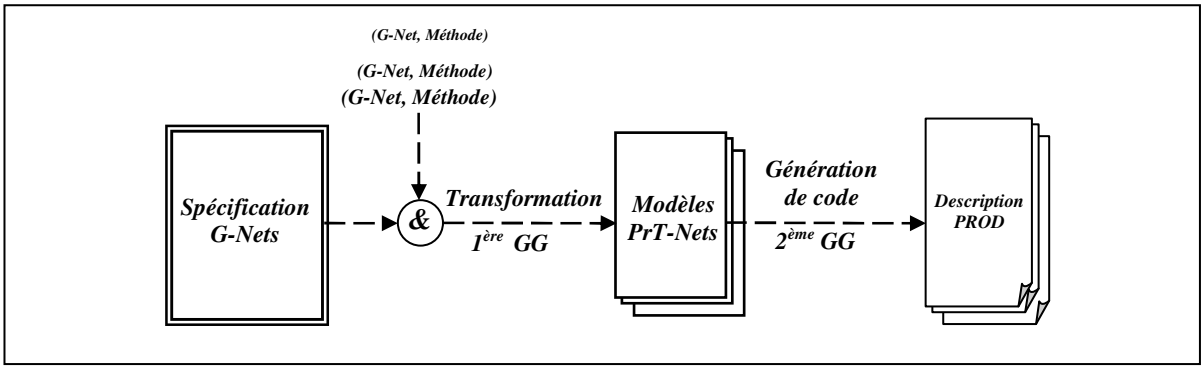


Figure 5.7: La Plate-forme Formelle proposée pour les G-Nets

Dans cette section, nous utilisons AToM^3 pour définir les deux grammaires de graphes pour notre plateforme.

5.4.1 1ère GG: Transformation G-Nets vers PrT-Nets

Afin de produire les modèles PrT-Nets équivalents à une spécification G-Nets, nous avons proposé une grammaire de graphes avec 21 règles [Kerkouche09b] qui seront exécutées dans un ordre ascendant de leurs priorités. Durant l'exécution des règles de la grammaire, les graphes intermédiaires contiennent les éléments des deux formalismes: les G-Nets et les PrT-Nets, mais à la fin de l'exécution le graphe obtenu est un modèle PrT-Nets.

L'action initiale de la 1ère Grammaire de graphes consiste à créer des variables globales nécessaires dans le processus de transformation. Les variables globales sont des variables qui sont accessibles pour toutes les règles de la grammaire:

– La Liste *List_Visited_GNets_Names*: elle est utilisée pour enregistrer les noms des G-Nets qui sont déjà transformés. Chaque G-Net dans *List_Visited_GNets_Names* est appelé en utilisant une de ses méthodes.

– La liste *List_Visited_Mtd_Names:* elle est utilisée pour enregistrer les noms des méthodes des G-Net transformés. Chaque méthode correspond à un G-Net qui se trouve dans le même rang dans la liste *List_Visited_GNets_Names.*

Il est important de noter qu'un G-Net peut être appelé par toutes ses méthodes. Ces deux listes sont utilisées pour éviter de transformer plusieurs fois le couple (*G-Net, Méthode*).

– La liste *List_Init_Places:* cette liste sert à contenir les places initiales de la méthode qui est en cours de transformation.

– La liste *List_Goal_Places:* cette liste sert à contenir les places finales de la méthode qui est en cours de transformation.

– Les variables *Current_GNets_Name* et *Current_Mtd_Name*: sont utilisées pour spécifier le couple (*G-Net, Méthodes*) qui est en cours de transformation.

Avant de commencer la transformation, l'action initiale invite l'utilisateur à sélectionner un G-Net et une de ses méthodes dans la spécification G-Nets. Le nom du G-Net sélectionné et le nom de la méthode sont affectés aux variables *Current_GNets_Name* et *Current_Mtd_Name* respectivement. Pour transformer le couple (*Current_GNets_Name*, *Current_Mtd_Name*) en un modèle PrT-Nets équivalent nous avons proposé les règles présentées dans les Figures 5.8 et 5.9. Ces règles sont décrites comme suit:

Règle N° 1, 2, 3, 4, 5 and 6: TrnsfArc_... (Priorité resp. 1, 2, 3, 4, 5 and 6). Ces règles sont appliquées pour remplacer les arcs (sortant d'une place ordinaire ou finale vers une transition, ou inversement) dans le modèle G-Nets par leurs arcs équivalents dans le modèle PrT-Nets. Pour appliquer ces règles, au moins une des extrémités de l'arc (la place ou la transition) dans le modèle G-Nets est transformée. L'extrémité qui n'est pas encore transformée sera aussi traitée par ces règles. Par exemple, la règle N°2 transforme à la fois l'arc et la transition (voir la Figure 5.8). Plus précisément, cette règle consiste à créer une transition PrT-Nets et de l'attacher à la transition G-Nets. Ensuite de générer un arc dans le modèle PrT-Nets sortant du segment équivalent à la place G-Nets (attaché à la place G-Nets) vers la transition équivalente (attaché à la transition G-Nets).

Règle N° 7, 8, 9 and 10: TrnsfArc_... (Priorité resp. 7, 8, 9 and 10). Ces règles sont similaires aux règles précédentes, mais elles sont appliquées pour les places ISP du modèle G-Nets.

Règle N° 11, 12 and 13: TrnsfArc_... (Priorité resp. 11, 12 and 13). Puisque plusieurs places ISP peuvent appeler le même couple (*G-Net, Méthode*) dans la spécification G-Nets, ces règles sont appliquées pour s'assurer que le couple (*G-Net, Méthode*) est transformé une seule fois. Le couple (*G-Net, Méthode*) est transformé si le nom du G-Net et le nom de la méthode sont dans le même rang dans les deux listes *List_Visited_GNets_Namesb et List_Visited_Mtd_Names* respectivement. Par exemple, la règle N°11 est appliquée lorsque le couple (*G-Net, Méthode*) invoqué par la place ISP de sortie de l'arc est appelé par une autre place ISP traitée. Cette règle localise la place ISP traitée qui contient le même couple (*G-Net, Méthode*) afin de générer un arc dans le modèle PrT-Nets sortant de la transition (attaché à la transition G-Nets) vers la place ISP traitée.

Règle N°14: Gen_StartOfInvocation (Priorité14). Cette règle est appliquée pour relier l'interface d'invocation associée à la place ISP (créée par la règle N°18) avec le segment PrT-Nets attaché à la place initiale de la méthode appelée par le couple (*G-Nes, Méthode*).

Règle N°15: Trnsf_InitialPlace (Priorité15). Cette règle est appliquée pour localiser le G-Net à transformer, c'est-à-dire son nom est dans la variable *Current_GNets_Name*. Puis, elle associe à chaque place initiale du G-Net le segment PrT-Nets équivalent. Nous notons que les noms de toutes les places initiales sont dans la liste *List_Init_Place*.

Règle N°16: Gen_EndOfInvocation (Priorité16). Cette règle est appliquée pour relier le segment PrT-Nets attaché à la place finale de la méthode appelée à l'interface d'invocation associée à la place ISP (créée par la règle N°18).

Règle N°17: GetInitialPlaces (Priorité17). Cette règle est appliquée pour localiser le G-Net, dont le nom est égal à la valeur de la variable *Current_GNets_Name*, afin obtenir la liste des places initiales *List_Init_Place* de la méthode appelée (*Current_Mtd_Name*).

Règle N°18: GenInterface_ISP2GNets (Priorité18). Le rôle de cette règle est de générer l'interface d'invocation dans le modèle PrT-Nets qui imite le mécanisme d'appel entre les G-Nets. Plus précisément, cette règle génère un segment dans le modèle PrT-Nets qui relie le segment attaché à la place ISP du G-Net appelant aux segments attachés aux places initiales et finales du couple (*G-Net, Méthode*) appelé. En plus, cette règle se termine par une action qui affecte le nom du G-Net appelé par la place ISP à la variable *Current_GNets_Name* et le nom de la méthode utilisée à la variable *Current_Mtd_Name*. Avec ces nouvelles valeurs, la grammaire de graphes effectue la transformation du nouveau couple (*G-Net, Méthode*). Ensuite elle relie les segments attachés à ses places initiales et finales lorsque les règles N°14 et N°16 sont appliquées respectivement.

Règle N°19: DeleteGNetsPlace (Priorité19). Cette règle est appliquée pour supprimer toutes les places dans le modèle G-Nets.

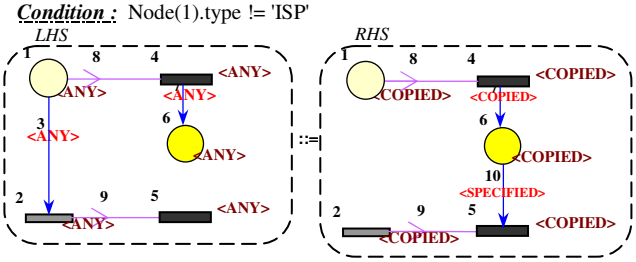
Règle N°20: DeleteGNetsTransition (Priorité20). Cette règle est appliquée pour supprimer toutes les transitions dans le modèle G-Nets.

Règle N°21: DeleteGNetsGSP (Priorité21). Cette règle est appliquée pour éliminer toutes les places spéciales GSP dans la spécification G-Nets.

A la fin d'exécution des règles, l'action finale détruit toutes variables globales utilisées.

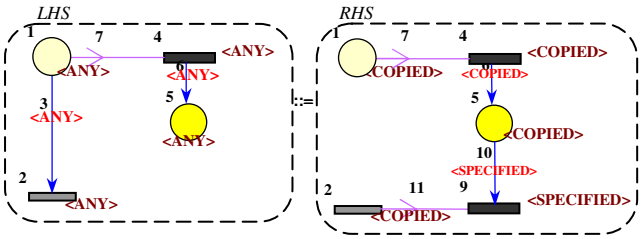
1.-TrnsfArc_VisitedPlace2VisitedTrans. Prio.1

Condition : Node(1).type != 'ISP'



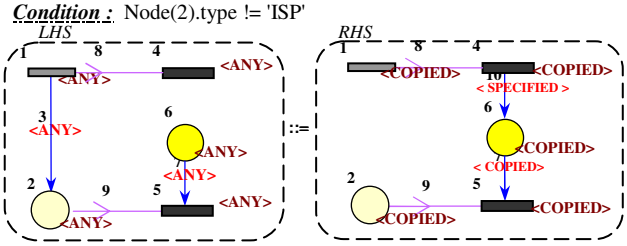
2.-TrnsfArc_VisitedPlace2Trans. Prio.2

Condition : Node(1).type != 'ISP'



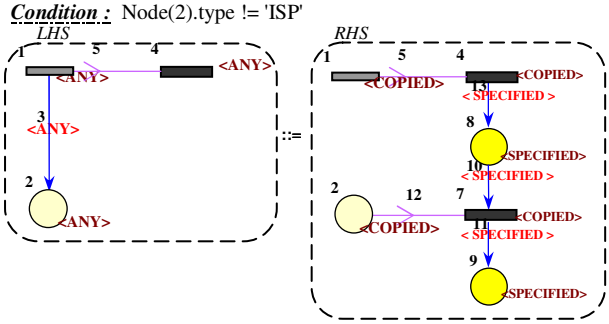
3.-TrnsfArc_VisitedTrans2VisitedPlace. Prio.3

Condition : Node(2).type != 'ISP'



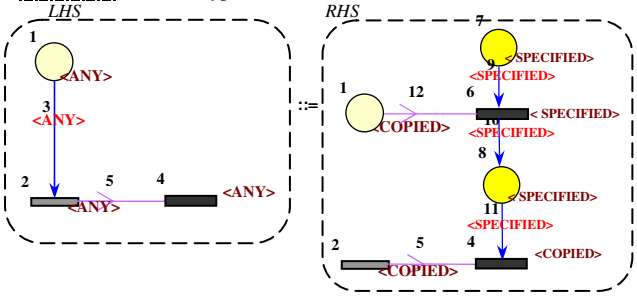
4.- TrnsfArc_VisitedTrans2Place.Prio.4

Condition : Node(2).type != 'ISP'



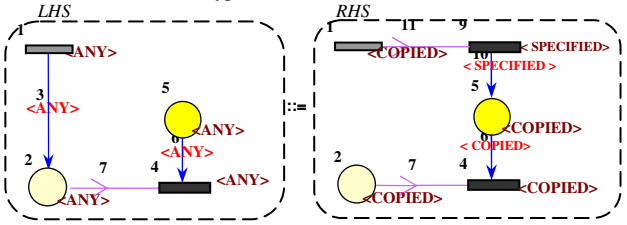
5.- TrnsfArc_ Place2VisitedTrans. Prio.5

Condition : Node(1).type != 'ISP'



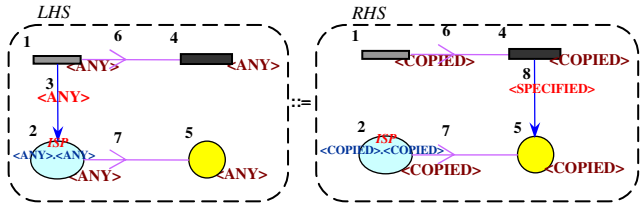
6.- TrnsfArc_ Trans2VisitedPlace. Prio.6

Condition : Node(2).type != 'ISP'



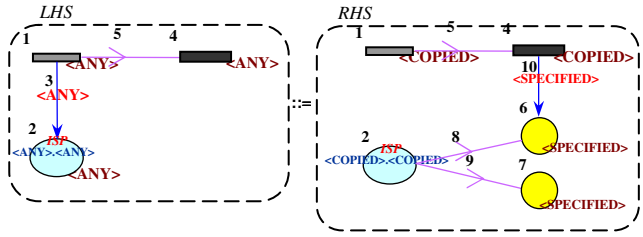
7.- TrnsfArc_ VisitedTrans2VisitedISP. Prio.7

Condition : Node(2).type = = 'ISP'



8.- TrnsfArc_ VisitedTrans2ISP. Prio.8

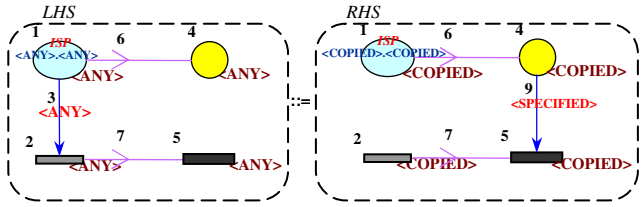
Condition : ((Node(2).type = = 'ISP') and
(Node(2).usingMethod not In *List_Visited_Mtd_Names)and*
(Node (2).invokedGNet not In *List_Visited_GNets_Names))*



Action : *List_Visited_GNets_Names*.append(Node(2).invokedGNet)
List_Visited_Mtd_Names.append(Node(2).usingMethod)

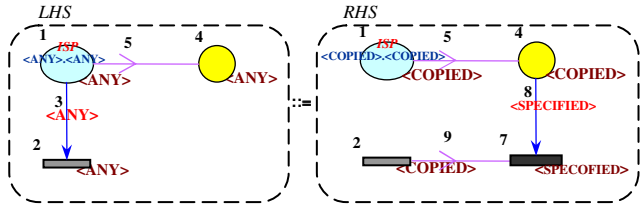
9.- TrnsfArc_VisitedISP2VisitedTrans. Prio.9

Condition : Node(1).type = = 'ISP'



10.- TrnsfArc_VisitedISP2Trans. Prio.10

Condition : Node(1).type = = 'ISP'



11.- TrnsfArc_VisitedTrans2CalledISP. Prio. 11

Condition : ((Node(2).type = = 'ISP') and (Node(4).type = = 'ISP')
and (Node(2).invokedGNets = = Node(4).invokedGNets)
and (Node(2). usingMethod = = Node(4). usingMethod))

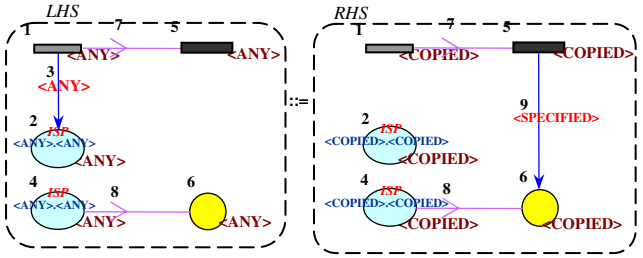
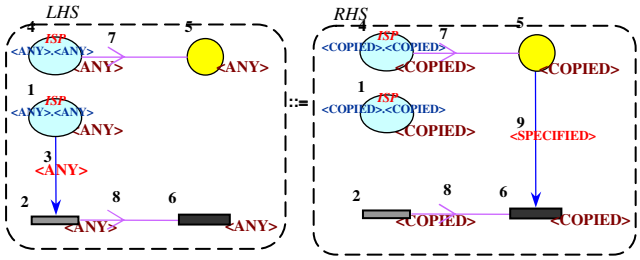


Figure 5.8: 1ère GG, Transformation des G-Nets vers des PrT-Nets, Règles 1-11

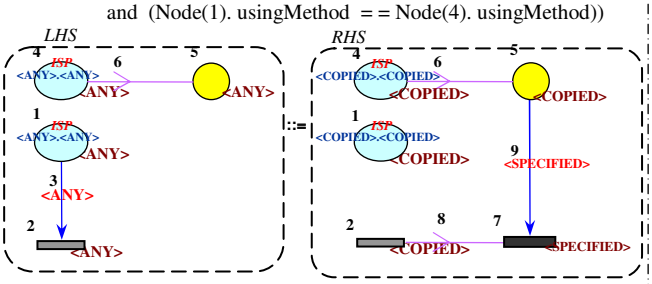
12.- TrnsfArc_CalledISP2VisitedTrans. Prio.12

Condition : ((Node(1).type = = 'ISP') and (Node(4).type = = 'ISP') and (Node(1).invokedGNets = = Node(4).invokedGNets) and (Node(1). usingMethod = = Node(4).usingMethod))



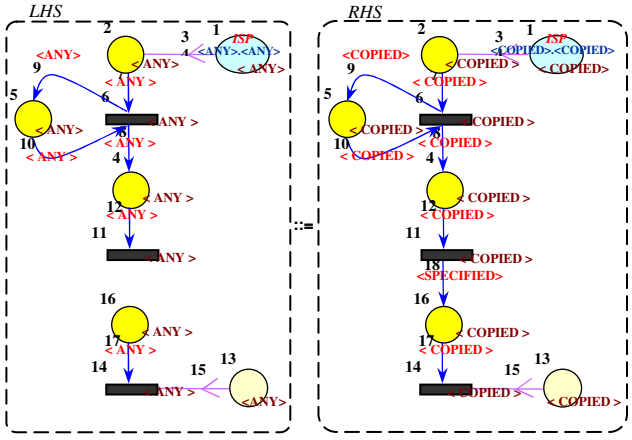
13. - TrnsfArc_CalledISP2Trans. Prio.13

Condition : ((Node(1).type = = 'ISP') and (Node(4).type = = 'ISP') and (Node(1).invokedGNets = = Node(4).invokedGNets) and (Node(1). usingMethod = = Node(4). usingMethod))



14.- Gen_StartOfInvocation. Prio.14

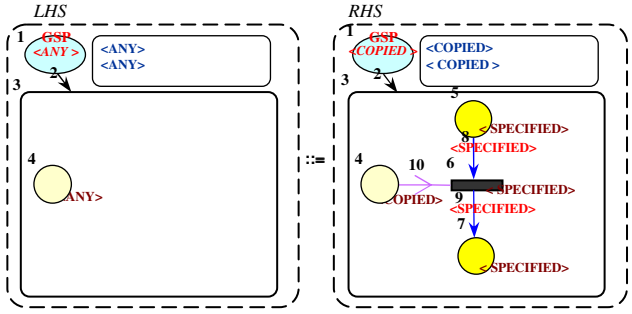
Condition : ((Node(1).type = = 'ISP') and (Node(13).type != 'ISP') and (Node(13).name in *List_Init_Places*))



Action : *List_Init_Places*.remove (Node(13).name)

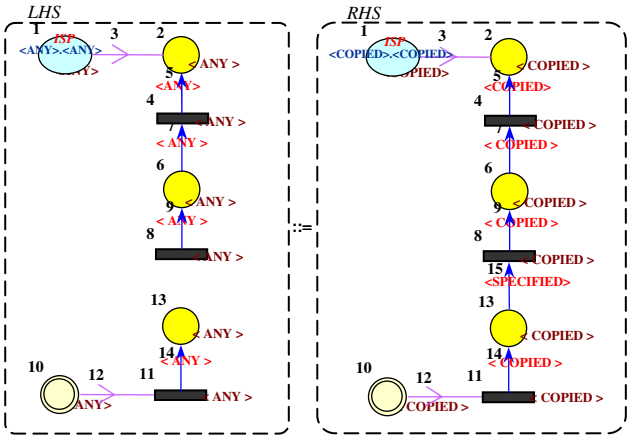
15.- Trnsf_InitialPlace. Prio.15

Condition : ((Node(1).name = = *Current_GNets_Name*) and (*Current_Mtd_Name* in MS) and (*List_Init_Places !=[]*) and (Node(4).name in *List_Init_Places*))



16.- Gen_EndOfInvocation. Prio.16

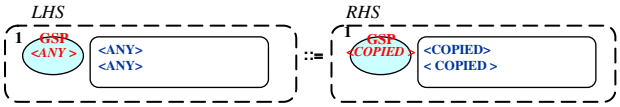
Condition : ((Node(1).type = = 'ISP') and (Node(10).type != 'Goal') and (Node(10).name in *List_Goal_Places*))



Action : *List_Goal_Places*.remove (Node(10).name)

17.- GetInitialPlaces. Prio.17

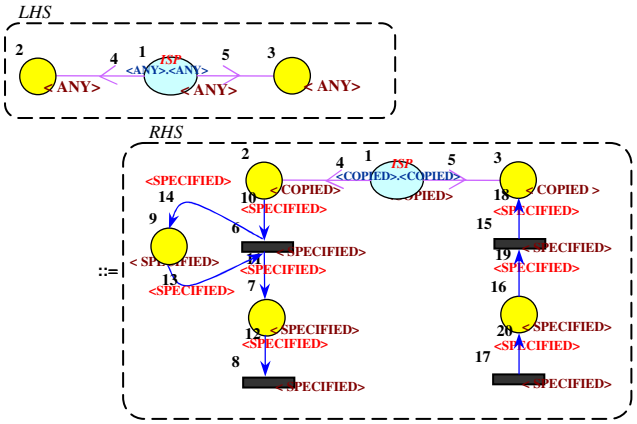
Condition : ((Node(1).name = = *Current_GNets_Name*) and (*Current_Mtd_Name* in MS))



Action : *List_Init_Places = Node(1).MS.Method..initPlaces*
List_Goal_Places = Node(1).MS.Method..goalPlaces

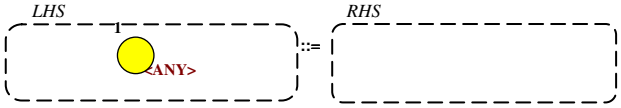
18.- GenInterface_ISP2InvokedGNet. Prio.18

Condition : ((Node(1).type = = 'ISP') and (Node(1).usingMethod not In *List_Visited_Mtd_Names))*

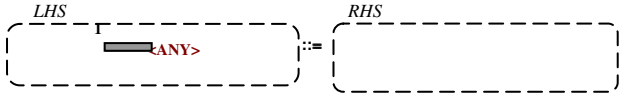


Action : *Current_GNets_Name = Node(1).invokedGNet*
Current_Mtd_Name = Node(1).usingMethod

19- DeleteGNetsPlace. Prio.19



20- DeleteGNetsTransition. Prio.20



21- DeleteGNetsGSP. Prio.21

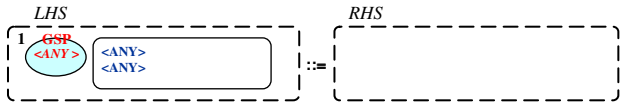


Figure 5.9: 1ère GG, Transformation des G-Nets vers des PrT-Nets, Règles 12-21

5.4.1 2ème GG: Génération de la description PROD

Afin de produire les spécifications PROD équivalentes aux modèles PrT-Nets, nous avons proposé une grammaire de graphes avec six règles qui seront exécutées dans un ordre ascendant de leurs priorités [Kerkouche09b] par l'outil AToM^3. L'application de cette grammaire à un modèle PrT-Net conduit à la génération d'un fichier qui contient sa description textuelle équivalente en PROD. Il est à noter que cette grammaire concerne la génération automatique de code, Par conséquent, aucune de ses règles ne va changer le modèle PrT-Net (la partie gauche est identique à la partie droite).

