الجمهورية الجزائرية الديمقراطية الشعبية

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DR. YAHIA FARES DE MEDEA

FACULTE DES SCIENCES ET DE LA TECHNOLOGIE

Département du Génie Électrique
et de l'Informatique



Mémoire de Fin d'Études

Présenté pour l'obtention du Diplôme
de Master en Informatique

Option : Ingénierie des Systèmes et Technologie de Web

THEME

Vérification des exigences *SysML* sur les diagrammes d'activités

Proposé et dirigé par :

- Mr. Rahim Messaoud

Co-encadré par :

- Mr. Hammad Ahmed

Préparé et présenté par :

- Zemir Abdenour
- Zerarka Abderrahmane

Remerciements

Avant tout, nous remercions ALLAH tout puissant, pour la force, le courage et la patience qu'il nous a donné durant toute les années d'étude, et la volonté d'arriver au terme de ce travail.

Nous remercions infiniment nos Parents pour tout ce qu'ils ont fait et ce qu'ils font encore pour nous.

Merci à monsieur Rahim Messaoud, maître assistant à la Faculté des Sciences et technologie de l'université de médea (YAHIA FARES), pour avoir accepté de nous encadrer. On le remercie pour leur aide inestimable, leur patience et leurs encouragements tout au long de ce travail. Il nous a donnée l'opportunité de travailler sur un sujet passionnant. Ses conseils ont été d'un apport considérable dans un souci de mieux s'investir dans le fond du sujet.

Nous remercions les membres du jury d'avoir accepter de juger notre travail.

Merci à tous les membres de notre famille et à nos amis et collègues pour leur soutien durant toute les années d'étude.

En fin, nous tenons à remercier tous ceux qui ont contribué d'une façon ou d'une autre à la réalisation de ce travail.

ملخص

إن التحقق من الأنظمة المعقدة أمر مطلوب جدا، خاصة إذا كانت هذه الأنظمة حاسمة. كما أنه يجب أن يكون هذا التحقق في المراحل الأولى من التطوير، أي في مرحلة التصميم ، والهدف من ذلك هو اكتشاف وتصحيح الأخطاء التي يمكن أن تسبب مشاكل وحتى وقوع أضرار بالغة. يمثل اختيار طريقة التحقق تحديا كبيرا لأنه لكل طريقة مزايا و عيوب، والطريقة المنهجية تعتبر فعالة جدا لأنها تستند إلى المفاهيم الرياضية وتعطي نتائج جيدة.

يقترح هذا العمل طريقة للتحقق من المتطلبات بلغة نمذجة الأنظمة (سيسمال) على مخطط الأنشطة. حيث يقوم هذا العمل على تحويل مخطط الأنشطة إلى شبكات بيتري و تحويل متطلبات سيسمال إلى الصيغ المنطقية سي تي ال، للسماح بالتحقق من خلال استغلال أدوات التحليل والتحقق.

الكلمات المفتاحية

مخطط الأنظمة، مخطط المتطلبات، لغة نمذجة الأنظمة، شبكات بتري، التحقق، هندسة الأنظمة.

SysML, Activity Diagram, Requirements, Verification, CPNtools, EMF, Petri Nets, ActRL, IDM , ATL, Xtext, CTL, ASK-CTL.

Résumé

La vérification des systèmes complexes est indispensable, lorsque ces derniers sont critiques. Cette vérification doit être dès les premières phases de développement c-à-d au moment de la conception, l'objectif est de pouvoir détecter et corriger des erreurs qui peuvent causer des problèmes et même des dégâts catastrophiques. Le choix de méthode de vérification est un défi, car chaque approche a des avantages et des inconvénients, les approches formelles sont des approches efficaces car elles se basent sur des concepts mathématiques et donnent des résultats pertinents.

Ce travail propose une démarche pour la vérification des exigences $SysML$ sur les diagrammes d'activités. Il se base sur la transformation des diagrammes d'activités en réseaux de Petri et la translation des exigences $SysML$ en formules de logique CTL, pour permettre la vérification en exploitant des outils d'analyse et de vérification.

Mots clés :

SysML, Diagramme d'activités, Exigences, Vérification, CPNTools, EMF, RdP, ActRL, IDM , ATL, Xtext, CTL, ASK-CTL.

Abstract

Verification of complex systems is indispensable, when they are critical. This verification must be in the early stages of development ie to the moment of conception, the goal is to detect and correct errors that can cause problems and even catastrophic damage. The choice of verification method is a challenge because each approach has advantages and disadvantages, formal approaches are effective approaches because they are based on mathematical concepts and give relevant results.