L'action initiale de notre grammaire de graphes consiste à créer un fichier qui contiendra la description équivalente en PROD et à décorer toutes les places et les transitions du modèle par des variables temporaires utilisées pour spécifier les conditions des règles. Pour chaque transition, deux attributs sont ajoutés: "*current*" and "*visited*". L'attribut "*current*" est utilisé pour identifier la transition dans le modèle pour laquelle le code est en cours de génération, et l'attribut "*visited*" est utilisé pour indiquer que le code pour la transition est généré. Dans les places du modèle, trois attributs sont également ajoutés : "v*isited*", "*fromVisited*" et "*toVisited*". L'attribut "v*isited*" est utilisé pour indiquer que le code pour la place est généré. L'attribut "*fromVisited*" indique que cette place est traitée comme une place d'entrée pour la transition qui est en cours de traitement, alors que l'attribut "*toVisited*" indique si la place est traitée comme une place de sortie. Tous ces attributs temporaires sont initialisés à 0.

Les règles de notre grammaire sont présentées dans la Figure 5.10, et décrites comme suit :

Règle N°1: genPlaceDescription (priorité 1). Cette règle est appliquée pour localiser une place (non encore traitée : *visited = = 0*) pour générer sa description correspondante en PROD.

Règle N°2: genListOfInputPlace (priorité 2). Cette règle est appliquée pour localiser une place d'entrée (non encore traitée) de la transition qui est en cours de traitement (*current = = 1*) afin de générer le code PROD correspondant à cette place la liste "in{}" de la transition courante.

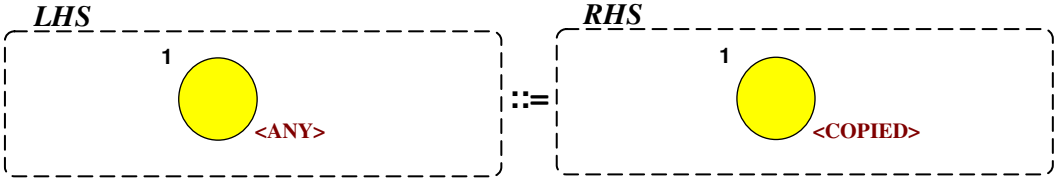
Règle N°3: genListOfOutputPlace (priorité 3). Cette est appliquée pour localiser une place de sortie (non encore traitée) de la transition qui est en cours de traitement (*current = = 1*) afin de générer le code PROD correspondant à cette place la liste "out{}" de la transition courante.

Règle N°4: EndOfTransDescription (priorité 4). Cette règle est appliquée pour générer le code PROD correspondant au contenu de l'attribut *condition* de la transition courante et de marquer cette transition traitée ("*visited = 1*").

Rule N°1- genPlaceDescription. Priority:1

Condition : Node(1). visited = = 0

LHS *RHS*

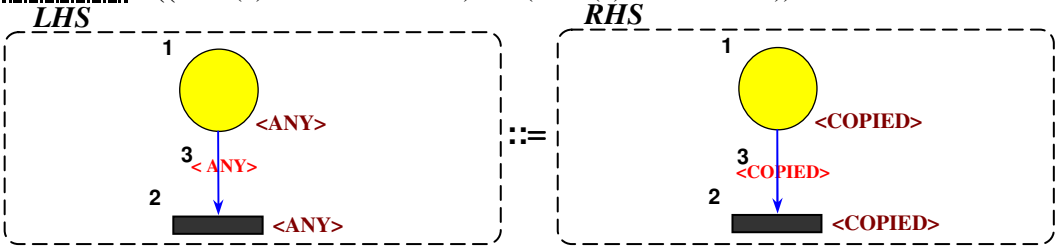


Action : Node(1). visited = 1

Rule N°2- genListOfInputPlace. Priority:2

Condition : ((Node(1).fromVisited = = 0) and (Node(2).current = = 1))

LHS *RHS*

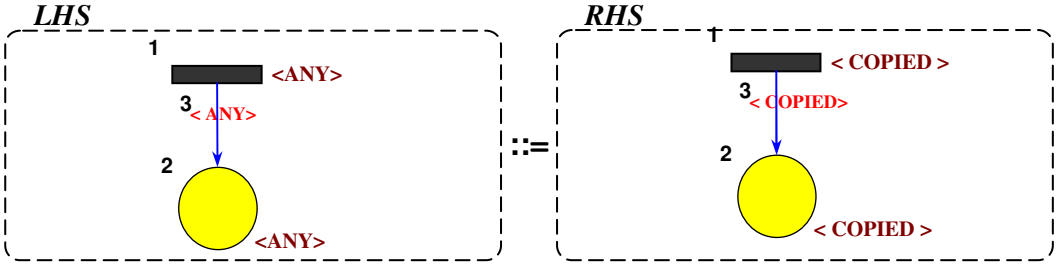


Action : Node(1).fromVisited = 1

Rule N°3- genListOfOutputPlace. Priority:3

Condition : ((Node(1). current = = 0) and (Node(2). toVisited = = 1))

LHS *RHS*

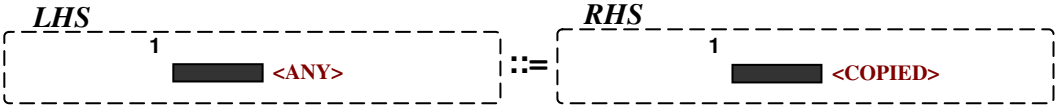


Action : Node(2). toVisited = 1

Rule N°4- EndOfTransDescription. Priority:4

Condition : ((Node(1). visited = = 0) and (Node(1).current = = 1))

LHS *RHS*

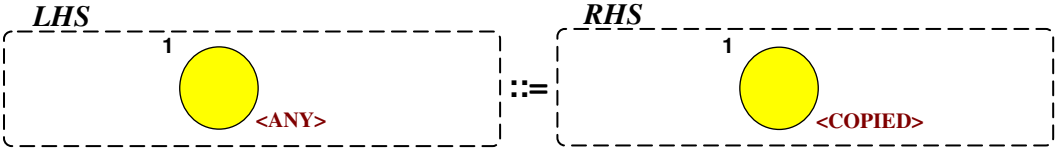


Action : Node(1).visited = 1
Node(1).current = 0

Rule N°5- InitialisePlaceFlags. Priority:5

Condition : ((Node(1).fromVisited = = 1) or (Node(1).toVisited = = 0))

LHS *RHS*

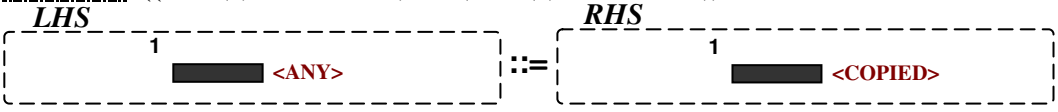


Action : Node(1). fromVisited = 0
Node(1). toVisited = 0

Rule N°6- SelectTransToDescribe. Priority:6

Condition : ((Node(1). visited = = 0) and (Node(1).current = = 0))

LHS *RHS*



Action : Node(1). current = 1

Figure 5.10: 2ème Génération de la description PROD

Règle N°5: InitialisePlace (priorité 5). Cette règle est appliquée pour localiser et initialiser les attributs temporaires dans les places pour traiter la prochaine transition.

Règle N°6: SelectTransition (priorité 6). Cette règle est appliquée afin de sélectionner une transition non encore traitée pour générer le code PROD correspondant.

À la fin d'exécution des règles, l'action finale de notre grammaire détruit les attributs temporaires des éléments du modèle et ferme le fichier de sortie.

Pour générer la description PROD à partir de l'éditeur des PrT-Nets, nous avons assigné l'exécution de cette grammaire de graphes à un bouton étiqueté " *Generate PROD Description*".

5.5 Exemple: Problème de Producteur/Consommateur

Le problème du producteur/consommateur est un exemple classique en informatique. Il s'agit d'un problème de synchronisation entre un ou plusieurs producteurs et un ou plusieurs consommateurs.

D'abord, nous présentons le problème et nous décrivons le modèle de cet exemple en utilisant l'outil généré. Puis, nous rapportons les résultats de l'application de nos grammaires de graphes sur cet exemple. La 1ère pour transformer le modèle G-Nets vers un modèle équivalent en PrT-Nets, et la 2ème pour générer le code PROD pour ce dernier.

5.5.1 Présentation du problème & sa spécification G-Nets

Dans cet exemple, nous supposons qu'un producteur est capable de produire *n* items (données) et un consommateur est en mesure de consommer un seul item à la fois. La Figure 5.11 présente une spécification G-Nets du problème producteur/consommateur créée dans notre outil.

Dans le G-Net "*Producer*", la méthode *mp* (method produce) est définie pour produire *n* items. Lorsque le G-Net "*Producer*" est appelé, un jeton contenant le nombre d'items (*n*) à produire est déposé dans la place *PR* (Producer ready). L'action associée à cette place décrémente simplement le champ n. Si n <0, la transition *pr* (producer resumes) est franchissable et l'appel du G-Net "*Producer*" est terminé lorsque la *GP* (Goal Place) est atteinte. Sinon, c'est la transition *rs* (request status consumer) qui est franchissable et le jeton est ajouté à la place *ISP* (*Consumer.ms*).

La place *ISP*(*Consumer.ms*) appelle le G-Net "*Consumer*" en utilisant la méthode *ms* (method status). Si le G-Net "*Consumer*" n'est pas prêt, la transition *nr* (consumer not ready) est franchie, et le G-Net "*Consumer*" est appelé une autre fois. Sinon, le consommateur est prêt et la transition *cr* (consumer ready) est franchissable.

Après le tir de la transition *cr,* un jeton est déposé dans la place *ISP (Consumer.mc)* et le G-Net "*Consumer*" est appelé par la méthode *mc* (method consume) avec l'item à consommer. Lorsque le jeton retourne du G-Net "*Consumer*", la transition *co* (consumed) est tirée et un jeton est ajouté à la place *PR* de nouveau.

Dans le G-Net "*Consumer*", huit places sont définies: *IN* (Inquire net Consumer), *RC* (ready to consume), *NotRC* (not ready to consume), *TC* (trigger consumer), *CO* (consuming), *WP* (waiting producer), *GPS* (end status) et *GPC* (item accepted). Ces places sont reliées par quatre transitions comme le montre la Figure 5.11. Les transitions sont: *na* (not available), *sa* (send acknowledge), *sc* (start consuming) et *ac* (already consumed).

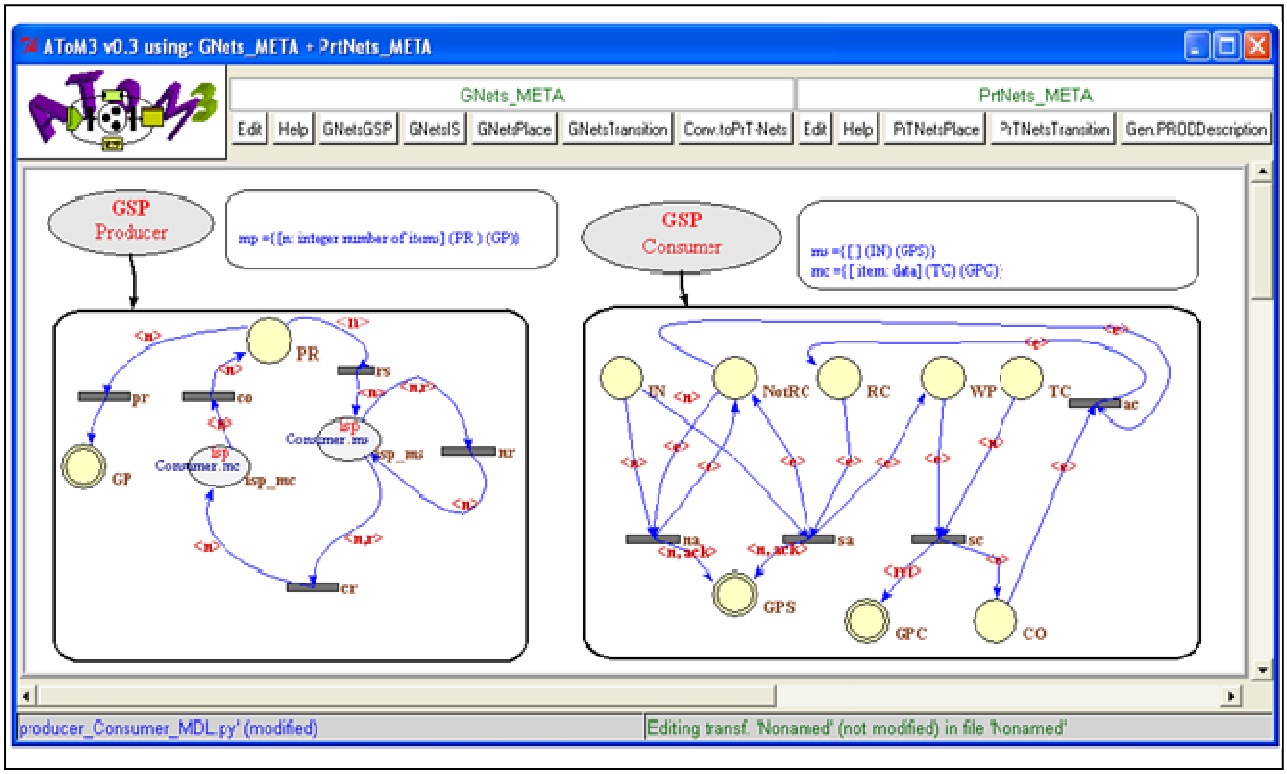


Figure 5.11: le problème de Producteur/Consommateur

5.5.2 Transformation de la spécification G-Nets en un modèle PrT-Nets

Cette étape doit avoir la spécification G-Nets du problème comme entrée. L'étape consiste à translater cette spécification graphique à un modèle équivalent décrit dans le formalisme PrT-Nets. Ce modèle contient d'une part l'équivalent de la structure interne de chaque G-Net et d'autre part, les interfaces qui imitent le mécanisme d'appel des G-Nets.

Après l'exécution de notre 1ère grammaire sur la spécification G-Nets du problème du producteur/consommateur, nous avons obtenu le modèle PrT-Nets équivalent présenté dans la Figure 5.12.

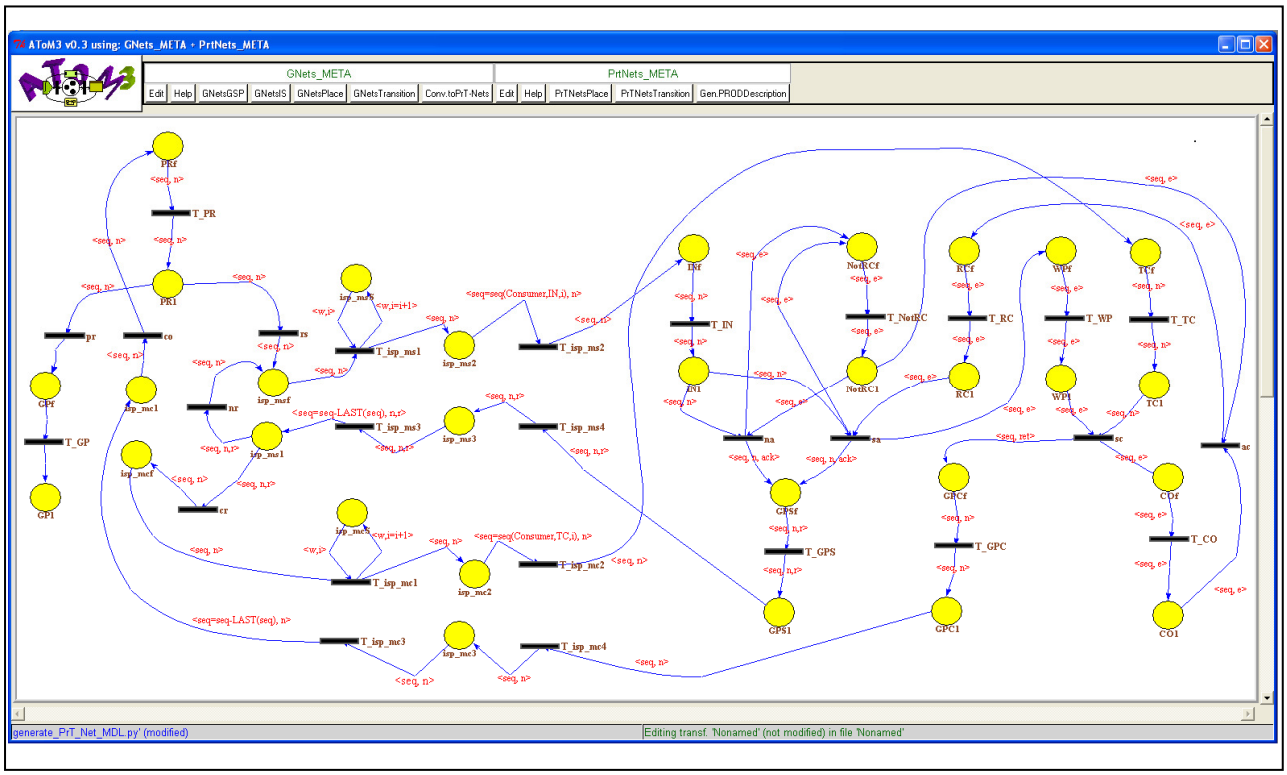


Figure 5.12: le modèle PrT-Nets équivalent au couple (Producer, mp)

5.5.3 Génération du Code PROD équivalent

Le résultat de l'étape précédente est l'entrée de celle-ci. Pour effectuer d'analyse d'atteignabilité du modèle PrT-Nets, nous avons besoin d'abord de générer sa description équivalente en PROD. En appliquant notre 2ème grammaire sur le modèle PrT-Nets, nous obtiendrons le fichier PROD présenté dans la Figure 5.13