This work proposes an approach for the verification of *SysML* requirements on activity diagrams. It is based on the transformation of activity diagrams to Petri nets and the translation of SysML requirements into logical formulas *CTL*, to allow verification by exploiting analysis and verification tools.

Key words :

SysML, Activity Diagram, Requirements, Verification, CPNTools, EMF, Petri Nets, ActRL, IDM , ATL, Xtext, CTL, ASK-CTL.

Table des matières

Remerciements **i**

Résumé **ii**

Introduction générale **1**

I Langage *SysML* **3**
I.1 Introduction . . . 3
I.2 Ingénierie Système . . . 3
I.3 Apports de SysML par rapport à UML . . . 4
I.3.1 SysML, un profil UML . . . 4
I.3.2 Présentation SysML . . . 5
I.4 Les diagrammes de *SysML* . . . 5
I.4.1 Modélisation structurelle . . . 6
I.4.2 Modélisation dynamique . . . 8
I.4.3 Modélisation des exigences . . . 13
I.5 Outils SysML . . . 15
I.6 Conclusion . . . 15

II Réseau de Petri pour la vérification formelle **16**
II.1 Introduction . . . 16
II.2 Les Réseaux de Petri . . . 16
II.2.1 Définitions formelles . . . 17
II.2.2 Marquage initial . . . 17
II.2.3 Représentation Matricielle . . . 18
II.2.4 Dynamique des *RdP* . . . 18
II.2.5 Propriétés comportementales . . . 20
II.2.6 Extensions des *RdP* . . . 21
II.2.7 Outils d'analyse des RdP . . . 25
II.3 Vérification . . . 26
II.3.1 Approches de vérification formelle . . . 27
II.3.2 Les propriétés à vérifier . . . 29
II.3.3 Spécification des propriétés . . . 29
II.3.4 Vérification par *CPNTools* . . . 33
II.4 Conclusion . . . 35

III Ingénierie Dirigée par les Modèles **36**
III.1 Introduction . . . 36
III.2 L'Ingénierie Dirigée par les Modèles . . . 36
III.2.1 La Modélisation . . . 37
III.2.2 La Méta-Modélisation . . . 37
III.3 L'Architecture Dirigée par les Modèles (MDA) . . . 38
III.3.1 Transformation et MDA . . . 39
III.3.2 Standards de l'OMG . . . 40
III.4 La transformation de modèles . . . 40
III.4.1 Techniques de transformations . . . 41
III.4.2 Types de transformation . . . 42
III.5 Language de transformation ATL . . . 43
III.5.1 La plate-forme de modélisation EMF . . . 43
III.5.2 Atlas Transformation Langage . . . 43
III.5.3 Programmation d'un module ATL . . . 44
III.6 Conclusion . . . 47

IV Méthodologie **48**
IV.1 Introduction . . . 48
IV.2 Besoin de formalisation des spécification SysML . . . 48
IV.3 Démarche générale . . . 49
IV.4 Conclusion . . . 50

V Langage ActRL pour l'expression des exigences SysML **51**
V.1 Introduction . . . 51
V.2 Langage ActRL . . . 51
V.2.1 Grammaire du langage . . . 51
V.2.2 Expression Prédicat . . . 52
V.3 Édition des expressions ActRL . . . 53
V.3.1 Language spécifique au domaine (DSL) . . . 53
V.3.2 Vue d'ensemble de l'outil XText . . . 53
V.3.3 Grammaire ActRL avec XText . . . 54
V.4 Intégration des expressions Actrl au diagrammes d'exigences . . . 58
V.5 Conclusion . . . 59

VI Translation des spécifications SysML en spécifications formelle **60**
VI.1 Introduction . . . 60
VI.2 Transformation des DA vers les RdP . . . 60
VI.2.1 Méta-modèles . . . 61
VI.2.2 Processus de transformation . . . 67
VI.3 Translation des expressions ActRL vers la logique ASK-CTL . . . 77

VI.3.1 De ActRL vers CTL 77
VI.3.2 De CTL vers ASKCTL 78
VI.4 Vérification 80
VI.5 Conclusion 81

VIIÉtude de cas **82**
VII.1Introduction 82
VII.2Les spécifications SysML 82
VII.2.1 Diagramme d'activité 82
VII.2.2 Diagramme d'exigence 83
VII.3Transformation des spécifications SysML en langages formelles 85
VII.3.1 Transformation de DA en RdP 85
VII.3.2 Translation des exigences ActRL en ASK-CTL 87
VII.4Verification 87
VII.5Conclusion 88

Conclusion générale **89**

Bibliographie **91**

Annexe **95**

Table des figures

I.1 Les diagrammes de SYSML 5
I.2 Diagramme de définition de bloc et leur diagramme interne d'un véhicule . 6
I.3 Exemple du diagramme de paquetage 7
I.4 Diagramme paramétrique d'une véhicule 8
I.5 Exemple de diagramme de cas d'utilisation 8
I.6 Diagramme de séquence d'une véhicule 9
I.7 Exemple de diagramme d'état 10
I.8 Diagramme d'activité d'un véhicule 10
I.9 Un exemple de diagramme d'exigence 13
I.10 Diagramme des exigences d'un véhicule 14

II.1 Exemple de Réseau de Petri 17
II.2 Graphe de Marquage 19
II.3 Exemple d'un *RdP* coloré 23
II.4 Réseau de Petri Hiérarchique 25
II.5 *RdPC* modélisé par *CPNTools* 26
II.6 Principe de model-checking 28
II.7 Système de transitions et espace d'états 29
II.8 Abréviations CTL 32

III.1 Relations entre monde réel, modèle, méta-modèle 38
III.2 Les relations de bases dans l'IDM. 38
III.3 Le processus en Y de l'approche MDA 40
III.4 Les standards de l'Architecture Dirigée par les Modèles 41
III.5 Concepts de base de la transformation de modèles 41
III.6 Les types de transformation 42
III.7 Un aperçu d'un fichier Ecore ouvert avec EMF 43
III.8 La structure du langage ATL 44

IV.1 Exigence *SysML* édité avec Papyrus 49
IV.2 Démarche générale 50

V.1 Les patterns et scopes de Dwyer 52
V.2 Démarche générale de fonctionnement de l'outil xtext 54
V.3 La représentation graphique des règles $Model$ et $Actrl$ 55

V.4 La représentation graphique de la règle pattern 56
V.5 La représentation graphique de la règle scope 57
V.6 La représentation graphique des éléments des expressions de prédicat . . . 58
V.7 Un aperçus sur exemple de modèle dans l'éditeur ActRL 58
V.8 Un exemple d'intégration des exigence écrites par ActRL dans le DE 59

VI.1 Démarche générale de transformation de DA vers RdP 60
VI.2 La partie supérieure de méta-modèle AD 61
VI.3 Les flux d'activité(ActivityEdge) 62
VI.4 Les nœuds d'activity(ActivityNode) 62
VI.5 Les nœuds de contrôle(ControlNode) 63
VI.6 Les nœuds d'action et d'objet (ActionNode,ObjectNode) 63
VI.7 MMDA sous extension (Ecore) 64
VI.8 Le *MMPNML* pnmlcoremodel extrait de *PNML framwork* 65
VI.9 Les modifications effectué sur MMPNML 66
VI.10*MMPNML* sous sa extesion ECORE 67
VI.11Processus transformation ATL des DAs vers RDPs 67
VI.12Transformation de "Model" 68
VI.13Transformation de "opaque action" 69
VI.14Transformation de "Call behavior action" 70
VI.15Transformation de "Controlflow" 71
VI.16Transformation de "PetriNet" 74
VI.17Transformation de "Page" 74
VI.18Transformation de "Arc" 75
VI.19Transformation de "Place" 75
VI.20Transformation de "Transition" 76
VI.21Transformation de "RefPlace" 76
VI.22Transforamations des expression ActRL vers ASK-CTL 77
VI.23Vérification de modèle 80

VII.1L'activité principale $proteger_du_feu$ 83
VII.2L'activité appelé $detecter_le_feu$ 83
VII.3Le diagramme d'exigence pour le système de protection contre les incendies 84
VII.4L'expression des exigences avec l'editeur ActRL 84
VII.5Le DE après l'intégration des expressions extrait de l'editeur *ActRL* 85
VII.6Interface graphique de l'application de la vérification 86
VII.7Le réseau de Petri dérivée après la translation 86
VII.8Exemple d'intégration du fonction ML dans l'outil *CPNTools* 87
VII.9Un résultat d'évaluation par *CPNTools* 87

Liste des tableaux

I.4.1 Les éléments du diagramme d'activité conforme aux spécifications **OMG** . 11