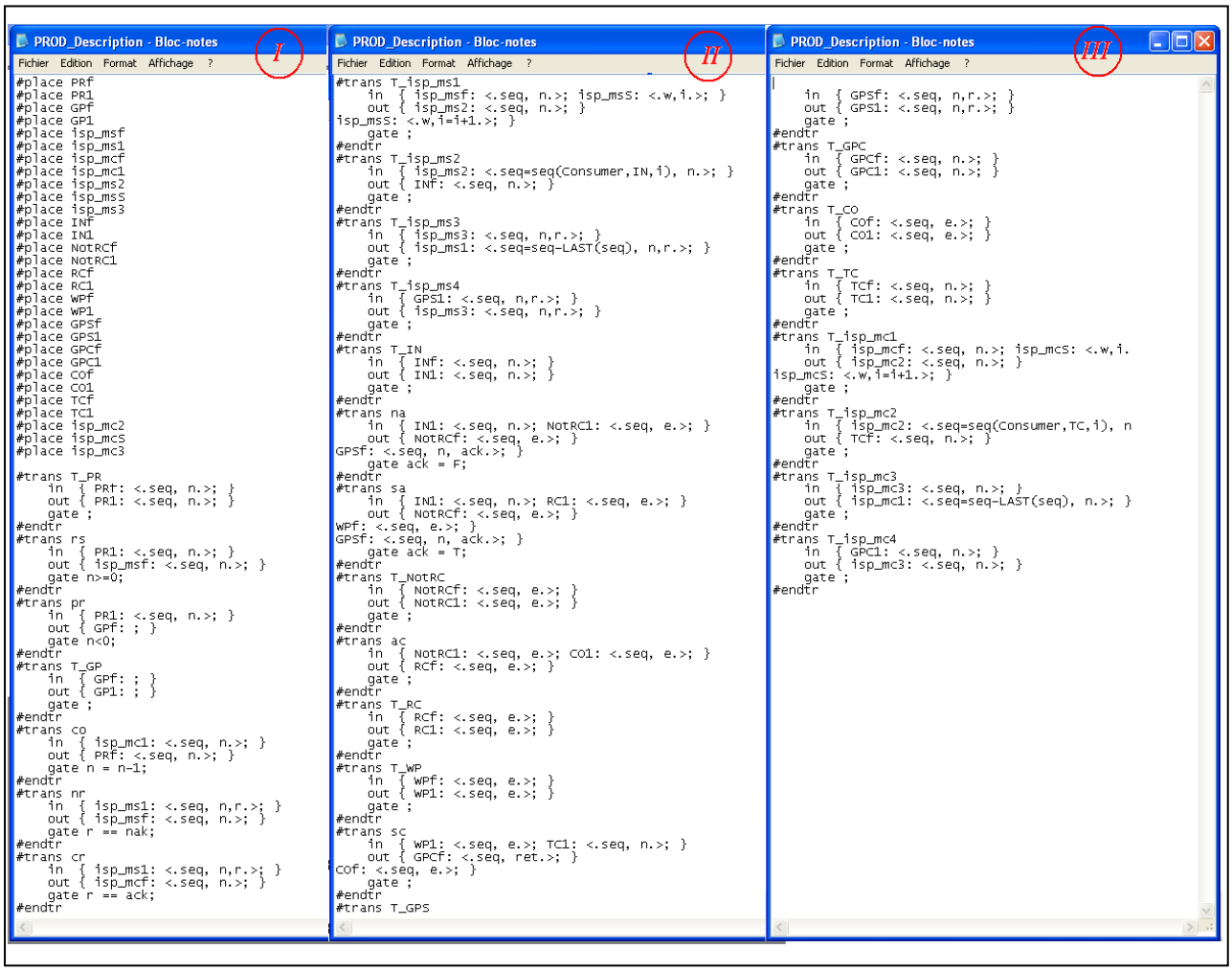


Figure 5.13: description PROD équivalente

5.6 Conclusion

Dans ce chapitre, nous avons proposé une plate-forme formelle et un outil de spécification et d'analyse des G-Nets basé sur la méta-modélisation et les grammaires de graphes. L'approche proposée permet de tirer profit des avantages de plusieurs langages et techniques formelles dans une plate-forme unifiée. Les G-Nets permettent de spécifier les systèmes dans une organisation très abstraite et modulaire. Les PrT-Nets, quant à elles, nous permettent de représenter un aspect comportemental du système dont le but est de l'analyser. L'analyseur PROD permet de vérifier les propriétés de chaque modèle PrT-Nets. Nous avons montré à travers un exemple notre plate-forme.

Notre objectif à travers cette plate-forme est d'assister et de faciliter l'utilisation des langages et méthodes formelles dans le processus de développement des systèmes d'information complexes.

Chapitre 06 :

Une Approche Intégrée UML/CPN pour la Modélisation et la Vérification du Comportement Dynamique des Systèmes

6.1 Introduction

Grâce à sa convivialité (ses notations graphiques) et sa relative souplesse d'emploi, le langage UML est devenu quasiment incontournable dans le domaine du développement des systèmes complexes. Il permet de décrire les différents aspects (statiques, dynamiques ou fonctionnels) de ces systèmes à travers différents types de diagrammes. Les aspects dynamiques sont largement modélisés avec les diagrammes d'états-transitions (les Statecharts) et les diagrammes de collaboration (ou de communication de en UML 2) [Heral87]. Les diagrammes d'états-transitions modélisent le comportement des objets du système en fonction des occurrences d'événements, tandis que les diagrammes de collaboration définissent les échanges de messages (ou d'événements) inter-objets [Booch99]. Cependant, UML souffre du manque de sémantique formelle, ce qui rend impossible de s'assurer de la cohérence et de vérifier le comportement des systèmes décrits par plusieurs diagrammes.

D'autre part, les méthodes formelles de conception permettent aussi la description de l'aspect dynamique des systèmes complexes mais de façon mathématique précise et rigoureuse. Leur principal intérêt est que la sémantique des langages de spécification est bien définie et permet donc de vérifier la correction des systèmes spécifiés. La spécification par les réseaux de Pétri (PNs) et leur extension "réseaux de Pétri colorés (CPNs)" sont parmi les méthodes formelles les plus connues. Elle offre plusieurs techniques de vérification et de validation permettant d'éviter beaucoup d'erreurs dès les premières phases du développement. En contrepartie, leur difficulté d'apprentissage et d'utilisation leur a souvent été reprochée.

L'utilisation conjointe des langages de spécification semi-formelle et formelle semble donc être un domaine prometteur avec pour objectif de tirer profit de ces deux types d'approches en termes de convivialité, d'expressivité et de rigueur.

Dans ce chapitre, nous proposons une approche intégrée UML/CPN qui répond à cette motivation. Cette approche permet d'une part de modéliser les aspects dynamiques des systèmes complexes avec un langage convivial et graphique; d'autre part de vérifier de façon formelle et potentiellement à un haut niveau d'abstraction les propriétés dynamiques de ces systèmes. Ce qui permet ainsi l'exploitation des outils disponibles pour les CPN, en particulier l'outil INA [INA] qui supporte aussi bien la simulation que la vérification des CPN.

Dans ce qui suit, nous examinons les travaux similaires relatifs à la vérification et à la transformation des diagrammes UML. Nous présentons notre approche intégrée de transformation automatique des diagrammes UML vers les réseaux de Petri Colorés. L'approche proposée est

basée sur la méta-modélisation et les grammaires de graphes en utilisant l'outil AToM^3. Enfin, nous illustrons par le biais une étude de cas les différentes transformations de l'approche.

6.2 Formalisation d'UML

UML n'est pas initialement basé sur une sémantique formelle qui permettrait de fonder le raisonnement et l'analyse de propriétés des systèmes. De nombreux travaux sont faits aussi bien sur la formalisation d'UML que sur le développement d'outils autour d'UML permettant d'appliquer des techniques d'analyse formelles à des spécifications UML. Ci-après, nous faisons un survol –non exhaustif– sur ces travaux de formalisation du langage UML.

6.2.1 Génération de tests

La génération de tests à partir de spécifications UML est relativement peu considérée par la communauté de chercheurs qui s'intéressent à formaliser UML.

Dans [Fleisch99], l'auteur a proposé une méthodologie de test basée sur des diagrammes décrivant la structure du système en utilisant la notation ROOM et sur un ensemble de cas d'utilisation accompagnés de scénarios sous forme de diagrammes de séquences UML. Les diagrammes décrivant le modèle sont compilés en un modèle de simulation.

Dans [Fröhlich00], l'approche proposée consiste à générer des tests à partir de cas d'utilisation. Les cas d'utilisation sont convertis en des diagrammes d'états-transitions équivalents. Ces diagrammes d'états-transitions sont réécrits, à leur tour, dans un langage de planification comme un ensemble de contraintes. L'outil "graphplan" permet de générer des tests en résolvant les contraintes de planification.

J. Offutt [Offutt99] présente une technique permettant de générer des jeux de données permettant de tester les différentes conditions apparaissant dans les gardes des transitions des machines à états.

6.2.2 Approches traductionnelles de la formalisation d'UML

De nombreuses techniques et algorithmes ainsi que plusieurs outils performants ont déjà été développés afin de répondre à des besoins similaires en rigueur formelle. En général, ces outils sont associés à un langage de spécification dédié. Ils nécessitent d'être adoptés avant de les utiliser. Traduire un sous-ensemble d'UML vers un de ces langages permet en effet à la fois:

- ✓ de donner indirectement une sémantique au sous-ensemble d'UML transféré.
- ✓ d'utiliser les techniques formelles associées à l'outil.

Plusieurs travaux ont adopté cette approche afin de permettre l'utilisation de techniques formelles avec UML.

6.2.2.1 Traduction vers Promela

Le modèle UML est traduit –au moins partiellement– dans le langage PROMELA [Holzman91] qui constitue le langage de spécification du model checker SPIN [Holzmann90, Holzmann97]. La limite de cet outil réside dans l'incapacité du model-checker à vérifier les systèmes réactifs ouverts. Cette approche a notamment été adoptée par [Paltor99], [Lilius99] et [Latella99].

6.2.2.2 Traduction vers SMV

Dans [Kwon00], l'approche présentée permet de faire du model-checking à partir des diagrammes d'états-transitions d'UML grâce à une traduction en SMV [McMillan92]. Cette approche ne considère qu'un seul diagramme d'états-transitions en isolation, sans envisager de communications inter-objets.

6.2.2.3 Traduction vers LOTOS

Une traduction d'un sous ensemble d'UML vers le langage de spécification LOTOS [ISO85] est proposée dans [Paulo00]. Cette traduction permet d'utiliser la boîte à outils CADP [Fernandez92, Fernandez96 et Garavel98] sur les spécifications OBLOG.

6.2.3.4 Traduction vers SDL

L'outil "ObjectGeade" permet de générer à partir des diagrammes d'états-transitions d'UML un processus du langage SDL. Ensuite, il est possible de simuler le comportement de la spécification. Il est à noter que la dernière révision de SDL est définie en tant que profile d'UML.

6.2.3 Les approches 'Méta'

Par opposition aux approches traductionnelles, il existe un courant visant à formaliser UML à l'aide d'UML lui-même. Son idée de base consiste à modéliser le domaine sémantique d'UML en UML. On retrouve dans cette catégorie certains travaux initiés par le groupe *precise UML* (pUML) [Clark01] ainsi que la proposition du groupe de travail sur l'Action Semantics [OMG03] qui travaille sur l'élaboration de la sémantique du langage UML.

6.2.4 Utilisation de Réseaux de Petri

De nombreux travaux se sont intéressés à la transformation des diagrammes d'UML vers les réseaux de Petri dans le but de bénéficier de leur batterie d'outils de vérification et d'analyse.

Les principaux travaux concernent les diagrammes dynamiques d'UML (les diagrammes de séquence, d'activité, d'états-transitions et de collaboration). Les autres diagrammes ont aussi été étudiés, mais pas autant que les diagrammes dynamiques. Nous citerons par exemple le travail dans [Bordbar 05] qui propose un outil CASE qui permet de transformer des diagrammes statiques d'UML vers Alloy. Alloy étant un langage formel textuel développé par Daniel Jackson qui permet la validation les diagrammes statiques d'UML.

Dans ce suit nous abordons les principaux travaux traitant les diagrammes dynamiques d'UML.

6.2.4.1 Diagramme de séquence

Les diagrammes de séquence sont souvent liés aux diagrammes de cas d'utilisation, car les diagrammes de séquence sont basés sur les scénarios décrits par les cas d'utilisation. Plusieurs approches de translation des diagrammes de séquence vers les réseaux de Petri ont été proposées. Dans [Fernandes07], les auteurs utilisent une étude de cas sur les spécifications d'un contrôleur d'ascenseur pour présenter une approche qui transforme des descriptions sous forme de diagrammes de cas d'utilisation et de séquence en modèle de réseau de Pétri coloré (CPN).

L'approche proposée dans [Staines07] transforme des diagrammes de séquence vers des "processors nets". Ces derniers combinent la théorie des réseaux de Petri avec d'autres notations sous forme d'instructions détaillées (les processors) associées aux transitions.

Nous citons également le travail de L.Guangyu et Y.Shuzhen [Guangyu09] qui ont présenté un algorithme de translation des diagrammes de séquence vers les réseaux de Petri Objets. Nous citons aussi les travaux de S. Emadi, dans [Emadi09a] et [Emadi09a], qui a proposé un algorithme de transformation d'un diagramme de cas d'utilisation et un diagramme de séquence en un model exécutable basé sur différentes extensions des réseaux de Petri.

6.2.4.2 Diagramme d'activité

Une approche de translation des diagrammes d'activité vers les réseaux de Petri a été proposée par T.S. Staines [Staines08]. La solution proposée consiste à transformer en premier lieu les diagrammes d'activité en modèles FMC-PND (Fundamental Modeling Concepts - Petri Net Diagram), puis à construire le CPN équivalent [Knöpfel 05]. Les règles de transformation des

différents opérateurs des diagrammes d'activités vers les FMC-PND ont été définies dans [Staines08].

D'autres travaux antérieurs à ce dernier sont ceux de H.Storrle. Dans ses travaux ([Störrle04a], [Störrle04b], [Störrle04c], [Störrle04d] et [Störrle05]), il a mis l'accent sur l'exploitation des diagrammes d'activités pour la modélisation du Dataflow en tentant de définir une sémantique formelle des diagrammes d'activités basée sur les réseaux de Petri.

Nous citons également le travail de B.Tebibel [Tebibel 07] qui présente une approche de formalisation modulaire des diagrammes d'activités partitionnés via les réseaux de Pétri à objet. Cette approche prend en considération les diagrammes d'objets et de séquence pour l'initialisation du modèle résultant.

6.2.4.3 Diagramme d'états-Transitions

Dans la littérature, les diagrammes d'états-transitions (les Statecharts) complémentés par les diagrammes de collaboration sont très répondus pour modéliser les aspects dynamiques des systèmes complexes et concurrents. Les diagrammes d'états-transitions représentent la façon dont évoluent (le cycle de vie) des objets appartenant à un système. La modélisation du cycle de vie est essentielle pour représenter et mettre en forme la dynamique du système. Les diagrammes de collaboration sont utilisés pour spécifier les interactions entre ces objets. De nombreux travaux se sont intéressés à la vérification de ces descriptions UML par différentes extensions de réseaux de Petri.

J.A Saldhana et M. Shatz [Saldhana01] ont proposé une méthodologie permettant de convertir les descriptions dynamiques du système en modèles des réseaux de Petri colorés (CPNs) pour supporter la simulation systématique. Les résultats de la simulation sont fournis sous forme d'un fichier de trace ainsi que d'un MSC (Message Sequence Charts) [Hu04]. Dans cette approche, les diagrammes d'états-transitions sont transformés en automates d'états finis qui sont ensuite transformés en un ensemble de modèles de réseaux de Petri colorés (CPNs). Pour obtenir un seul modèle, le diagramme de collaboration est utilisé pour guider l'interconnexion des modèles d'objets [Saldhana01]. Les auteurs ont développé un outil prototype pour permettre d'une part la génération automatique des réseaux de Petri colorés à partir d'une description UML; d'autre part de fournir à l'utilisateur une interface pour visualiser les résultats de la simulation.

Dans [King99], P. King et R. Pooley ont présenté une transformation des descriptions UML vers les réseaux de Petri stochastiques. La transformation est illustrée à travers un exemple d'un modèle du protocole qui gère les transactions dans une base de données distribuée. Les

diagrammes d'états-transitions et les diagrammes de collaboration sont combinés ensemble dans un même modèle pour représenter le comportement du système dans son intégralité. Le modèle résultat est transformé en un modèle des réseaux de Petri stochastique (SPN) qui va être utilisé comme modèle d'entrée de l'outil de vérification SPNP. L'outil SPNP permet d'analyser les propriétés temporaires du modèle d'entrée.

Une extension de ce travail peut être repérée dans [King00] où ils ont utilisé les réseaux de Petri stochastiques généralisés (GSPN).

6.2.5 UML et transformation de graphes

UML est un langage graphique. Les différents diagrammes utilisés dans sa spécification sont des graphes. Par conséquent, les approches de transformations de graphes sont utilisables pour transformer les modèles UML. Plusieurs travaux s'intéressent à la formalisation ou à l'enrichissement de la sémantique d'UML par les transformations de graphes.

Dans [Holscher06], les auteurs proposent une transformation du modèle UML en un système de transformations de graphes. Le système de transformations de graphes se compose de règles de transformation et d'un graphe de travail représentant l'état courant du système modélisé, ce qui permet aux concepteurs d'exécuter leurs modèles UML. La simulation de l'exécution visuelle du système est effectuée en appliquant les règles de transformation sur le graphe de travail. Le but de ce travail est la validation des modèles UML dans des phases préliminaires avant toute implémentation du système. L'approche couvre les diagrammes: de classe, de cas d'utilisation, d'objet, d'état transition et d'interaction.

Dans le même contexte, [kuske02] présente une association des règles de transformation de graphes aux opérations dans le diagramme de classe et aux transitions dans le diagramme d'états-transitions. Les différentes règles de transformation sont combinées dans un même système pour obtenir une description cohérente et unique de la sémantique.

Dans [Engels00] le diagramme de collaboration est interprété par un ensemble de règle de transformation, ce qui permettra de spécifier une sémantique dynamique pour les modèles UML. Dans [Baldan05] un cadre de la vérification du diagramme d'activité est présenté à l'aide des règles de transformations de graphes.

Dans [Ehrig05], L'équipe de K. Ehrig présente un outil graphique appelé "TIGER" (Transformation based generation of modeling environments). Cet outil permet de générer à partir de modèles graphiques (source) d'autres modèles graphiques (destination). Les auteurs exposent l'exemple de la dérivation des réseaux de Petri équivalents à des diagrammes d'activités. Après

dérivation du modèle, l'outil propose un moyen pour vérifier et valider les modèles obtenus. L'accent est mis beaucoup plus sur l'aspect visuel, en particulier sur la façon visuelle pour spécifier les règles de transformations.

Dans [Guerra03], E. Guerra et J. De Lara présentent l'idée d'une plate-forme permettant de vérifier une modélisation UML. L'approche proposée consiste à établir des méta-modèles pour l'ensemble des diagrammes UML puis à les transformer vers plusieurs catégories des réseaux de Petri (PN). La syntaxe des PNs est spécifiée également par un méta-modèle. La transformation est formellement décrite par la notion des grammaires de graphe. Dans leur article [Guerra03], les auteurs ont défini les règles de transformation des diagrammes de séquence vers réseaux de Petri ordinaire.

6.3 L'Approche Intégrée UML/CPN Proposée

A travers l'analyse des travaux présentés dans la section précédente, nous nous sommes intéressés aux deux points suivants:

- L'approche générale de simulation UML-CPN proposée par J.A Saldhana et M. Shatz [Saldhana01] et l'initiative de transformation proposée qui est basée sur bonne séparation des préoccupations relatives à la construction du modèle CPN global du système.
- La définition syntaxique (la méta-modélisation) des diagrammes UML et des réseaux de Petri et la description formelle de la transformation par la notion des grammaires de graphes présentées dans l'approche d'E. Guerra et J. De Lara [Guerra03].

Dans cette section, nous présentons notre approche intégrée avec les outils associés qui permettent de modéliser certaines vues dynamiques des systèmes exprimées à l'aide des modèles UML, puis de transformer ces derniers vers leurs modèles équivalents en CPN dans le but de les vérifier de façon formelle. La vérification formelle des modèles CPN se fait en exploitant l'analyseur INA [INA] qui supporte la vérification des CPN. L'approche est basée sur l'utilisation combinée de la méta-modélisation et des grammaires de Graphes.

Dans notre approche, nous supposons que le comportement du système est spécifié par un ensemble de diagrammes d'états-transitions et un diagramme de collaboration. Afin de dériver le modèle CPN global du système à partir de cette spécification comportementale, nous avons automatisé l'approche proposée par J.A Saldhana et M. Shatz [Saldhana01]. Pour analyser le modèle CPN obtenu, nous avons également automatisé la génération de la spécification INA équivalente.

Puisque les diagrammes d'états-transitions peuvent contenir des états composites, la conversion efficace vers les CPN exige que ces états soient "aplatis". Les diagrammes d'états-transitions sont d'abord convertis en automates d'états finis (Flat State Machine (*FSM*)) qui ne contiennent que des états simples et des transitions. Ensuite, les modèles FSMs sont transformés en un type de réseaux de Petri à objets appelé (Object Net Modèles (*ONM*)) [Saldhana01]. Enfin, le diagramme de collaboration est utilisé pour interconnecter l'ensemble des modèles ONMs afin d'obtenir un seul modèle représentant le comportement du système dans son intégralité (voir la Figure 6.1). À partir du modèle CPN global résultat, n'importe quel outil d'analyse des réseaux de Petri colorés peut être utilisé pour vérifier le comportement du système représenté.