VI.2.1 Table contient le reste des règles de transformations 71
VI.3.2 Translation des expressions *ActRL* en CTL extrait de [1] 77
VI.3.3 De ActRL vers CTL vers ASC-CTL 79

Liste des acronymes

UML	Unified Modeling Langage
SysML	System Modeling Langage
OMG	Object Management Groupe
BDD	Block Definition Diagram
IBD	Internal Block Diagram
BDD	Block Definition Diagram
DA	Diagramme d'Activité
DE	Diagramme d'Exigence
MMDA	Méta-modèle de diagramme d'Activité
RdP	Réseau de Petri
RdPC	Réseau de Petri coloré
RdPCH	Réseau de Petri coloré hiérarchique
PNML	Petri Net Markup Language
MMPNML	Méta-modèle PNML
IDM	Ingénierie Dirigée par les Modèles
OCL	Object Constraints Langage
XML	eXtensible Markup Language
XMI	XML Metadata Interchange
ATL	Atlas Transformation Langage
EMF	Eclipse Modeling Framework
TMF	Textual Modeling Framework
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
CPN	Colored Petri Net
DSL	Domain Specific Langage
EBNF	Extended Backus-Naur Form
ActRL	Activité Requirement Language
ML	Meta Langage
SML	standard ML

Introduction générale

Les systèmes complexes sont par leur nature difficiles à construire, et dans le but de garantir la sûreté et la fiabilité des systèmes critiques, il est primordial de maîtriser les différentes étapes du développement d'un système critique pour obtenir au final un système digne de confiance, c'est à dire satisfaisant les exigences qui lui sont associées. Pour vérifier qu'un système présente un comportement attendu, une première approche consiste à construire ce système, puis à observer son comportement face à différents stimuli (tests du système). Ce type d'approche présente plusieurs inconvénients majeurs, tout d'abord, il nécessite de disposer du système réel, et donc d'être en fin du cycle de développement du système. Par conséquent, en cas de test non conforme aux attentes, des travaux très coûteux sur le système peuvent être nécessaires. De plus, ce type d'approche ne permet pas d'obtenir de garantie absolue, il est très improbable de pouvoir observer leur impact sur le système par test, et par conséquent, cette approche de validation par tests s'avère insuffisante.

Une deuxième approche consiste à se baser sur les activité de validation et de vérification pour développer des systèmes critiques. La modélisation permet de séparer les différentes préoccupations qui interviennent dans le cycle de développement. La Vérification et la Validation ($V\&V$) sont les activités dans le processus de développement assurant que le système satisfait ses exigences (qu'il se comporte correctement par rapport aux attentes des utilisateurs et à sa spécification). La vérification concerne l'implémentation du système et sa spécification explicite tandis que la validation relie un système et les contraintes implicites de l'utilisateur final.

La vérification de ce type de systèmes consiste donc à garantir l'absence de risques de dysfonctionnement ou au moins limiter leur impact. Pour cela, de nombreuses méthodes de vérification et d'analyse ont été développées durant les dernières décennies. L'objectif était d'offrir des mécanismes pour assurer dès la phase de conception la fiabilité et la sûreté de fonctionnement du système. Ces techniques associent à la modélisation des méthodes d'analyse et de vérification pour valider les modèles du système à construire par rapport à ces exigences. Cependant, les exigences exprimées sur les modèles sont une partie très utile pour l'analyse, mais cette tâche est généralement très difficile à remplir à cause de manque des langages bien spécifiques pour l'expression des exigences des utilisateurs.

Dans ce contexte et à partir de ces méthodes d'analyse, nous proposons dans ce travail une démarche pour vérifier des spécifications SysML. En particulier, nous considérons la vérification des exigences SysML sur les diagrammes d'activité.

Ce travail sera organisé en sept chapitres :

- **Chapitre I :** Ce chapitre présente le langage $SysML$ et ses différents diagrammes. Particulièrement, nous focalisons sur la présentation du digramme d'activités et de digramme d'exigences.
- **Chapitre II :** Dans ce deuxième chapitre, nous présenterons le formalisme des réseaux de Pétri, leurs extensions et les techniques de vérification à base de ce formalisme. Ensuite, nous présenterons CPNtools, un outil de vérification pour les réseaux de Petri HCPN. Enfin nous décrierons la technique de vérification sur modèle, "$Model-Checking$", et les logiques temporelles qui permets la spécification des propriétés sur le modèle.
- **Chapitre III :** Ce chapitre présentera le domaine de l'Ingénierie Dirigée par les Modèles (IDM), et les différentes notions qui servent pour la transformation des modèles.
- **Chapitre IV :** Dans ce chapitre, nous décrirons notre méthodologie pour vérifier les exigences $SysML$ sur les diagrammes d'activités en spécifiant les différentes étapes pour arriver à la phase de vérification.
- **Chapitre V :** Dans ce chapitre, nous donnerons une description du langage $ActRL$, un langage proposé pour la formalisation des exigences $SysML$.
- **Chapitre VI :** Dans ce chapitre, nous proposons une transformation des diagrammes d'activité $SysML$ vers les réseaux de pétri colorées hiérarchiques. Ensuite, la translation des expression $ActRL$ en logique $ASK\text{-}CTL$. Finalement, nous décrirons notre processus de vérification en utilisant l'outil CPN-tools.
- **Chapitre VII :** Ce dernier chapitre illustre notre travail par un cas d'étude pratique consiste à valider notre travail.

Chapitre I

Langage *SysML*

I.1 Introduction

La modélisation a pour objectif de structurer les informations et les activités d'une organisation : données, traitements, et flux d'informations entre les entités. L'*UML* est l'un des langages utilisés pour modéliser, spécifier, visualiser, modifier et construire les documents nécessaires au bon développement d'un logiciel et offre un standard de modélisation pour représenter l'architecture logicielle.

En janvier 2001 l'initiative *SysML* est issue d'une décision du groupe de travail sur la modélisation des systèmes pilotés par les modèles de l'(INCOSE) [1], les avantages de *SysML* sur *UML* pour l'ingénierie système deviennent évidents et *SysML* se définit comme une extension d'un sous-ensemble d'*UML*. *SysML* est spécialisé dans la modélisation des systèmes et offre aux ingénieurs systèmes plusieurs améliorations notables par rapport à *UML*. Ce chapitre, se poursuit par un aperçu de chacun des diagrammes proposés par *SysML*, en commençant par la modélisation structurelle du système suivi par la modélisation dynamique, en concentrant sur le diagramme d'activité, puis les diagrammes des exigences.

I.2 Ingénierie Système

Les méthodes d'ingénierie système (IS) reposent sur des approches de modélisation, de simulation et de validation pour valider les spécifications des systèmes à concevoir. La modélisation a donc été couramment utilisée pour la décomposition fonctionnelle et structurelle du système. L'alternative consiste à effectuer une transition vers une approche orientée modèles (MBSE) [2], permettant de réaliser un modèle cohérent du système, stocké et géré dans un référentiel avec un outil informatique. Cette approche permet la réalisation d'un ensemble organisé de modèles, s'appliquant à différents niveaux de granularité tels que l'aspect opérationnel (contexte et utilisation du système), l'aspect fonctionnel (structure et sous-fonctions du système), et l'aspect physique (architecture). La modélisation permet de maîtriser la complexité du système étudié, car chaque modèle donne accès à une représentation abstraite de différents aspects du système.

1. International Council on Systems Engineering
2. model-based systems engineering

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.

L'ingénierie système est un ensemble de méthodes coopératives et interdisciplinaires qui englobent l'ensemble des activités adéquates pour concevoir, vérifier, développer, faire évoluer et vérifier un système qui peut être un ensemble de produits, processus et compétences humaines [2].

I.3 Apports de SysML par rapport à UML

La modélisation avec le langage *UML* est une pratique bien établie dans l'ingénierie logicielles. Bien que le langage *UML* permette par son caractère à usage général d'adresser de nombreux besoins pour l'IS [3], il a été nécessaire de l'adapter par la définition d'un profil spécifique à l'IS.

I.3.1 SysML, un profil UML

Le besoin de définir un langage basé sur *UML* pour l'IS a été introduit en 2001, par l'organisation internationale de l'ingénierie système INCOSE, qui s'est mise en relation avec l'OMG, organisme responsable d'*UML*. Dès lors, plusieurs membres de l'industrie (BAE, Motorola, Boeing...), éditeurs d'outils (IBM, Sparx Systems...), universités et organisations ont travaillé sur la définition du langage de modélisation système *SysML* car la sémantique de *SysML* est plus riche et flexible, elle impose moins de restrictions liées à la vision d'*UML* centrée sur le logiciel, et ajoute deux nouveaux types de diagrammes, le diagramme des exigences (requirements) peut être utilisé pour la gestion des exigences alors que le diagramme paramétrique peut être utilisé pour l'analyse des performances et l'analyse quantitative. Grâce à ces améliorations, *SysML* est capable de modéliser une large gamme de systèmes, incluant tant du matériel, que du logiciel, de l'information, des processus, du personnel, ou des équipements (au sens large).