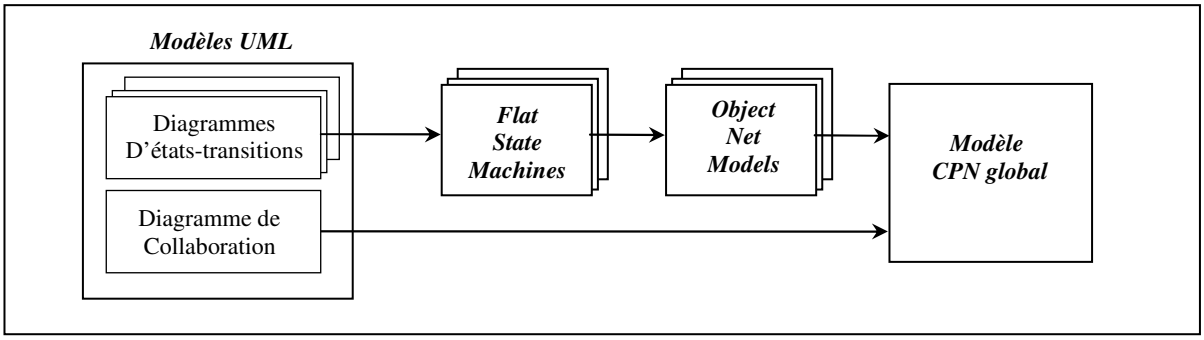


Figure 6.1: L'architecture de l'approche de J.A Saldhana et M. Shatz [Saldhana01]

Un modèle ONM est dérivé d'un diagramme d'états-transitions. Il décrit le comportement et le mécanisme de routage des jetons dans l'objet. La structure d'un modèle ONM est constituée de deux parties: un modèle du cycle de vie de l'objet (lifetime model (*LM*)) et la structure de routage des jetons (voir la Figure 6.2).

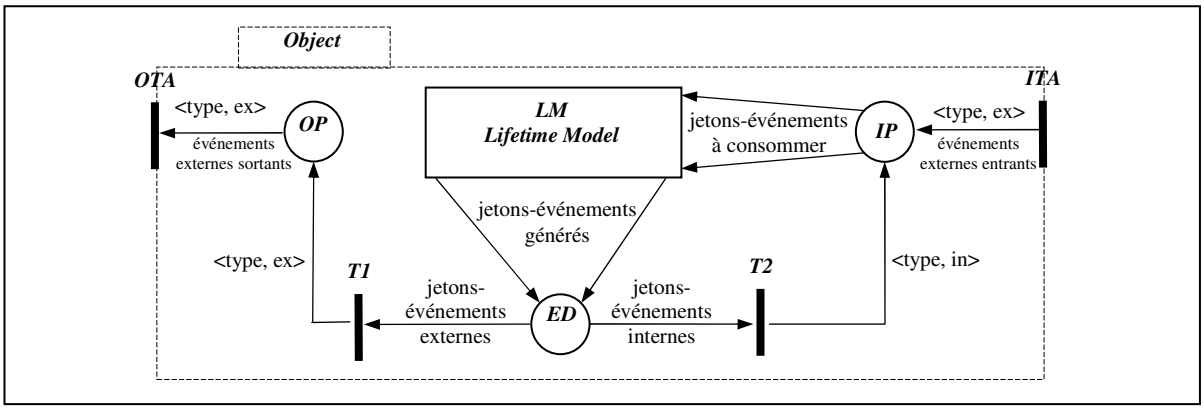


Figure 6.2: La structure d'un modèle Object Net Model (ONM)[Saldhana01]

LM représente le modèle CPN équivalent à un diagramme d'états-transitions d'un objet. L'idée de base de la transformation consiste à remplacer chaque état par une place et chaque transition par une transition dans le modèle CPN. Le concept d'événements est un concept clé dans

la sémantique des diagrammes d'états-transitions. La production, l'envoi et la consommation des événements déterminent la sémantique des diagrammes d'états-transitions. Par conséquent, cette sémantique est remplacée par celle des jetons dans les CPNs. Les différents événements définissent les types des jetons dans le modèle CPN.

La structure de routage des jetons contient trois places (in-place (*IP*), out-place (*OP*) et event-dispatcher place (*ED*)) et quatre transitions (*T1*, *T2*, input transition arc (*ITA*) et output transition arc (*OTA*)). Puisque les événements sont remplacés par des jetons dans CPN, on se réfère aux jetons qui remplacent les événements dans les diagrammes d'états-transitions par *jetons-événements*. La place IP de l'objet contient les jetons-événements à consommer par LM de l'objet. Par conséquent, des arcs sortants de la place IP sont ajoutés vers les transitions (dans le modèle LM) associées aux transitions déclenchées (sensibilisées) par ces événements dans le diagramme d'états-transitions de l'objet. La place ED contient les jetons-événements qui seront générés par l'objet. Chaque transition dans LM qui génère un jeton-événement est reliée à la place ED. Le jeton-événement généré doit avoir un type: *externe* ou *interne*. Si le type du jeton-événement est *interne*, il sera acheminé vers la place IP, sinon (le type est *externe*) il passera à la place OP. La place OP contient les jetons-événements envoyés aux autres objets. Afin d'assurer la communication inter-objets, une place particulière (Internal Linking Place (*ILP*)) est utilisée pour acheminer les jetons-événements entre les modèles ONMs. Pour plus de détails voir [Saldhana01].

Notre approche comprend deux étapes comme l'illustre la Figure 6.3 [kerkouche10b]:

- ⇨ La première étape consiste à méta-modéliser les diagrammes UML et les formalismes considérés. Plus précisément, nous avons défini un méta-modèle pour les diagrammes d'états-transitions et un autre méta-modèle pour les diagrammes de collaboration en utilisant l'outil de méta-modélisation AToM^3. De même, nous avons également défini des méta-modèles pour les FSMs et les ONMs. Ensuite, nous avons utilisé l'outil AToM^3 pour générer automatiquement un outil de modélisation visuelle pour tous les diagrammes UML et les formalismes en fonction de leurs méta-modèles.
- ⇨ La deuxième étape consiste à définir les transformations de modèles. Afin de parvenir à un processus automatique et correcte de la transformation, nous avons proposé d'utiliser les grammaires de graphes et le système de réécriture de graphes de l'outil AToM^3. Dans notre approche, nous avons défini trois grammaires de graphes:
  - ➢ ***1ère GG***: pour convertir les diagrammes d'états-transitions en modèles FSMs équivalents.

- ***2ème GG***: pour transformer les modèles FSMs obtenus de la 1ère GG vers leurs modèles ONMs équivalents et de les interconnecter en fonction des échanges de messages/événements décrits dans le diagramme de collaboration. Le modèle obtenu de cette étape est un modèle CPN unique pour le système étudié.
- ***3ème GG***: pour générer la description INA équivalente au modèle CPN global.

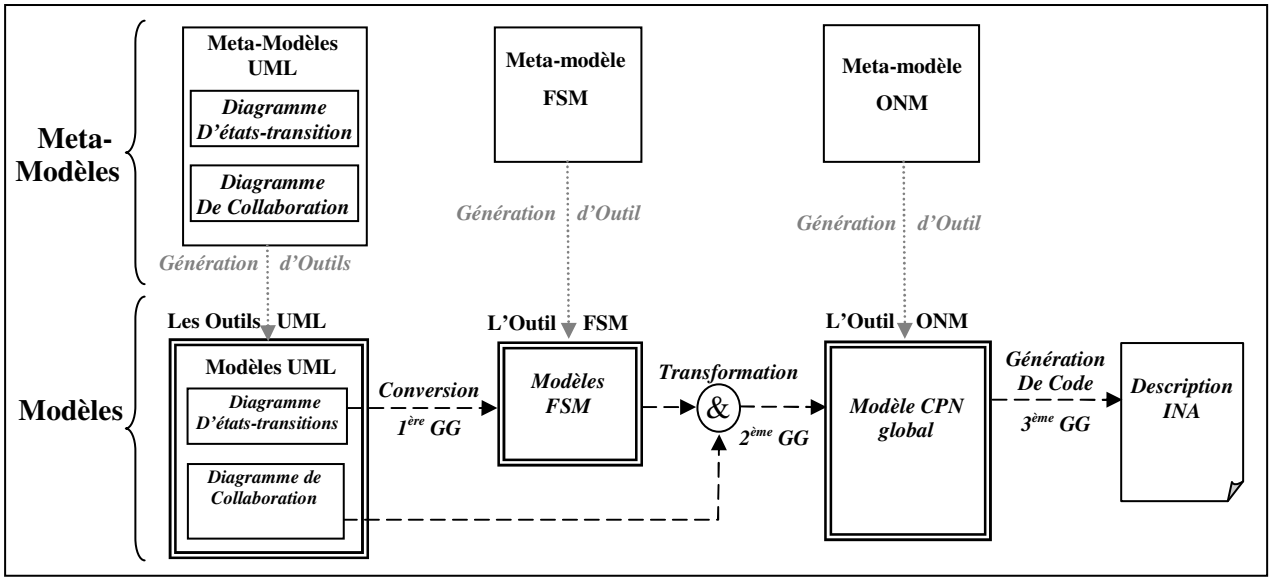


Figure 6.3: L'Approche Proposée

6.3.1 Méta-Modélisation des Diagrammes UML et des Formalismes considérés

Pour définir un langage de modélisation, il est nécessaire de définir sa syntaxe abstraite (les éléments du langage, leurs attributs, leurs relations et leurs contraintes), ainsi que les informations concernant sa syntaxe concrètes (l'apparence graphique de ses éléments et leurs relations). Le méta-formalisme utilisé dans notre travail est le diagramme de classes UML et les contraintes sont exprimées dans le code Python.

Dans notre travail, nous considérons un sous-ensemble du diagramme d'états-transitions d'UML [Booch99]. Un diagramme d'états-transitions est constitué d'états (simples et composites), de transitions, d'événements, d'actions qui génèrent les événements, et d'états initiaux et finaux. Un état composite peut contenir (envelopper) des sous-états qui peuvent être séquentiels ou concurrents. Un diagramme d'états-transitions est graphiquement représenté par un graphe comportant des états, matérialisés par des rectangles aux coins arrondis, et des transitions, matérialisées par des arcs orientés liant les états entre eux.

Nous avons proposé pour méta-modéliser les diagrammes d'états-transitions les classes et les associations suivantes (voir la Figure 6.4 (a)) [Kerkouche09c, kerkouche09d]:

Classe "Statechart": elle représente un diagramme d'états-transitions. Elle possède un attribut clé *Name*.

Classe "SC_State": Cette classe décrit les états simples. Elle possède trois attributs: *Name*, *EntryAction* et *ExitAction*.

Classe "SC_CompositeState": Cette classe représente les états composites. Elle hérite de *SC_State* tous ses attributs, les multiplicités et les associations.

Classe "SC_Initial": Cette classe représente l'état initial d'un diagramme d'états-transitions. Elle représente aussi l'état initial d'un état composite séquentiel ou les états initiaux d'un état composite concurrent.

Classe "SC_Final": cette classe représente l'état final d'un diagramme d'états-transitions.

Association "StatechartStart": Cette association relie un diagramme d'états-transitions avec son état initial.

Association "SC_InitialConnection": Cette association permet de relier l'état initial du diagramme d'états-transitions avec un état (simple ou composite).

Association "SC_FinalConnection": Cette association permet de relier un état (simple ou composite) avec un état final.

Association "SC_Transition": Cette association représente la transition d'un état source à un état destination (qui peut-être l'état source). Il contient deux attributs: *event* et *action*.
Association "has_Inside": Il s'agit d'une association entre un état composite et un autre état (qui peut-être aussi composite). Cette association exprime la notion d'hiérarchie des états, c'est-à-dire les états qui sont à l'intérieur un état composite.

Association "has_Initial": Cette association exprime la notion d'hiérarchie entre un état composite et son état initial.

Association "has_Final": Cette association exprime la notion d'hiérarchie entre un état composite et son état final.

Puisque les diagrammes de collaboration consistent en des objets s'interagissent par envoi des messages (ou des événements), nous avons proposé pour les méta-modéliser une classe "*CollaborationObject*" pour représenter les objets et une association "*CollaborationLink*" pour représenter les interactions inter-objets comme l'illustre la Figure 6.4(b) [Kerkouche09c, kerkouche09d].

Un FSM est un automate d'états finis sans états composites. La Figure 6.4(c) présente notre méta-modèle proposé pour le formalisme FSM. Il se compose d'une classe "*FSMState*" pour représenter les états et une association "*FSMTransition*" pour représenter les transitions [Kerkouche09c]. Le dernier formalisme utilisé dans notre approche est le formalisme ONM. Les ONMs sont une catégorie des réseaux de Petri à objets qui utilise les places et les transitions. Pour définir un méta-modèle pour les ONMs nous avons proposé deux classes et deux associations comme le montre la Figure 6.4(d) [Kerkouche09c, kerkouche09d].

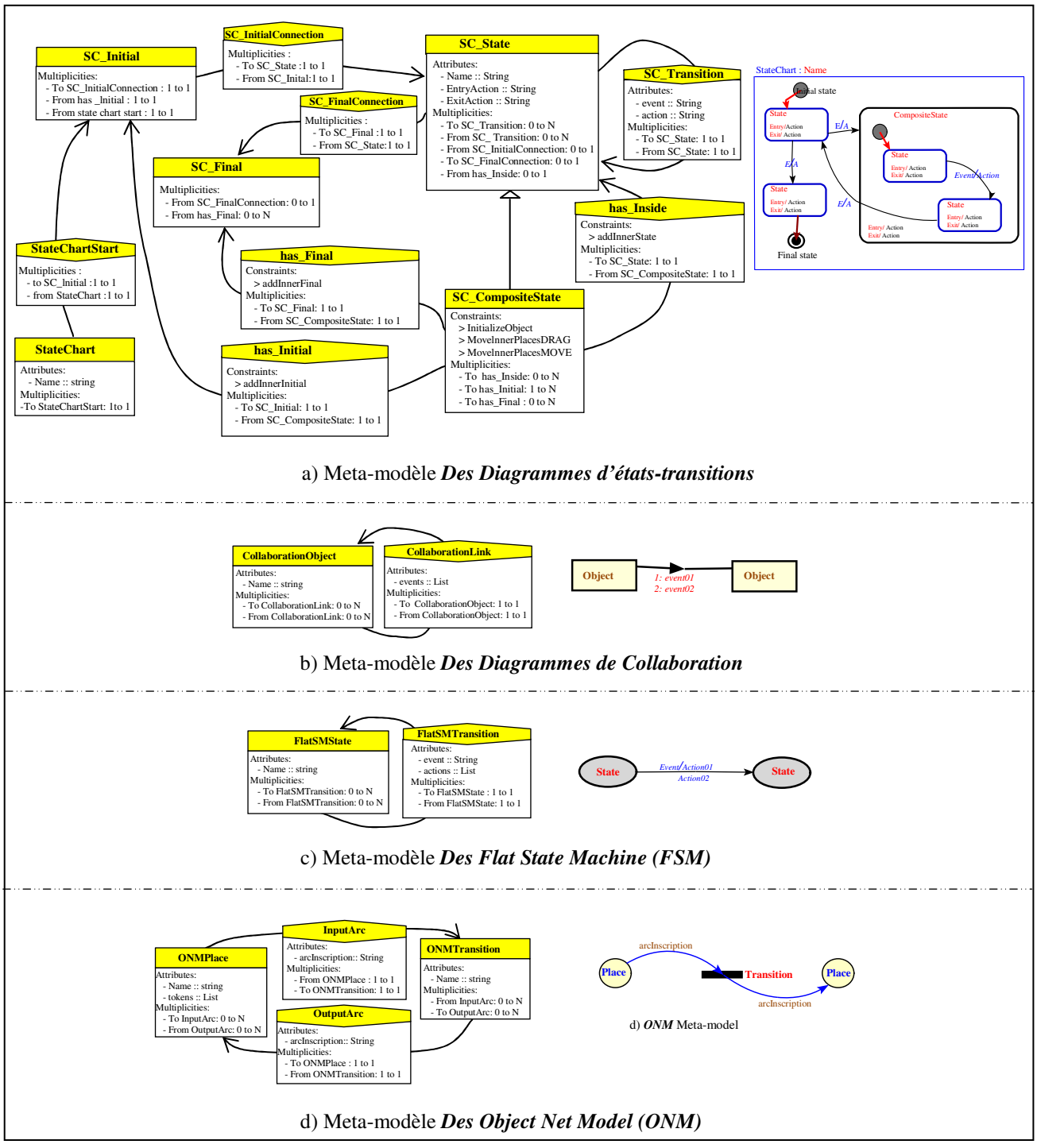


Figure 6.4: Méta-Modèles des Diagrammes/Formalismes utilisés

Nous avons également spécifié l'aspect graphique de chaque entité des diagrammes et des formalismes considérés en fonction de sa notation graphique appropriée. Il est à noter que les associations "*has_Inside*", "*has_Initial*" et "*has_Final*" définies dans le méta-modèle des diagrammes d'états-transitions pour exprimer la composition ne sont pas représentées graphiquement.

En utilisant AToM^3, nous avons généré pour chaque diagramme/formalisme un éditeur graphique permettant de créer les entités définies dans son méta-modèle. Puisque AToM^3 est un outil de modélisation multi-paradigme, nous pouvons avoir dans l'interface utilisateur tous les outils générés à la fois (voir Figure 6.11).

6.3.2 Les Grammaires de graphes

Dans cette section, nous allons utiliser AToM^3 pour définir les trois grammaires de graphes qui définissent les transformations de modèles dans notre approche.

6.3.2.1 1ère GG : Conversion des diagrammes d'états-transitions en des modèles FSMs

Pour convertir un diagramme d'états-transitions en son équivalent en modèle FSM, nous avons proposé 23 règles [Kerkouche10a, kerkouche10b]. Les règles sont présentées dans les Figures 6.5 et 6.6.

L'action initiale de notre 1ère grammaire de graphes consiste à décorer tous les états composites et les transitions du modèle par des variables temporaires utilisées pour spécifier les conditions des règles. Pour chaque état composite, deux attributs sont ajoutés: "*Traversed*" et "*Processed*". L'attribut "*Traversed*" est utilisé pour indiquer que l'état composite a été traversé par la grammaire, alors que l'attribut "*Processed*" est utilisé pour indiquer que l'état composite est déjà converti en FSM. Dans les transitions, un attribut est également ajouté: "*Converted*" qui indique que la transition est traitée ou non. Tous ces attributs temporaires sont initialisés à 0.

La stratégie de la conversion de la 1ère GG peut être résumée par les étapes suivantes:

La première étape consiste à sélectionner un diagramme d'états-transitions (non traité) et de convertir son état initial en un état FSM (règle N°22 ou règle N°23 pour l'état composite). Plus précisément, cette règle consiste à associer un état FSM à l'état initial du diagramme d'états-transitions par le biais d'un arc générique qui permet de relier les éléments des langages différents dans l'outil AToM^3.

Dans la deuxième étape, la grammaire parcourt le diagramme d'états-transitions sélectionné à travers ses transitions à partir de l'état initial (converti dans la 1ère étape) aux états

suivants, et ainsi de suite jusqu'à l'état final. Les états suivants peuvent être des états simples ou états composites. Pour les états simples, ces états et les transitions entrantes seront convertis en des états et des transitions FSM (les règles N°5 et N°6). Dans le cas où l'état simple suivant est déjà traité, il ne sera pas converti une autre fois; c'est pourquoi la règle N°5 a une priorité plus élevée que la règle N°6. Lorsque l'état suivant est un état composite, la grammaire marque cet état comme traversé (*Traversed* = 1) sans qu'elle convertit la transition entrante (la règle N°7). A la fin de cette étape, tous les états du diagramme d'états-transitions du premier niveau de la hiérarchie sont traversés, les états simples sont convertis et les états composites ne sont pas encore convertis.

Dans la troisième étape, les états composites traversés (*Traversed* = 1) du diagramme d'états-transitions seront convertis. La stratégie de conversion des états composites est la même utilisée pour convertir les diagrammes d'états-transitions. Elle est basée sur le processus du parcours de l'état composite et cela de façon récursive pour ses états composites imbriqués (les règles N°1, N°2, N°3 et N°4).

La dernière étape consiste à relier au modèle FSM associé au diagramme d'états-transitions les segments FSM équivalents à ses états composites, et ainsi de suite pour tous les autres niveaux de la hiérarchie des états composites. Par exemple, la règle N°14 convertit la transition sortante d'un état composite à un état simple en sa transition FSM équivalente dans le modèle FSM.

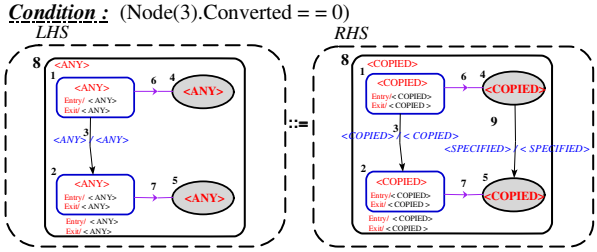
Pour convertir un état composite concurrent, la grammaire génère deux états FSM spéciaux (*Fork* et *Joint*) qui imitent la sémantique de la concurrence (la règle N°20). L'état *Fork* dénote l'activation parallèle de tous les états successeurs, alors que l'état *Joint* synchronise l'activation des états successeurs de l'état composite concurrent. La conversion des états imbriqués est effectuée comme décrit dans la deuxième étape. Ensuite, tous les états initiaux de l'état composite seront reliés à l'état *Fork* (la règle N°18) et tous les états finaux seront reliés à l'état *Joint* (la règle N°17). La Figure 6.9 montre la conversion d'un état composite concurrent en modèle FSM.

Pour convertir un état composite séquentiel, la grammaire localise l'état initial de l'état composite et génère un état FSM équivalent (la règle N°21). A partir de l'état initial elle commence le processus de conversion décrit dans la deuxième étape.

A la fin d'exécution des règles, l'action finale détruit les attributs temporaires des éléments du diagramme d'états-transitions.

1.- Conv_TransitionBetweenNestedStes. Prio.1

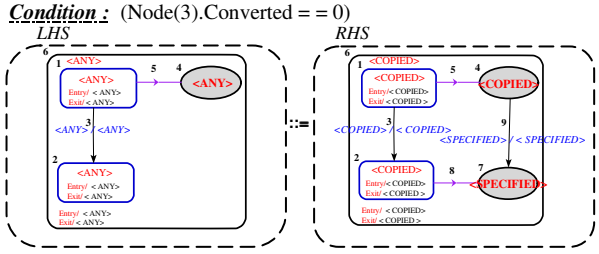
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

2.- Conv_NestedTransitionAndSte. Prio.2

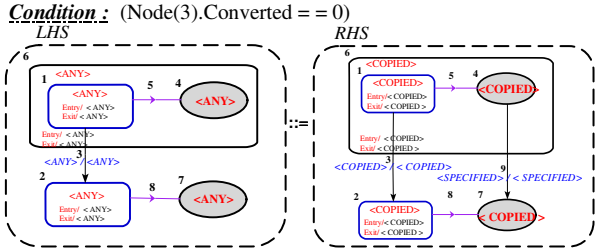
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

3.- Conv_Transition_FromNestedSte2SimpleSte. Prio.3

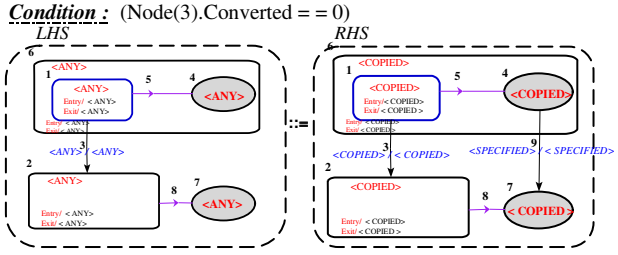
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

4.- Conv_Transition_FromNestedSte2CompositeSte. Prio.4

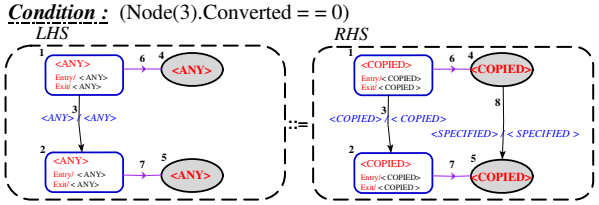
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

5.- Conv_Transition_ BetweenSimpleStes. Prio.5

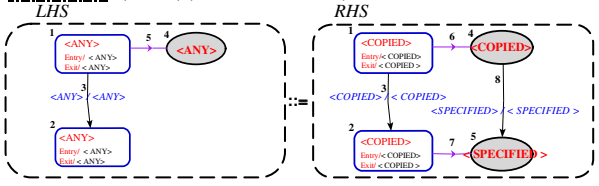
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

6.- Conv_TransitionAndSimpleSte. Prio.6

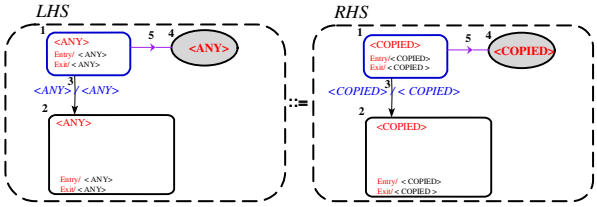
Condition : (Node(3).Converted = = 0)



Action : Node(3).Converted = 1

7.- SetCurrent_CompositeSteFromSimpleSte. Prio.7

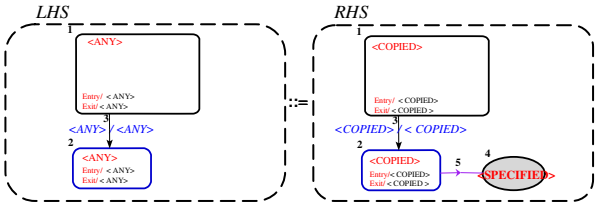
Condition : ((Node(3).Converted = = 0) and
(Node (2).Traversed = = 0) and (Node (2).Processed = = 0))



Action : Node(2). Traversed = 1

8.- Conv_NextSimpleSteFromCompositeSte. Prio.8

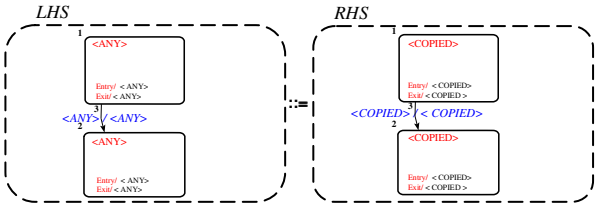
Condition : ((Node(3).Converted = = 0) and
(Node (2).Traversed = = 1) and (Node (2).Processed = = 0))



Action :

9.- SetCurrent_CompositeSteFromCompositeSte. Prio.9

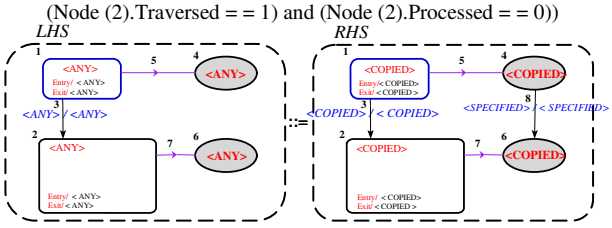
Condition : ((Node(3).Converted = = 0) and
(Node (2).Traversed = = 1) and (Node (2).Processed = = 0) and
(Node (3).Traversed = = 0) and (Node (3).Processed = = 0))



Action : Node(2). Traversed = 1
Node(3). Converted = 1

10.- Conv_EntryTransitionFromSimpleSte. Prio.10

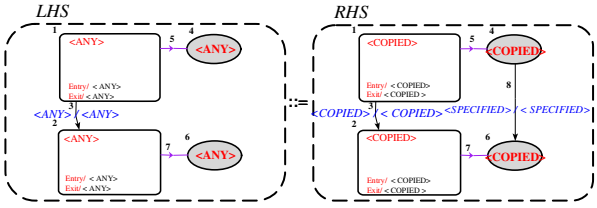
Condition : ((Node(3).Converted = = 0) and
(Node (2).Traversed = = 1) and (Node (2).Processed = = 0))



Action : Node(3).Converted = 1

11.- Conv_ EntryTransitionFromConcurrentCompositeSte. Prio.10

Condition : ((Node(3).Converted = = 0) and
(Node (1).Traversed = = 1) and (Node (1).Processed = = 0))
(Node (2).Traversed = = 1) and (Node (2).Processed = = 0))

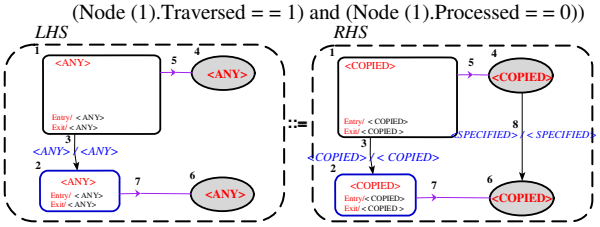


Action : Node(3).Converted = 1

Figure 6.5: 1ère GG, les Règles N°1-11.

12.- Conv_ExitTransitionFromConcurrentCompisiteSte. Prio.12

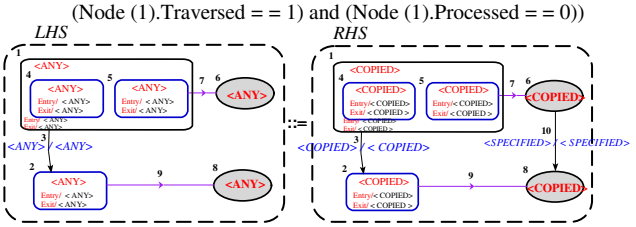
Condition : ((Node(3).Converted = = 0) and (Node (1).Traversed = = 1) and (Node (1).Processed = = 0))



Action : Node(3).Converted = 1

13.- Conv_OtherExitTransitionsFrom SequentialCompositeSte2SimpleSte. Prio.13

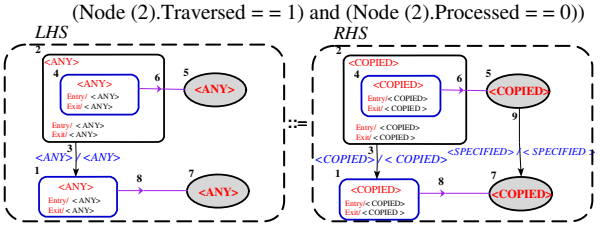
Condition : ((Node(3).Converted = = 0) and (Node (1).Traversed = = 1) and (Node (1).Processed = = 0))



Action : Node(3).Converted = 1

14.- Conv_ExitTransitionsFrom SequentialCompositeSte2SimpleSte. Prio.14

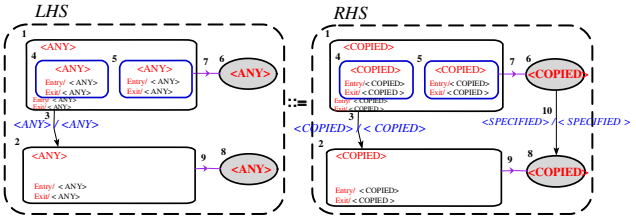
Condition : ((Node(3).Converted = = 0) and (Node (2).Traversed = = 1) and (Node (2).Processed = = 0))



Action : Node(3).Converted = 1

15.- Conv_OtherExitTransitionsFromSequential CompositeSte2CompositeSte. Prio.15

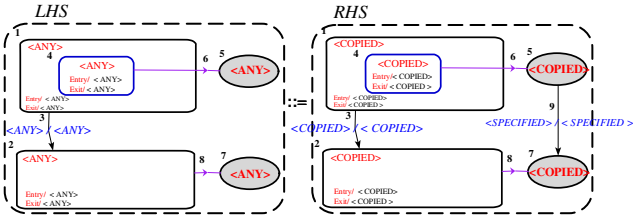
Condition : (Node(3).Converted = = 0) (Node (1).Traversed = = 1) and (Node (1).Processed = = 0)) and (Node (2).Traversed = = 1) and (Node (2).Processed = = 0))



Action : Node(3).Converted = 1

16.- Conv_ExitTransitionsFromSequential CompositeSte2CompositeSte. Prio.16

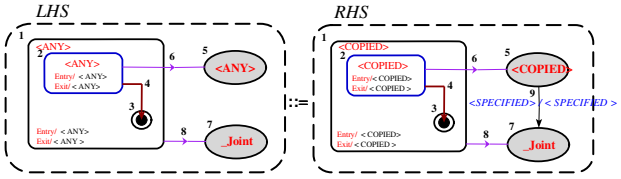
Condition : (Node(3).Converted = = 0) (Node (1).Traversed = = 1) and (Node (1).Processed = = 0)) and (Node (2).Traversed = = 1) and (Node (2).Processed = = 0))



Action : Node(3).Converted = 1

17.- Add_EndingTransitionInConcurrentCompositeSte. Prio.17

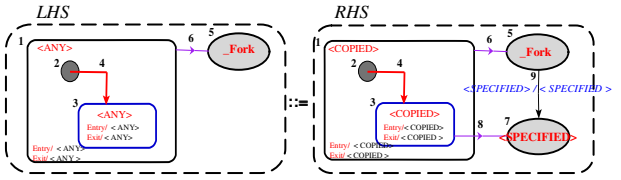
Condition : ((Node (1).Traversed = = 1) and (Node (1).Processed = = 1))



Action :

18.- Conv_InitialSteInConcurrentCompositeSte. Prio.18

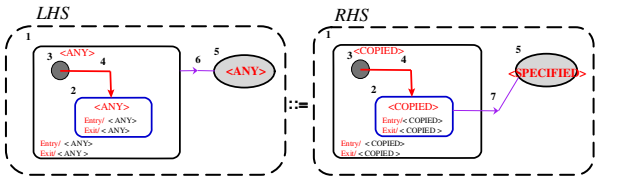
Condition : ((Node (1).Traversed = = 1) and (Node (1).Processed = = 1))



Action :

19.- Conv_InitialSteInSequentialCompisiteSte. Prio.19

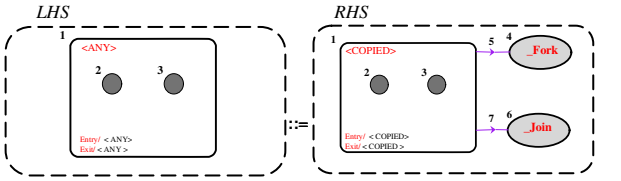
Condition :



Action :

20.- Conv_ConcurrentCompisiteSte. Prio.20

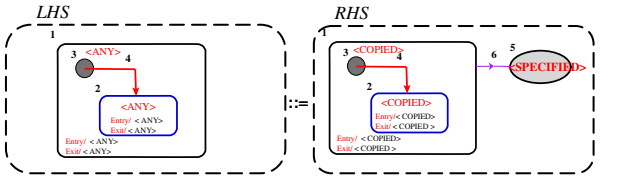
Condition : (Node (1). Traversed = = 1) and (Node (1). Processed = = 0)



Action : Node (1). Processed = 1

21.- Conv_SequentialCompisiteSte. Prio.21

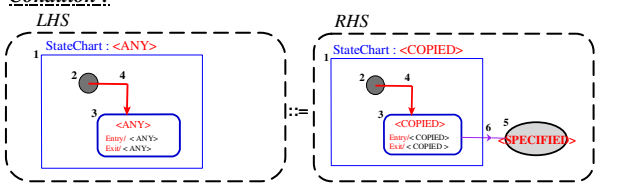
Condition :



Action :

22.- Conv_InitialState. Prio.22

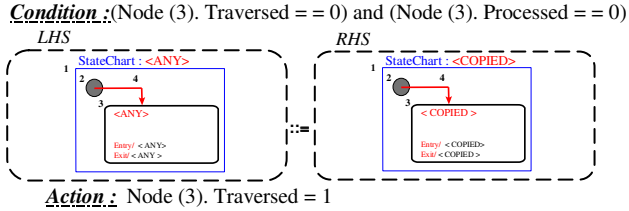
Condition :



Action :

23.- Conv_ InitialCompositeState. Prio.23

Condition :(Node (3). Traversed = = 0) and (Node (3). Processed = = 0)



Action : Node (3). Traversed = 1

Figure 6.6: 1ère GG, les Règles N°12-23

6.3.2.2 2ème GG : Transformation des modèles FSMs et diagramme de Collaboration vers un modèle CPN

La 2ème GG transforme les modèles FSM obtenus de la 1ère grammaire vers un modèle global en réseaux de Petri coloré (CPN). Dans cette grammaire de graphes, nous avons proposé 20 règles qui seront exécutées dans l'ordre ascendant de leurs priorités (voir les Figures 6.7 et 6.8) [Kerkouche10a, kerkouche10b]. Durant l'exécution des règles de la grammaire, les graphes intermédiaires contiennent les éléments des trois diagramme/formalismes: les diagrammes d'états-transitions, les FSMs et les CPNs. A la fin de l'exécution le graphe obtenu est un modèle CPN global du système étudié.

L'idée derrière la transformation peut être décrite dans les étapes suivantes:

Dans la première étape, la grammaire de graphes sélectionne un diagramme d'états-transitions afin de lui générer une structure de routage des jetons (les places (IP, ED et OP), les transitions (OTA, ITA, T1 et T2)) et une place équivalente à son état initial (la règle N°14).

La deuxième étape consiste à créer le modèle LM pour le diagramme d'états-transitions sélectionnés en se basant sur son modèle FSM équivalent. Le processus de transformation est basé sur le parcours du modèle FSM de la même manière que dans la 1ère GG. Une place est créée pour chaque état FSM, et une transition est créée aussi pour chaque transition FSM (les règles N°1 et N°2). Pour les états de type *Fork* ou *Joint*, la grammaire génère des transitions en ONM (les règles N°3, N°4 et N°5). La Figure 6.9 montre la transformation d'un état composite concurrent vers son équivalent en ONM.

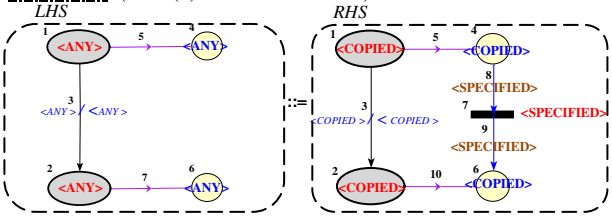
Dans la troisième étape, la grammaire relie LM du diagramme d'états-transitions à sa structure de routage des jetons. Pour chaque transition FSM qui génère des événements, un arc de sortie sera créé à partir de la transition ONM associée à la transition FSM vers la place ED. Si l'événement est un événement externe, l'inscription de l'arc de sortie porte le symbole "ex". Les événements externes d'un objet sont spécifiés dans le diagramme de collaboration (la règle N°8). Sinon, l'événement sera considéré comme un événement interne (la règle N°9). D'autre part, le modèle LM est relié à la place IP de la structure de routage de la même manière que pour la place ED mais par des arcs d'entrée (les règles N°6 et N°7). Enfin, les transitions ITA et OTA sont reliés à la place ILP (les règle N°10 et N°11).

La dernière étape consiste à supprimer tous les éléments des diagrammes d'états-transitions, de diagramme de collaboration et le formalisme FSM afin d'avoir un seul modèle CPN qui représente le système (les règles N°15, N°16, N°17, N°18, N°19, N°20).

1.- Conv_FSMTransition2ONMTransition. Prio.1

Condition : (Node(3).Transformed = = 0)

LHS ::= RHS

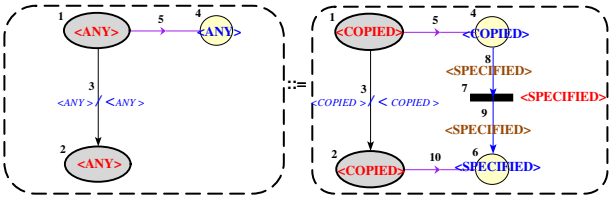


Action : Node(3). Transformed = 1

2.- Trsfr _FSMSteAndTransition2ONMPlaceAndTransition. Prio.2

Condition : (Node(3). Transformed = = 0)

LHS ::= RHS

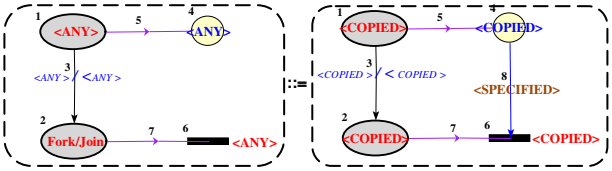


Action : Node(3). Transformed = 1

3.- Conv_InputTransitionOf_ForkOrJoinSte. Prio.3

Condition : (Node(3). Transformed = = 0)

LHS ::= RHS

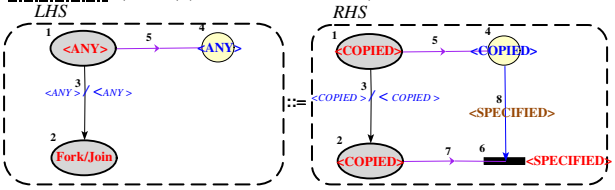


Action : Node(3). Transformed = 1

4.- Conv_ForkOrJoinSteAndFSMTransition. Prio.4

Condition : (Node(3). Transformed = = 0)

LHS ::= RHS

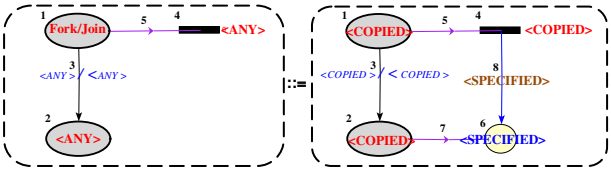


Action : Node(3). Transformed = 1

5.- Conv_FSMSteAndTransitrionFromForkOrJoinSte.Prio.5

Condition : (Node(3). Transformed = = 0)

LHS ::= RHS

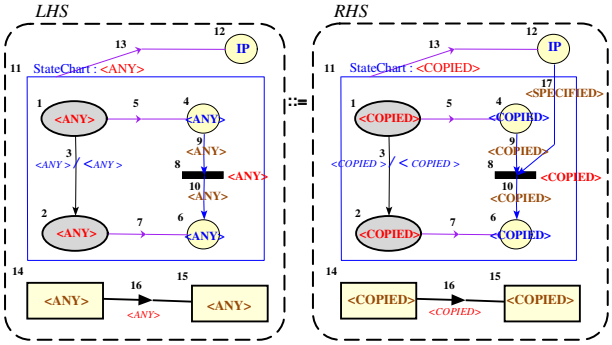


Action : Node(3). Transformed = 1

6.- Gen_InputArcForExternalEvent. Prio.6

Condition : ((Node (11).Name = = Node (14).Name) And (Node (3).action IN Node (16).events))

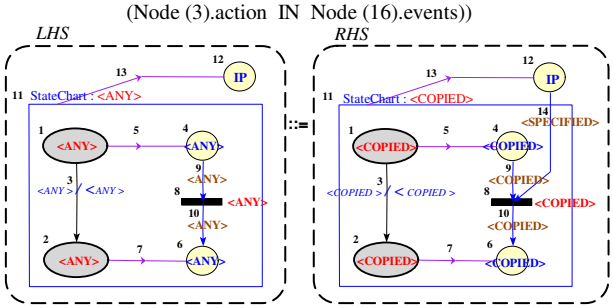
LHS ::= RHS



7.- Gen_InputArcForInternalEvent. Prio.7

Condition : ((Node (11).Name = = Node (14).Name) And (Node (3).action IN Node (16).events))

LHS ::= RHS

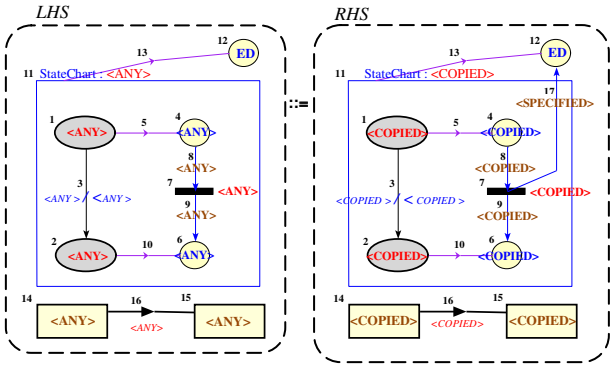


Action :

8.- Gen_OutputArcForExternalEvent. Prio.8

Condition : ((Node (11).Name = = Node (14).Name) And (Node (3).action IN Node (16).events))

LHS ::= RHS

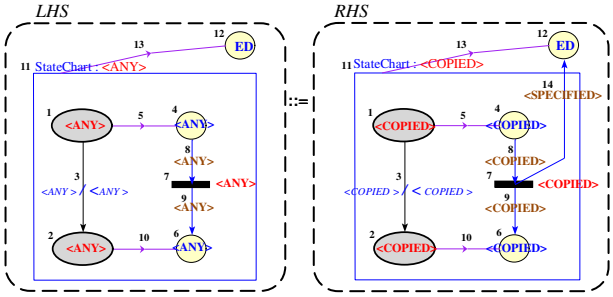


Action :

9.- Gen_OutputArcForInternalEvent. Prio.9

Condition : ((Node (11).Name = = Node (14).Name) And (Node (3).action IN Node (16).events))

LHS ::= RHS

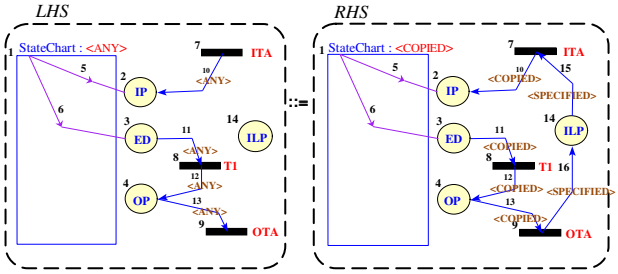


Action :

10.- Relate_ITA_And_OTATransition2ILPPlace. Prio.10

Condition :

LHS ::= RHS



Action :

Figure 6.7: 2ème GG, les Règles N°1-10

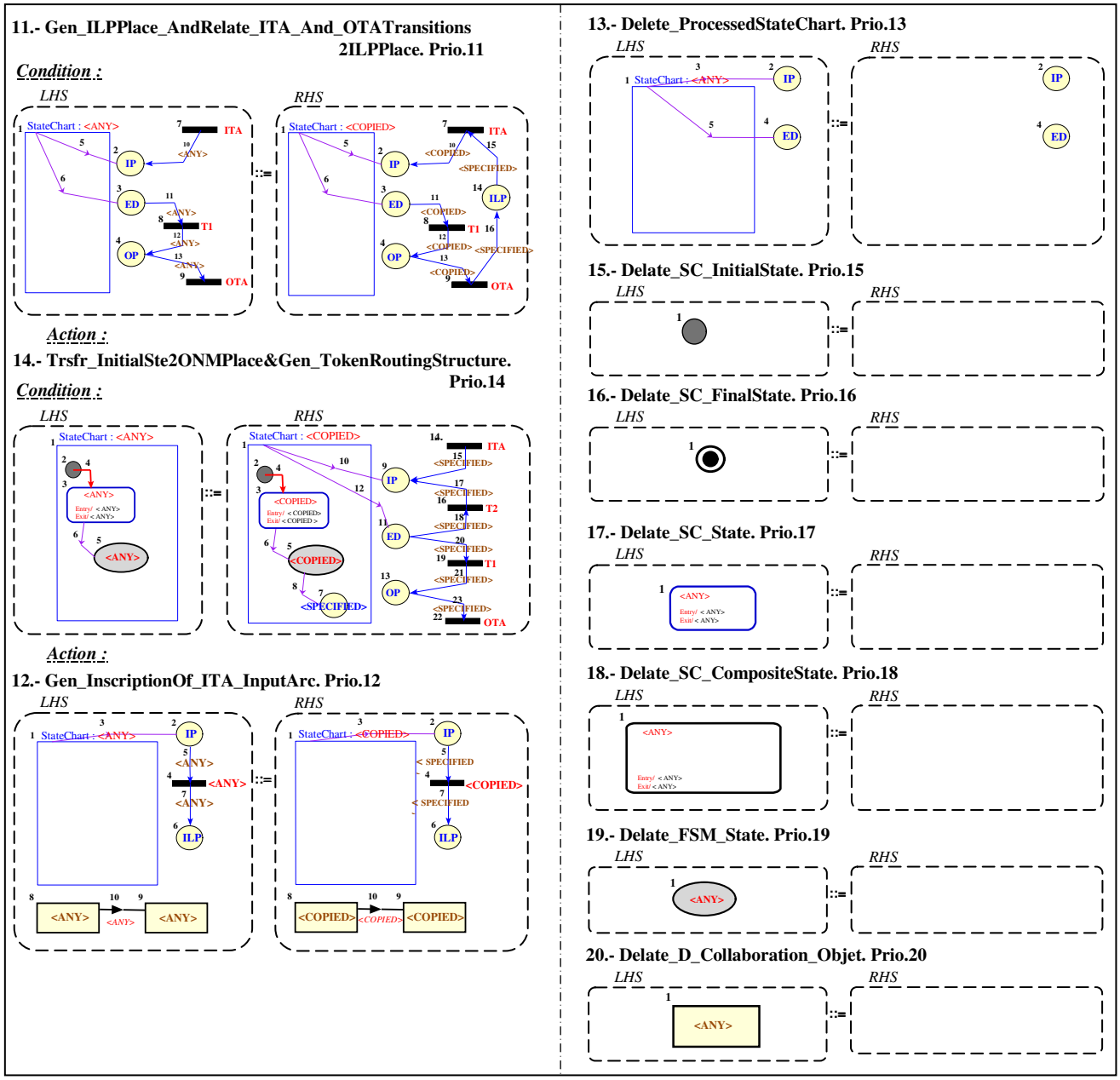


Figure 6.8: 2ème GG, les Règles N°11-20

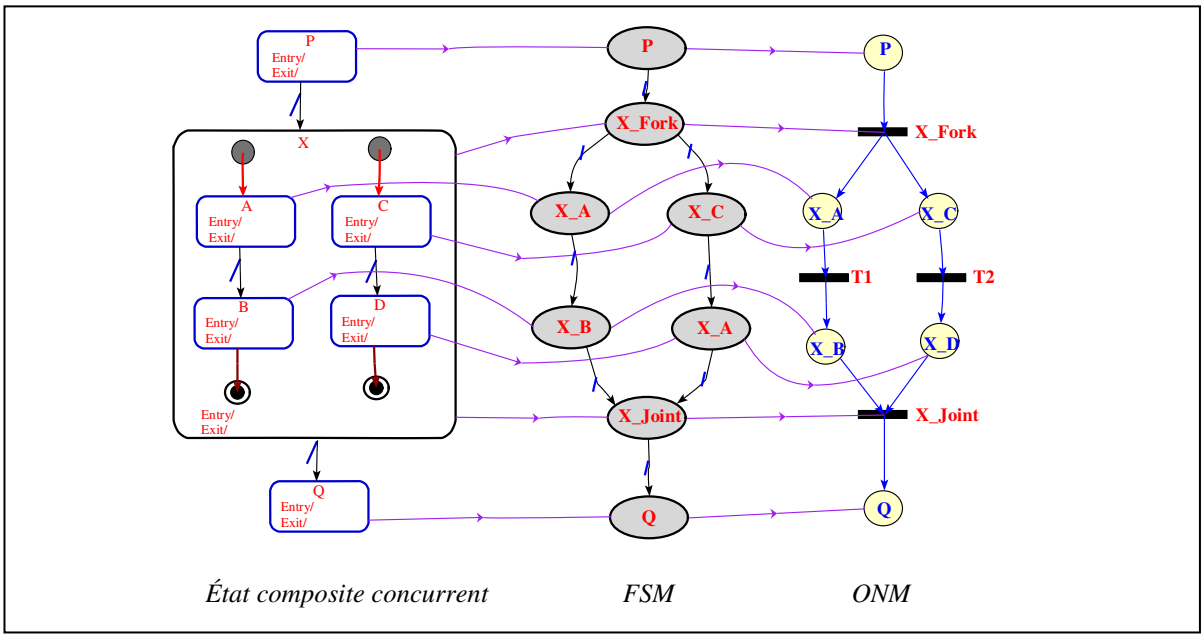


Figure 6.9: Transformation d'un état composite concurrent

6.3.2.3 3ème GG : Génération de la spécification INA

Afin d'analyser les modèles CPNs, il est nécessaire de les traduire en leurs spécifications INA équivalentes. Ceci est réalisé en définissant une grammaire de graphes de dix règles qui seront exécutées dans un ordre ascendant de leurs priorités [Kerkouche09d, Kerkouche10b] par le système de réécriture de graphes de l'outil AToM^3. Nous sommes concernés par la génération automatique de code. Par conséquent, aucune règle ne va changer le modèle CPN. L'application de cette grammaire à un modèle CPN conduit à la génération d'un fichier " *.cnt*" qui contient sa description INA équivalente. Le fichier " *.cnt*" est composé de la structure du réseau déplié (qui décrit les couleurs des sous-places, leurs marquages initiaux, et leurs liens avec les sous-transitions), et les informations de dépliage.

L'action initiale de notre grammaire de graphes consiste à créer un fichier " *.cnt*" qui contiendra la description INA équivalente et à décorer toutes les places et les transitions du modèle par des variables temporaires utilisées dans la spécification des conditions des règles. Pour chaque place, deux attributs sont ajoutés: "*current*" and "*visited*". L'attribut "*current*" est utilisé pour identifier la place dans le modèle pour laquelle le code est en cours de génération, alors que l'attribut "*visited*" est utilisé pour indiquer que le code pour la place est généré (*visited = = 1*: pour la structure de dépilage, *visited = = 2*: pour la section d'information des sous-places et *visited = = 3*: pour la section d'agrégation). Dans chaque transition, trois attributs sont également ajoutés: "*PRE*", "*POST*" et "*visited*". L'attribut "*PRE*" indique que la transition est traitée comme une transition d'entrée pour la place courante, alors que l'attribut "*POST*" indique que la transition est

traitée comme une transition de sortie. L'attribut "*visited*" est utilisé pour indiquer que le code pour la transition est généré (visited = = 1: pour la section d'information des sous-transitions, visited = = 2: pour la section d'agrégation). Tous ces attributs temporaires sont initialisés à 0.

Les principales étapes de la génération automatique de code peuvent être décrites comme suit:

La première étape consiste à sélectionner une place non encore traitée et lui associer un numéro. Pour les sous-places de la place sélectionnée (*current == 1*), la grammaire associe également un numéro pour chacune de ses sous-places (la règle N°6).

Dans la deuxième étape, la grammaire génère la structure du réseau déplié. Pour chaque sous-place dans la place courante, une ligne est générée dans le fichier " .cnt" qui contient, dans l'ordre suivant:

- le numéro de la sous-place et son marquage (la règle N°4).
- la liste des sous-transitions PRE (la règle N°1 plusieurs itérations).
- la liste des sous-transitions POST (la règle N°2 plusieurs itérations).

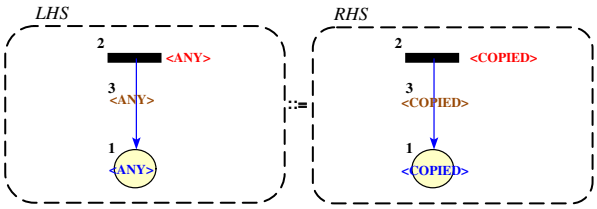
La règle N°3 est appliquée pour initialiser les attributs temporaires dans les transitions pour le traitement de la prochaine sous-place dans la place courante. La règle N°5, quant à elle, est utilisée pour marquer la place comme traité pour la génération de la structure du réseau dépilée "*visited =1*". Ce processus est répété pour toutes les places non traitées dans le modèle CPN (la règle N°6 pour la sélection d'une autre place "*visited =0*").

La dernière étape consiste à générer les informations de dépliage du modèle CPN. La règle N°7 (la règle N°8, respectivement) génère les numéros associés aux sous-places (sous-transitions, respectivement) ainsi que leurs couleurs qui représentent les différents types d'événements. Les règles N°9 et N°10 génèrent, à leurs tours, les informations d'agrégation pour les places et les transitions respectivement. La règle N°9 génère pour chaque place le numéro associé, le nom et la liste des numéros de ses sous-places, alors que la règle N°10 génère pour chaque transition le numéro associé, le nom et la liste des numéros de ses sous-transitions.

La grammaire possède également une action finale qui détruit les attributs temporaires des entités et ferme le fichier *" .cnt*".

1.- genListOfPreTransitions. Prio.1

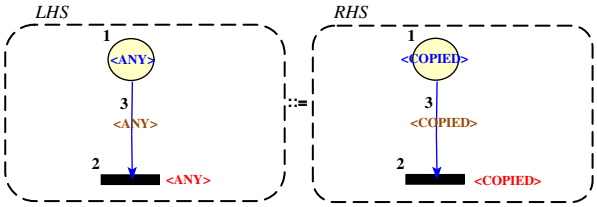
Condition : ((Node(1).current = = 1) and (Node(1).visited = = 0) and (Node(2).PRE = = 0))



Action : Node(2).PRE = 1

2.- genListOfPostTransitions. Prio.2

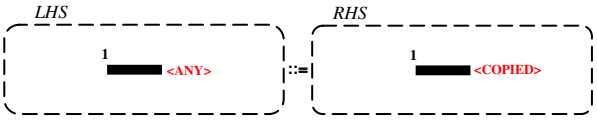
Condition : ((Node(1).current = = 1) and (Node(1).visited = = 0) and (Node(2).POST = = 0))



Action : Node(2).POST = 1

3.- InitialiseTransition. Prio.3

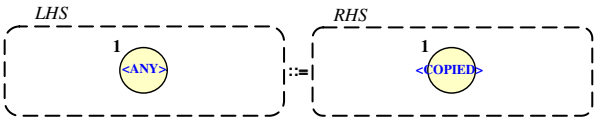
Condition : ((Node(1).PRE = = 1) or (Node(1).POST = = 0))



Action : Node(1).PRE = 1
Node(1).POST = 1

4.- SubPlaceDescription. Prio.4

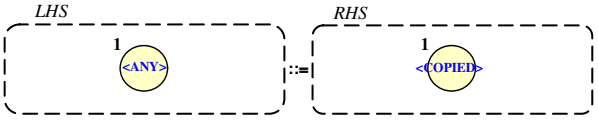
Condition : ((Node(1).current = = 1) and (Node(1).visited = = 0))



Action :

5.- EndOfPlaceDescription. Prio.5

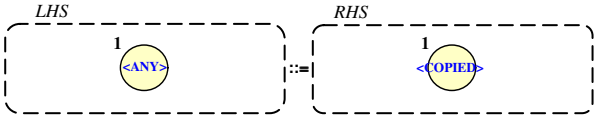
Condition : ((Node(1).current = = 1) and (Node(1).visited = = 0))



Action : Node(1).current = 0
Node(1).visited = 1

6.- SelectPlaceToDescribe.Prio.6

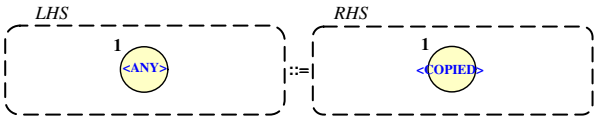
Condition : ((Node(1).current = = 0) and (Node(1).visited = = 0))



Action : Node(1).current = 0
Node(1).visited = 1

7.- genSubPlaceInformation. Prio.7

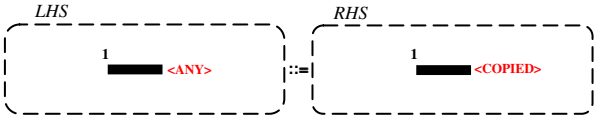
Condition : ((Node(1).current = = 0) and (Node(1).visited = = 1))



Action : Node(1).current = 0
Node(1).visited = 2

8.- genSubTransitionInformation . Prio.8

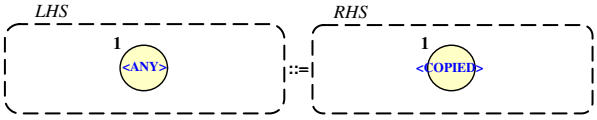
Condition : (Node(1).visited= = 0)



Action : Node(1).visited = 1

9.- genPlaceAggregation. Prio.9

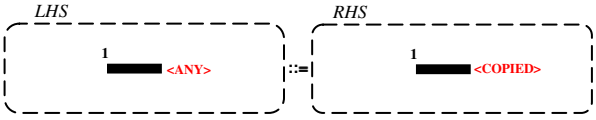
Condition : ((Node(1).current = = 0) and (Node(1).visited = = 2))



Action : Node(1).current = 0
Node(1).visited = 3

10.- genTransitionAggregation. Prio.10

Condition : (Node(1).visited= = 1)



Action : Node(1).visited = 2

Figure 6.10: 3ème GG : Génération de la spécification INA

6.4 Une Etude de cas: La Machine ATM

Afin de mettre en évidence notre approche, nous avons opté pour l'étude de cas d'une machine qui distribue des billets de banque ATM (Automated Teller Machine) pour montrer toutes les étapes de transformations en utilisant l'outil proposé.

La Figure 6.11 présente les diagrammes d'états-transitions et le diagramme de collaboration créés dans notre outil (en haut du canevas de l'outil).

Dans cet exemple, La machine ATM [Booch99] permet de délivrer des billets d'argent aux clients. Elle possède trois états de base: '*Idle*' (en attente d'interaction avec le client), '*Active*' (traitement d'une commande d'un client) et '*Maintenance*' (en rupture de billets par exemple). Lorsqu'elle est dans l'état '*active*', le comportement de l'ATM suit le chemin suivant: valider le client ('*Validate*'), sélectionner la transaction ('*Selecting*'), traiter la commande ('*Processing*') et imprimer un reçu ('*Printing*'). Après l'impression du reçu, la machine retourne à l'état de repos ('*Idle*'). Par ailleurs, le client peut à tout moment annuler la transaction et la machine retourne à l'état ('*Idle*').

Le diagramme d'états-transitions '*ATM_User*' montre les différentes actions de l'utilisateur. Tous les événements inscrits dans un diagramme d'états-transitions, qui ne sont pas représentés par le diagramme de collaboration, sont considérés comme des événements internes.

Afin d'analyser cette spécification du comportement de la machine ATM, nous devons transformer cette spécification dans son modèle CPN global. Pour réaliser cette transformation dans notre outil, nous avons exécuté la 1ère grammaire de graphes pour obtenir les modèles FSMs équivalents aux diagrammes d'états-transitions (voir la Figure 6.11 en bas du canevas de l'outil).

Ensuite, nous avons également exécuté la 2ème grammaire de graphes pour synthétiser le modèle CPN global à partir des modèles FSMs obtenus par la 1ère grammaire. Le modèle CPN global obtenu par la 2ème grammaire est présenté dans la Figure 6.12.

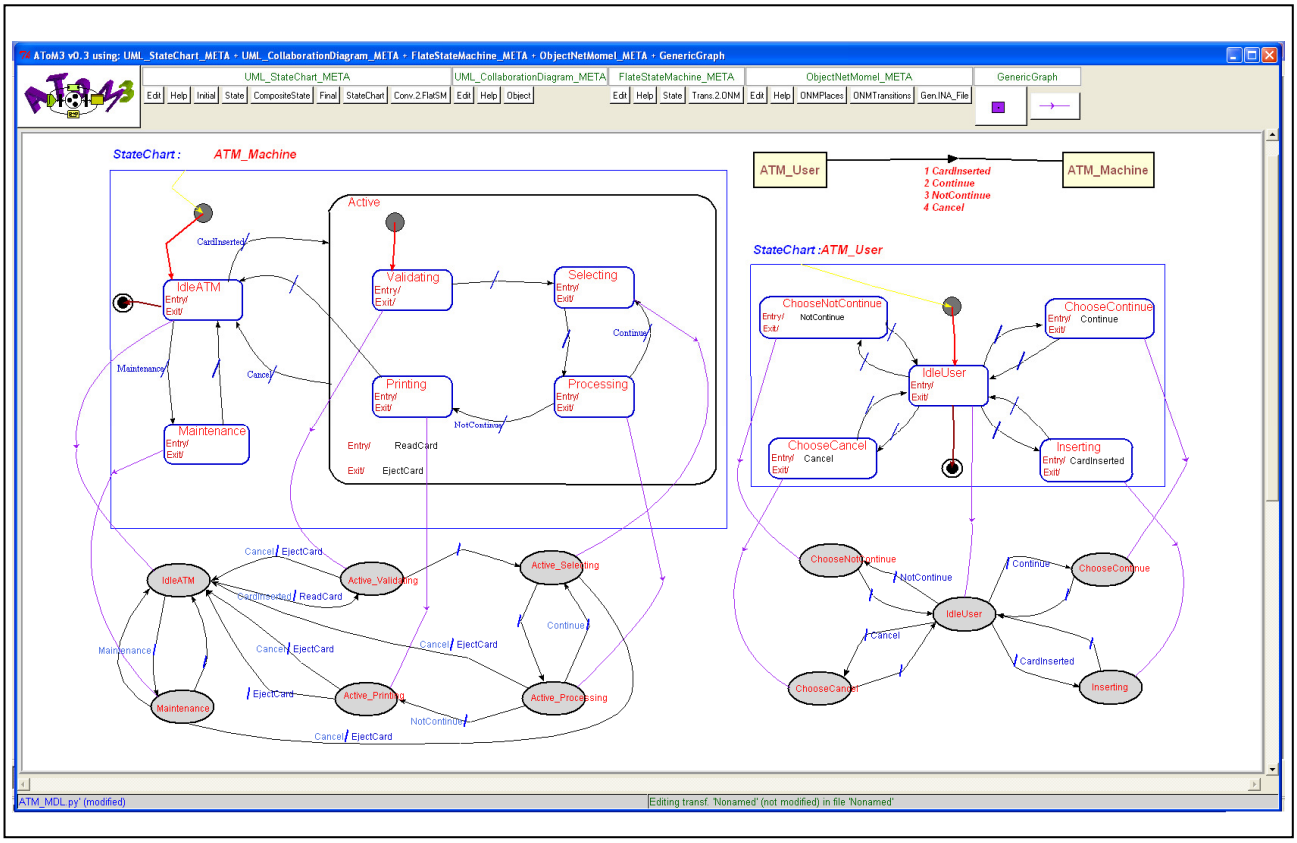


Figure 6.11: la modélisation UML de la machine ATM & les modèles FSM équivalents

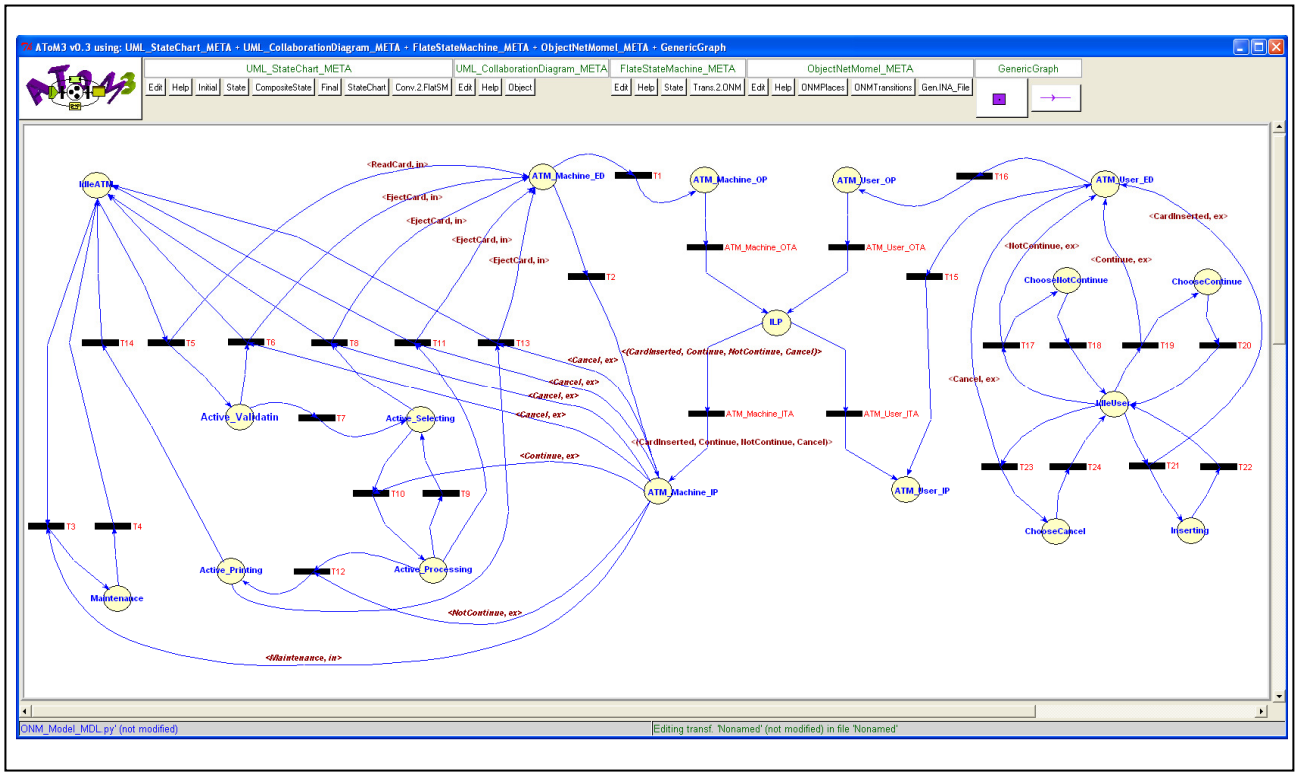


Figure 6.12: le modèle CPN global

Pour effectuer l'analyse du modèle CPN en utilisant l'analyseur INA, nous avons besoin de sa spécification INA équivalente. Pour générer la description INA dans notre outil, nous avons exécuté la 3ème grammaire de graphes. Le fichier " *ATM.cnt*" généré par la grammaire est présenté dans la Figure6.13.

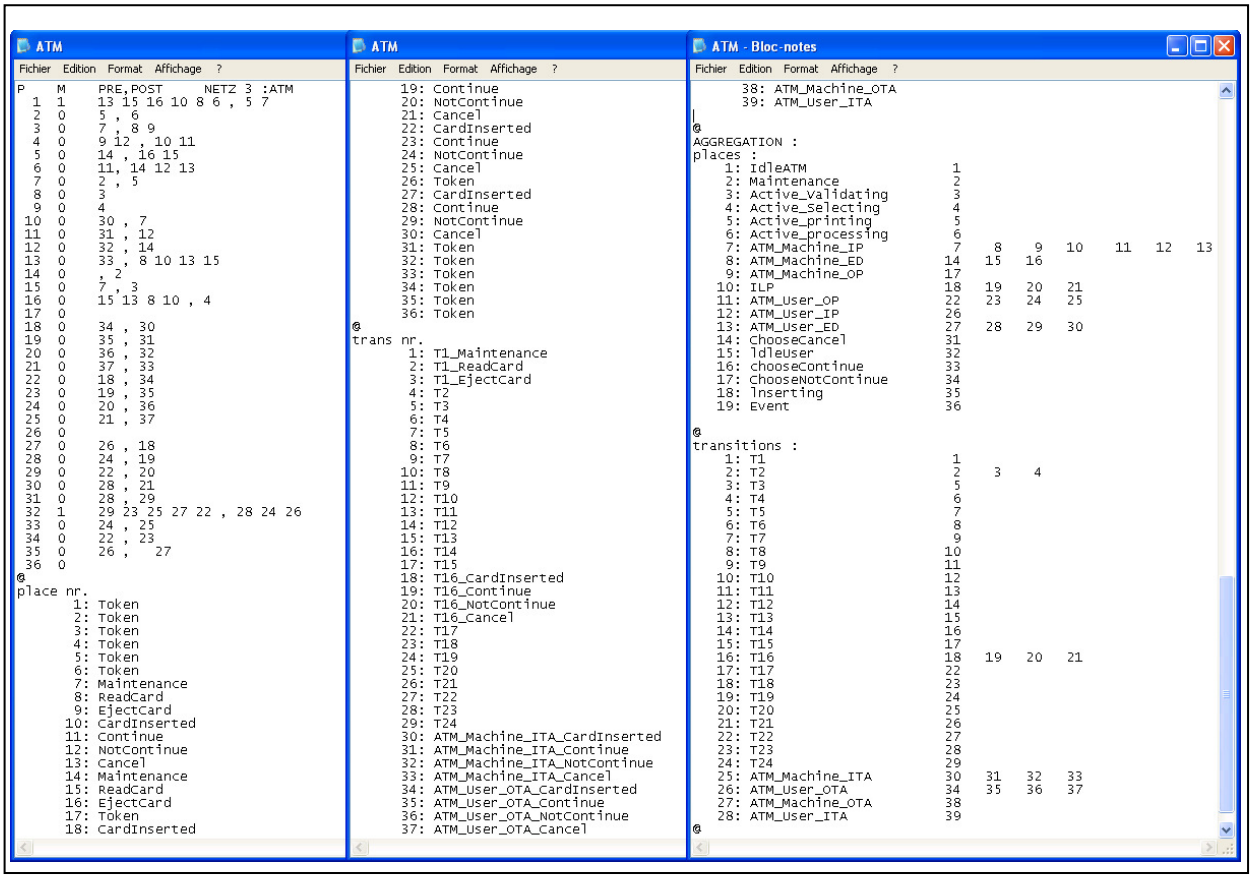


Figure 6.13: Spécification INA du modèle CPN global

Pour analyser les propriétés de la spécification du comportement de la machine ATM, nous avons appliqué l'analyseur INA sur le fichier généré ("*ATM.cnt*") et nous avons obtenu le résultat suivant (voir la Figure 6.14)

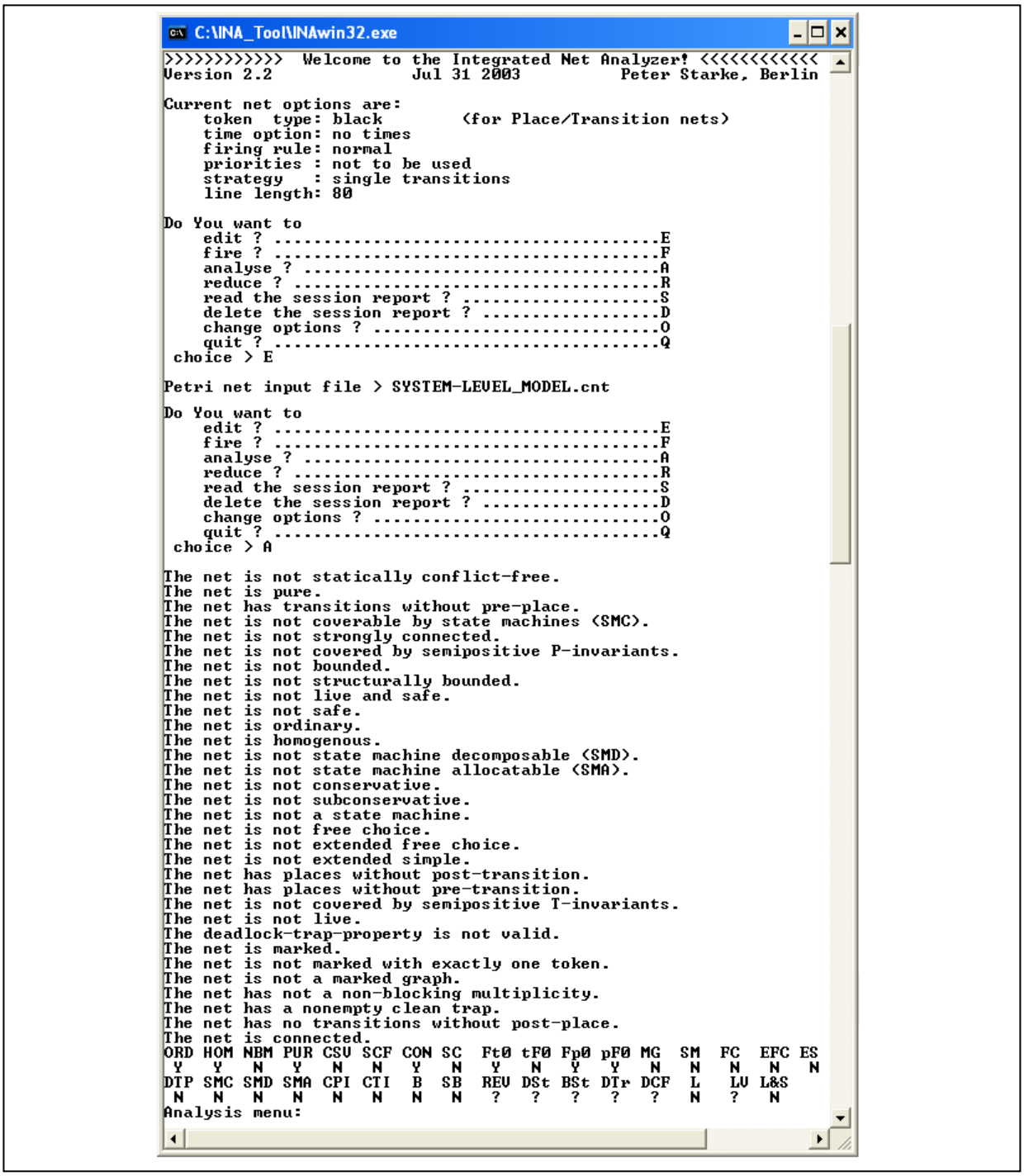


Figure 6.14: Résultat de l'application de l'analyser INA sur le fichier " ATM.cnt"

En analysant les résultats: "The net is not live and safe" et "The deadlock-trap-property is not valid", nous pouvons déduire que le modèle de la Machine ATM contient une situation d'inter-blocage.

6.5 Conclusion

Dans ce chapitre nous avons proposé une approche intégrée UML/CPN pour la modélisation et la vérification des modèles dynamiques des systèmes complexes. Plus précisément, nous avons proposé une approche automatisée et des outils permettant de transformer le comportement dynamique des systèmes exprimés en diagramme d'états-transitions et diagramme de collaboration vers leur équivalent en réseaux de Petri colorés. Cette transformation vise à automatiser et à assister l'utilisation conjointe des notations semi-formelles conviviales (les diagrammes UML) et les notations formelles analysables (les réseaux de Petri colorés). Pour analyser les propriétés dynamiques du système représenté par les modèles CPN, nous avons proposé d'utiliser l'analyseur INA. Notre approche est basée sur la méta-modélisation et transformation de graphes en utilisant l'outil AToM^3.

Dans un travail futur, nous envisageons de transformer d'autres diagrammes UML vers des réseaux de Petri colorés et d'utiliser les techniques de réduction des modèles CPN afin d'optimiser les modèles avant de procéder à l'analyse.

Conclusion générale

Dans cette thèse nous nous sommes intéressés à la modélisation et la vérification multi-paradigmes. La modélisation multi-paradigmes basée principalement sur les concepts de l'ingénierie dirigée par les modèles (IDM). L'approche de modélisation Multi-paradigmes cherche à apporter de nouvelles fonctionnalités aux différents modèles qui sont utilisés pour concevoir et analyser ces systèmes. Un des points clés dans cette approche est la possibilité de transformer les modèles, d'un espace de modélisation ou d'un niveau d'abstraction vers un autre. Son objectif est de faciliter l'utilisation conjointe de différents modèles qui sont utilisés pour représenter et/ou analyser les différentes propriétés ou aspects du système pendant le cycle de développement.

C'est dans ce contexte que se situe cette thèse. Le travail réalisé est divisé en trois parties :

- ☑ Dans la première partie, nous avons proposé un support outillé pour manipuler et simuler les ECATNets. notre outil basé sur la méta-modélisation et une grammaire de graphe qui assure la génération automatique des descriptions équivalentes en Maude.
- ☑ Dans la deuxième partie, nous avons proposé une plate-forme formelle et un outil graphique pour la spécification et l'analyse des systèmes logiciels complexes en utilisant les G-Nets. L'outil proposé permet d'exploiter les différents langages et techniques formels dans un cadre uni dont le but est de cumuler leurs avantages.
- ☑ Enfin, dans la dernière partie, nous avons défini notre approche de formalisation du langage UML par le biais des réseaux de Petri colorés. Nous avons développé un outil basé sur la méta-modélisation et les grammaires de graphes pour transformer le diagramme de statechart et le diagramme de collaboration en un modèle en réseaux de Perti colorés afin de vérifier les propriétés comportementales des systèmes. L'outil INA est alors utilisé pour l'analyse des propriétés du système modélisé

Dans un travail futur, nous comptons continuer la transformation des autres diagrammes UML (diagramme état/transition, diagramme de cas d'utilisation, diagramme d'activité, etc …) vers les réseaux de Petri en utilisant toujours la technique de transformation de graphes et l'outil ATOM^3.

Pour mieux valider nos approches, des cas réels et plus complexes seront pris en considération.

Nous planifions également de réaliser une implémentation de nos approches en utilisant d'autres outils de transformation de graphes tels que AGG et VIATRA2 dans un but de comparaison des performances.

Dans le cadre du rétro-ingénierie (reverse engineerig), une perspective intéressante de ce travail consisterait en l'obtention d'interprétations automatiques des résultats d'analyse des réseaux de Petri sur les modèles sources. Par exemple, comment interpréter un interblocage dans un modèle réseaux de Petri dans un modèle d'états-transitions?

Bibliographie

Bibliographie

[AFIS] AFIS: L'Ingénierie Système. http://www.afis.fr/praout/ingsys/ingsys.htm.

[AGG06] AGG home page. http ://tfs.cs.tu-berlin.de/agg, Last Consultation : October 2006.

[Andries99] 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.

[AToM3] AToM3 (2006). Home page: http://atom3.cs.mcgill.ca/

[Attiogbé07] J. Christian Attiogbé, "Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette", In HDR, Université de Nantes, Nantes Atlantique Université, 13 septembre 2007.

[Balasubramanian06] Daniel Balasubramanian, Anantha Narayanan, Chris vanBuskirk and Gabor Karsai, "The Graph Rewriting and Transformation Language: GReAT", In A. Zündorf and D. Varró, editors, Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006), volume 1 of ECEASST, 2006.

[Behrmann04] Gerd Behrmann, Alexandre David and Kim G. Larsen, " A tutorial on UPPAAL", In Proceedings on the International School on Formal Methods for the Design of Computer, Communication, and Software Systems, volume 3185 of Lecture Notes in Computer Science, pages 200-236, Bertinora, Italy, Springer-Verlag.2004.

[Benzaken91] Claude Benzaken, "Systèmes Formels: Introduction à la logique et à la théorie des langages", Editions Masson, 1991.

[Bernot91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre, "Software testing based on formal specifications: a theory and a tool", Software Engineering Journal, 6(6):387–405, ISSN 0268-6961, November 1991.

[Bert95] Didier Bert, Rachid Echahed, Paul Jacquet, Marie-Laure Potet and Jean-Claude Reynaud, "Spécification, généricité, prototypage : aspects du langage LPG", In Technique et Science Informatiques, 14(9): pages 1097-1129, Hermes 1995.

[Bettaz92] Mohamed Bettaz and Mourad Maouche, "How to specify Non Determinism and True Concurrency with Algebraic Term Nets", Lecture Notes in Computer Science, N 655, Spring Verlag, Berlin, pages 11-30, 1992.

[Bettaz93] Mohamed Bettaz, Mourad Maouche, Moussa Soualmi and Madani Boukebeche. "Protocol Specification Using ECATNets", Networking and Distributed Computing, pp 7-35, 1993.

[Bézivin 02] Jean Bézivin and Xavier Blanc, "Promesses et interrogations de l'approche MDA", Journal Développeur Référence – Septembre 2002, 2002.

[Bézivin 04] Jean Bézivin, "Sur les principes de base de l'ingénierie des modèles", RSTI-L 'Objet,. 10(4) :145–157, 2004.

[Booch99] Grady Booch, Jim Rumbaugh and Ivar Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[Bordbar 05] Behzad Bordbar and Kyriakos Anastasakis, " UML2ALLOY: A Tool for Lightweight Modelling of Discrete Event Sysytems ", IADIS International Conference in Applied Computing, V(1). pp.209-216. Algarve, Portugal, 2005.

[Boudiaf04] Noura Boudiaf and Alloua Chaoui, "Dynamic Analysis Algorithm for ECATNets", In Proceedings of MOCA'04, pages 47-63, ISSN 0105-8517, Daniel Moldt (Ed.), Aarhus, University, Denmark October 11-13, 2004.

[Boudiaf06] Noura Boudiaf, Kamel Barkaoui and Alloua Chaoui, "Implémentation des Règles de Réduction des ECATnets dans Maude", 6ème Conférence Francophone de MOdélisation et SIMulation - MOSIM'06, Rabat, Maroc, avril 2006.

[Burmester04] Sven Burmester, Holger Giese, Jorg Niere, Matthias Tichy, Jorg P. Wadsack, Robert Wagner, Lothar Wendehals and Albert Zundorf, "Tool Integration at the Meta-Model Level within the FUJABA Tool Suite", International Journal on Software Tools for Technology Transfer (STTT), vol. 6, n° 3, p. 203-218, 2004.

[Charles97] Nathan Charles, Howard Bowman and Simon J. Thompson, "From Act-one to Miranda, a Translation Experiment", In Computer Standards and Interfaces Journal, 19(1), May 1997.

[Clark01] Tony Clark, Andy Evans and Stuart Kent, "The metamodelling language calculus Foundation semantics for UML", In Proceedings of Heinrich Hussmann, editor, Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS

2001 Genova, Italy, April 2- 6. 2001 Proceedings, volume 2029 of LNCS, pages 17–31. Springer, 2001.

[Cliff86] Jones Cliff, "Systematic Software Development Using VDM", Prentice-Hall ISBN 978-0138807252, 1986.

[Czarnecki03] Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches", OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Czarnecki06] Krzysztof Czarnecki and Simon Helsen, "Feature-based survey of model transformation approaches", IBM Syst. J., 45(3):621–645, ISSN 0018-8670, 2006.

[Davis03] James Davis, "GME: the generic modeling environment", In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 82–83, New York, NY, USA, ACM, ISBN 1-58113-751-6, 2003.

[De Lara02] Juan De Lara and Hans Vangheluwe, "AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling", In Proceedings of Fundamental Approaches to Software Engineering, FASE'02, Vol. 2306. LNCS. Grenoble, France, pages 174-188, 2002.

[De Lara04] Juan De Lara and Hans Vangheluwe, "Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3", In Manuel Alfonseca, Software and Systems Modelling, Vol. 3, Springer-Verlag, Special Section on Graph Transformations and Visual Modeling Techniques, pages194-209,2004.

[Deng93] Yi Deng, Shi-Kuo Chang, Jorge C. A. de Figueired and Angelo Psrkusich, " Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems", In Proceedings of the 14th International Conference on Application and Theory of Petri Nets, Chicago, 206-223, 1993.

[Deng93] Yi Deng, S. K. Chang, Jorge C. A. de Figueired and Angelo Perkusich, "Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems", In Proceedings of the 14th International Conference on Application and Theory of Petri Nets, Chicago, 1993, 206-223.

[Ehrig05] Karsten Ehrig, Claudia Ermel and Stefan Hansen, "Towards Model transformation in Generated Eclipse Editor Plug-ins", Preliminary Version, GraMoT - International Workshop on Graph and Model Transformation, Institut f¨ur Softwaretechnik und Theoretische Informatik Technische Universit¨at Berlin, 2005.

[Emadi09a] Sima Emadi and Fereidoon Shams, "Transformation of Usecase and Sequence Diagrams to Petri Nets", In Proceedings of the ISECS International Colloquium on Computing, Communication, Control, and Management, 2009.

[Emadi09b] Sima Emadi and Fereidoon Shams ,"Mapping annotated use case and sequence Diagrams to a Petri Net Notation forPerformance Evaluation", In Proceedings of the Second International Conference on Computer and Electrical Engineering, 2009.

[Engels00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel and Stefan Sauer, "Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML", In Proceedings of the 3rd International Conference on the UML 2000, YORK, ROYAUMEUNI, Octobre 2000.

[Fairley85] Richard Fairley, "Software Engineering Concepts", MacGraw-Hill, New York, NJ. 1985.

[Favre 06] Jean-Marie Favre , Jacky Estublier and Mireille Blay-Fornarino, " L'Ingénierie Dirigée par les Modèles: au-delà du MDA", Traité IC2 – Information – Commande – Communication, Hermès – Lavoisier, 2006.

[Fernandes07] João M. Fernandes, Simon Jens, Bæk Jørgensen and Óscar Ribeiro, "Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net", In proceeding of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, IEEE Computer Society Press, Mai 2007.

[Fernandes07] Joao M. Fernandes, Simon Tjell, Jens Baek Jorgensen and Oscar Ribeiro, "Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net", Dep. Informática / CCTC, Universidade do Minho, Braga, Portugal, Dept. Computer Science, University of Aarhus, Aarhus, Denmark, 2007

[Fernandez92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodriguez and Joseph Sifakis, "A toolbox for the verification of program", In Proceedings of the International Conference on Software Engineering, ICSE'14, Melbourne, Australia, pages 246–259, May 1992.

[Fernandez96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier and Mihaela Sighireanu, "A protocol validation and verification toolbox", In Rajeev Alur and Thomas A. Henzinger, editors, Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA), volume 1102, pages 437–440, August 1996.

[Fleisch99] Wolfgang Fleisch, "Applying use cases for the requirements validation of component-based real-time software", In Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99), May 1999.

[Fröhlich00] Peter Fröhlich and Johannes Link, "Automated test case generation from dynamic models", In E. Bertino, editor, Proceedings of ECOOP 2000, volume 1850 of LNCS, pages 472–491. Springer, 2000.

[Garavel98] Hubert Garavel, "An open software architecture for verification, simulation and testing", In Proceedinf of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), volume 1384. Springer-Verlag, Lecture Notes in Computer Science, 1998.

[Gargantini09] Angelo Gargantini, Elvinia Riccobene and Patrizia Scandurra, " Integrating Formal Methods with Model-driven Engineering ", In Proceedings of the Fourth International Conference on Software Engineering Advances, Porto, Portugal , ISBN: 978-0-7695-3777-1, 2009.

[Genrich81] Hartmann J. Genrich and Kurt Lautenbach, "System Modelling with High-Level Petri Nets", Theoretical Computer Science, vol. 13, 1981.

[Genrich81] Hartmann J. Genrich and Kurt Lautenbach, "System Modelling with High-Level Petri Nets", In Proceedings of the Theoretical Computer Science, vol. 13, 1981.

[Gerber02] Anna Gerber, Michael Lawley , Kerry Raymond , Jim Steel and Andrew Wood, "Transformation: The Missing Link of MDA", In : Graph Transformation. Volume 2505 of Lecture Notes in Computer Science, Springer- Verlag (2002) 90-105 Proc. 1st International Conference. Graph Transformation, Barcelona, Spain 2002.

[Goguen96] Joseph A. Goguen and Malcolm Grant, "Algebraic Semantics of Imperative Programs", The MIT Press, ISBN 978-0262071727, May 22, 1996.

[Greenfield04] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent and John Crupi, " Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools", Wiley & Sons, 2004.

[Guangyu09] Guangyu Li and Shuzhen Yao, "Research on Mapping Algorithm of UML Sequence Diagrams to Object Petri Nets", In Proceedings of the 2009 WRI Global Congress on Intelligent Systems. V(4), pages 285 – 289,19-21 May 2009.

[Guerra03] Esther Guerra and Juan de Lara, "A Framework for the Verification of UML Models. Examples using Petri Nets", Escuela Politecnica Superior, Ingeniera Informatica, Universidad Autonoma de Madrid, 2003.

[Guttag85] John V. Guttag, James J. Horning and Jeannette M. Wing, "The Larch Family of Specification Languages", In Software, IEEE , ISSN 0740-7459 vol 2 . pp. 24-36. Sept. 1985.

[Harel 85] David Harel and Amir Pnueli, "Logics and models of concurrent systems", volume 13 of Nato Asi Series F: Computer And Systems Sciences, chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, ISBN 0-387-15181-8, New York, NY, USA, 1985.

[Heral87] David Heral, "Statechart: A Visual Formalism for Complex Systems", Science of Computer Programming. Vol.8, pp.231-274. 1987.

[Holland 98] John .H Holland, "Emergence: From Chaos to Order", Helix Books Addison-Wesley, 1998.

[Holscher06] Karsten Holscher, Paul Ziemann and Martin Gogolla, "On translating UML models into graph transformation systems", Journal of Visual Languages and Computing, pages 78-105, ELSEVIER, 2006.

[Holzman90] Gerard J. Holzman, "Design and Validation of Computer Protocols", Prentice Hall, 1990.

[Holzman91] Gerard J. Holzman, "Design and Validation of Computer Protocols", Prentice Hall, 1991.

[Holzmann97] Gerard J. Holzmann, "The model checker SPIN", IEEE Transactions on Software Engineering, 23(5):279–295, May 1997.

[Hu04] Zhaoxia Hu and Sol M. Shatz, "Mapping UML Diagrams to a Petri Net Notation for System Simulation", In Proceedings of Software Engineering and Knowledge Engineering (SEKE'04), Concurrent Software Systems Laboratory University of Illinois at Chicago, 2004.

[INA] INA home page : www2.informatik.hu-berlin.de/~starke/ina.html

[INCOSE] INCOSE: A Consensus of the INCOSE Fellows. http://www.incose.org/practice/fellowsconsensus.aspx.

[ISO 8402] ISO 8402 Qualité: Concepts et Terminologie. Partie 1: Termes génériques et Définitions.

[ISO85] ISO. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO/ DP 8807, March 1985.

[Jensen97] Kurt Jensen, "A Brief Introduction to Coloured Petri Nets", Lecture Notes in Computer Science, No 1217, Springer-Verlag, 1997, pp 203-208.

[Jensen98] Kurt Jensen, "An Introduction to the Practical Use of Coloured Petri Nets", Lecture Notes in Computer Science, No 1492, Springer-Verlag, pp 237-292, 1998.

[Jouault06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev and Patrick Valduriez, " ATL: a QVT like transformation language", In Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA, pages 719– 720, 2006.

[Karsai04] Gabor Karsai and Aditya Agrawal, "Graph Transformations in OMG's Model-Driven Architecture", Lecture Notes in Computer Science, Vol 3062, 243-259, Springer Berlin /Heidelberg, juillet 2004.

[Kent02] Stuart Kent, "Model driven engineering", In Butler, Michael J., Luigia Petre, and Kaisa Sere (editors): Third International Conference on Integrated Formal Methods (IFM), volume 2335 of Lecture Notes in Computer Science, pages 286–298, Turku, Finland, Springer, May 2002.

[Kerkouche08] Elhillali Kerkouche and Alloua Chaoui, "On the use of Meta-Modelling and Graph Grammars to process and simulate ECATNets model", In proceedings of MS'2008, Port Said, EGYPT, April 8-10, 2008.

[Kerkouche09a] Elhillali Kerkouche and Alloua Chaoui. "A Graphical Tool Support to Process and Simulate ECATNets Models Based on Meta-Modelling and Graph Grammars", INFOCOMP Journal of Computer Science, Vol. 8, N° 4, pp. 37-44, 2009.

[Kerkouche09b] Elhillali Kerkouche and Alloua Chaoui, "A Formal Framework and a Tool for the Specification and Analysis of G-Nets Models Based on Graph Transformation", In Proceedings of International Conference on Distributed Computing and Networking ICDCN'09, LNCS, Vol. 5408, Springer-Verlag Berlin Heidelberg, India, 2009, pages 206–211, 2009.

[Kerkouche09c] Elhillali Kerkouche, Allaoua Chaoui and Khaled Khalfaoui, "Transforming UML models to colored petri nets models using graph grammars", In proceedings of ISCC 2009: pages 230-236, , Sousse, Tunisia, July 5 - 8, 2009

[Kerkouche09d] Elhillali Kerkouche, Allaoua Chaoui, El Bay Bourennane and Ouassila Labbani, "Modelling and verification of Dynamic behaviour in UML models, a graph transformation based approach", In proceedings of SEDE'2009, Las Vegas, Nevada, USA, 22-24 June 2009.

[Kerkouche10a] Elhillali Kerkouche, Allaoua Chaoui, El Bay Bourennane and Ouassila Labbani, "A UML and Colored Petri Nets Integrated Modeling and Analysis Approach using Graph Transformation", Journal of Object Technology, Vol 9, No. 4, pp. 25-43, July 2010.

[Kerkouche10b] Elhillali Kerkouche, Allaoua Chaoui, El Bay Bourennane and Ouassila Labbani, "On the Use of Graph Transformation in the Modeling and Verification of Dynamic Behavior in UML Models", Journal of Software, Vol 5, No 11, pp 1279-1291, Nov 2010.

[King00] Peter King and Rob Pooley, "Derivation of Petri Net Performance Models from UML Specifications of Communications Software", In Haverkort, Bohnenkamp and Smith Eds, Computer Performance Evaluation - Tools 2000, Proceedings of the 11th International Conference on Tools and Techniques for Computer Performance Evaluation, Illinois, pages 262-276, Mars 2000.

[King99] Peter King and Rob Pooley, "Using UML to derive stochastic Petri net models", UKPEW '99, University of Bristol, UK Performance Engineering Workshop, Juillet 1999.

[Knöpfel 05] Andreas Knöpfel, Bernhard Gröne and Peter Tabeling, "Tabeling, Fundamental Modeling Concepts", Wiley, West Sussex UK, pages1- 321,2005.

[kuske02] Sabin kuske, Martin Gogolla, Ralf Kollman and Hans-Jorg Krewoski, "An integrated sementics for UML Class, Object and state diagrams based on graph transformation", In Proceedings of Third International Conference on Integrated Formal Methods (IFM'02), Lecture Notes in Computer Science, M. Butler, K. Sere (Eds.), vol. 2335, pages 11-28, Springer, Berlin, 2002.

[Kwon00] Gihwon Kwon, "Rewrite rules and operational semantics for model checking UML statecharts", In Andy Evans, Stuart Kent, and Bran Selic, editors, UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings, volume 1939 of LNCS, pages 528–540. Springer, 2000.

[Latella99] Diego Latella, Istvan Majzik and Mieke Massink, "Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker", Formal Aspects of Computing, 11:637–664, 1999.

[Lavagno03] Luciano Lavagno, Grant Martin and Bran V. Selic, "UML for real: design of embedded real-time systems", Kluwer Academic Publishers, Norwell, MA, USA, 2003, ISBN 1-4020-7501-4.

[Lédeczi01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle and Gábor Karsai, "Composing Domain-Specific Design Environments", IEEE Computer, vol. 34 no. 11, pp. 44-51, November 2001.

[Lilius99] Johan Lilius and Ivan Porres Paltor, "Formalising UML state machines for model checking", In Robert France and Bernhard Rumpe, editors, UML'99 – The Unified Modeling Language. Beyond the Standard. Second International Conference ,Fort Collins, CO, USA, October 28-30. 1999, Proceedings, volume 1723 of LNCS, pages 430–445. Springer, 1999.

[Marvin 68] Marvin Minsky, "Matter, mind, and models", Semantic Information Processing, pages 425–432, 1968.

[McMillan92] Kenneth L McMillan, "The SMV system, symbolic model checking - an approach", Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[Meseguer00] José Meseguer, "Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems". In S. Smith and C.L. Talcott, editors, Formal Methods for Open Object-based Distributed Systems, (FMOODS'2000), p. 89-117. Kluwer, 2000.

[Meseguer92] José Meseguer, "Conditional Rewriting Logic as an Unified Model of concurrency", Theoretical Computer Science, 1992.

[Meseguer96] José Meseguer, "Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report", Seventh International Conference on Concurrency Theory (CONCUR'96), Volume 1119 of LNCS, Springer Verlag, p.331-372, 1996.

[Milner80] Robin Milner, "A calculus of communicating systems", in Lecture Notes in Computer Science, vol.92, Springer-Verlag, Berlin, 1980.

[Milner93] Robin Milner, "Elements of interaction: Turing Award Lecture", Communications of the ACM, 36(1):78–89, CODEN CACMA2. ISSN 0001-0782, January 1993.

[Murata89] Tadao Murata, "Petri nets: Properties, Analysis and Applications", In Proceedings of the IEEE,Vol. 77, No 4, April 1989.

[Offutt99] Jeff Offutt and Aynur Abdurazik, "Generating tests from UML specifications", In Robert France and Bernhard Rumpe, editors, UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins,CO, USA, October 28-30. 1999, Proceedings, volume 1723 of LNCS, pages 416–429. Springer, 1999.

[OMG03] Updated joint initial submission against the action semantics for UML RFP, available at http://cgi.omg.org/cgi-bin/doc?ad/00-08-03.

[OMG03a] "Object Management Group: OMG Unified Modeling Language pecification", Version 1.5, mars 2003.

[OMG04] "Object Management Group (OMG), Model Driven Architecture (MDA)", site Internet, http://www.omg.org/mda,2004.

[OMGb] OMG: MetaObject Facility (MOF), version 2.0. http://www.omg.org/mof/.

[OMGc] OMG: Common Warehouse Metamodel (CWM), version 1.1. http://www.omg.org/technology/documents/formal/cwm.htm.

[OMGd] OMG: Object Constraint Language (OCL), version 2.0. http://www.omg.org/ocl/.

[OMGe] OMG: *XML Metadata Interchange* (XMI), version 2.0. http://www.omg.org/xml/.

[Owre93] Sam Owre, John Rushby and Natarajan Shnkar, "The PVS Specification Languages", In http:// www.csl.sri.sri.com/pvs-language/ 1993.

[Paltor99] Ivan Paltor and Johan Lilius, "vUML: A tool for verifying UML models", In Robert J. Hall and Ernst Tyugu, editors, Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999.

[Paulo00] Paulo J. F. Carreira and Miguel E. F. Costa, "Automatically verifying an objectoriented specification of the steam-boiler system", In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000), pages 345–360. GMD, 2000.

[Petit99] Michaël Petit, "Formal requirements engineering of manufacturing systems: a multiformalism and component-based approach", Thèse de doctorat de l'Université de Namur, Belgique, Octobre 1999.

[PROD] PROD Tool, Helsinki University of Technology, Finland. Available at http://www.tcs.hut.fi/Software/prod/

[Python] Python home page: htpp://www.python.org

[Ranger08] Ulrike Ranger and ErhardWeinell, "The graph rewriting language and environment PROGRES", Applications of Graph Transformations with Industrial Relevance, LNCS, 5088, 2008.

[Roques 00] Pascal Roques and Franck Vallée, "UML en action : De l'analyse des besoins à la conception en Java", Eyrolles, 2000.

[Rozenberg99] Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", Vol.1. World Scientific, 1999.

[Saldhana01] John Anil Saldhana and Sol M. Shatz, "UML Diagrams to Object Petri Net Models : An Approach for Modeling and Analysis", In Proceedings of PDCS'2001, the 14 th International Conference on Parallel Systems, UML and Petri nets relations and distributed computing systems, Las Vegas, Nevada, April 2001.

[Schürr99] Andy Schürr, Andreas J. Winter and Albert Zündorf, "The PROGRES approach : language and environment", World Scientific Publishing Co., Inc., River Edge, NJ, USA, pages 487–550, 1999.

[Smith02] Graeme Smith, Florian Kammuller and Thomas Santen, "Encoding Object-Z in Isabelle/HOL", In la revue Lecture Notes in Computer Science, vol.2272, pp.82-99, 2002.

[Staines07] Tony Spiteri Staines, "Supporting UML Sequence Diagrams with a Processor Net Approach", Journal of Software, VOL. 2, NO. 2, AUGUST 2007.

[Staines08] T. Spiteri Staines, "Intuitive Mapping of UML 2 Activity Diagrams intoFundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets", In Proceedings of the 15thIEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems, Belfast, Northern Ireland, IEEE Xplore, pp.191 – 200, 2008.

[Störrle04a] Harald Storrle, "Semantics and Verification of Data Flow in UML2 Activities", In Mark Minas, Hrsg., Proc. Intl. Ws. on Visual Languages and Formal Methods (VLFM‟04), Seiten 38–52. IEEE Press, 2004.

[Störrle04b] Harald Störrle, "Semantics of Control-Flow in UML 2.0 Activities", In Proceedings of the IEEE Symposium on Visual Languages – Human Centric Computing – VLHCC'04, pages 235-242, September 26-29, 2004.

[Störrle04c] Harald Störrle, "Semantics of Expansion Nodes in UML 2.0 Activities", In Proceedings of the 2nd Nordic Workshop on UML, Modeling, Methods and Tools – NWUML'04. 2004.

[Störrle04d] Harald Störrle, "Semantics of Exceptions in UML 2.0 Activities", Technical Report 0403, Ludwig-Maximilians-Universität München, Institut für Informatik. 2004.

[Störrle05] Harald Störrle and Jan Hendrik Hausmann, "Towards a Formal Semantics of UML 2.0Activities", P. Liggesmeyer, K. Pohl, M. Goedicke Eds. Software Engineering, Essen, Germany, pages 117-128, 2005.

[Sztipanovits05] Janos Sztipanovits, Gabor Karsai, Csaba Biegl, Ted Bapty, Ákos Lédeczi and Amit Misra, "A MULTIGRAPH: architecture for model-integrated computing", In proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95), pp. 361-368, Southern Florida, USA, November 6-10, 1995.

[Taentzer03] Gabriele Taentzer, "AGG : A graph transformation environment for modeling and validation of software", In Proceedings of Applications of Graph Transformations with Industrial Relevance : Second International Workshop, AGTIVE 2003,LNCS 3062, pages 446_453, Charlottesville, VA, USA, September-October, 2003.

[Taentzer05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Gabriele Taentzer, Daniel Varro and Szilvia Varro-Gyapay, "Model Transformation by Graph Transformation : A Comparative Study", Model Transformations in Practice Workshop at MoDELS, Montego, 2005.

[Tebibel 07] Thouraya Bouabana-Tebibel, "Roles at the basis of UML validation", Journal of Computing and Information Technology", 15, N°2, pages. 171-183, 2007.

[Truong06] Ninh Thuan Truong, "Utilisation de B pour la vérification de spécifications UML et le développement formel orienté objets", Dans la thèse de doctorat en Informatique de l'Université Nancy2, au LORIA à Vandoeuvre, 2006.

[Turner07] Kenneth J. Turner, "Representing and analysing composed web services using CRESS", in Journal of network and computer applications, ISSN 1084-8045 vol. 30, n°2, pp. 541-562, 2007.

[Vangheluwe02] Hans Vangheluwe and Juan De Lara, "Meta-models are models too", In Proceedings of the 2002 Winter Simulation Conference, (pp. 597-605), Eds : E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 2002.

[Vangheluwe02] Hans Vangheluwe, Juan de Lara and Pieter J. Mosterman, "An Introduction to Multi-Paradigm Modelling and Simulation", Tutorial, In Proceedings AI Simulation and Planning AIS-2002. Pages 9-20. Lisbone. Avril 2002.

[Vangheluwe03] Hans Vangheluwe and Juan de Lara, "Computer Automated Multi-Paradigm Modelling: Meta-Modelling And Graph Transformation", Invited paper in 2003 Winter Simulation Conference (IEEE and Society for Modelling and Computer Simulation). New Orleans, Decembre 2003.

[Varró03] Dániel Varró and András Pataricza, "Vpm : A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml (the mathematics of metamodeling is metamodeling mathematics)", Software and System Modeling, 2(3) :187–210, 2003.

[Varró06] Daniel Varró, András Balogh and András Pataricza, "The viatra2 transformation framework - model transformation by graph transformation", Eclipse Modeling Symposium, 2006.

[Wing90] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods", *Computer*, vol. 23(9):8-23, 1990.