SysML est un langage plus réduit qu'*UML*, ce qui facilite son apprentissage et son utilisation. *SysML* supprime beaucoup de concepts d'*UML* trop liés à sa vision centrée sur le logiciel. L'ensemble du langage *SysML* est plus petit, tant en nombre de types de dia-

3. ingénierie système

grammes qu'en nombre de concepts totaux. *SysML* gère mieux les notations tabulaires. Il fournit des tableaux d'allocations flexibles qui supportent l'allocation des exigences (requirements), l'allocation fonctionnelle, et l'allocation structurelle, ce qui facilite l'automatisation de la vérification et de la validation [3].

I.3.2 Présentation SysML

SysML est basé sur *UML*, et réutilise une partie d'*UML*2, et apporte également ses propres définitions (extensions *SysML*). Il prend en charge la spécification, l'analyse, la conception, la vérification et la validation pour le développement des systèmes. Il propose un ensemble de diagrammes adaptés à l'ingénierie système.

SysML offre l'avantage d'être accessible aux développeurs logiciels qui vont retrouver de nombreuses similitudes avec *UML*2. Il permet de produire des spécifications dans un langage unique pour des équipes hétérogènes, responsables de la réalisation des blocs matériels et logiciels. Les connaissances sont ainsi modélisées dans un référentiel unique qui améliore la communication entre les différentes équipes participantes, et permet la réutilisation des blocs réalisés.[4]

I.4 Les diagrammes de *SysML*

À partir des informations recueillies dans la spécification OMG d'*UML*/*SysML* [5], les diagrammes *SysML* se divise en deux catégories : les diagrammes de modélisation structurelle et des diagrammes de modélisation de comportement, c'est à dire dynamique. La figure I.1 suivante présente les neuf diagrammes proposés par *SysML* :

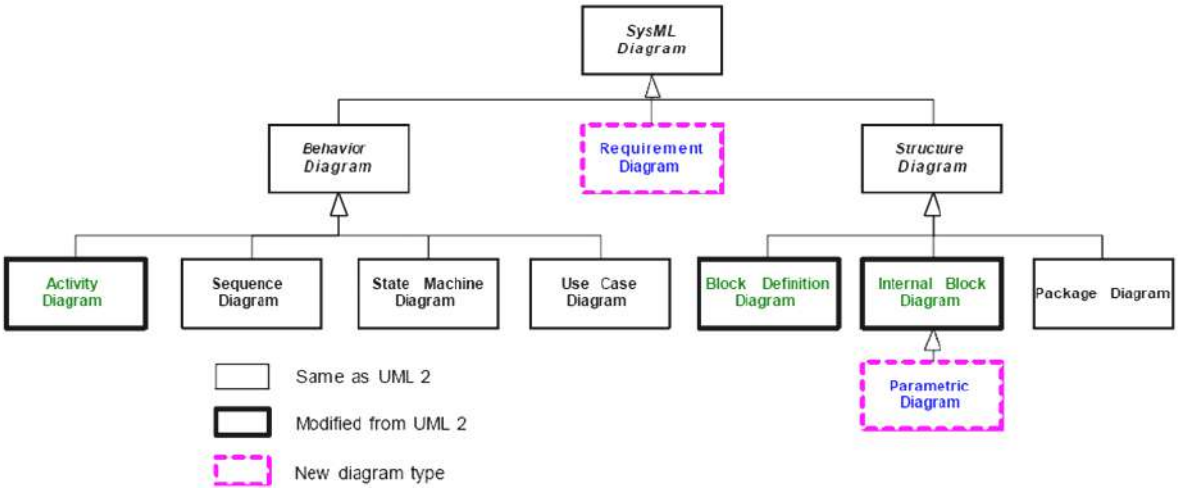


FIGURE I.1 – Les diagramme de SYSML [5]

I.4.1 Modélisation structurelle

La modélisation structurelle permet de décrire les caractéristiques statiques d'un modèle, elle consiste à modéliser les relations et les dépendances entres les différents constituants du système à concevoir.

I.4.1.1 Diagramme de définition de bloc

Le diagramme de définition de bloc (BDD) représente la vue boîte noire d'un bloc. Ainsi, le bloc principal et la hiérarchie des blocs qui le composent, qu'ils soient logiciels ou matériels, sont spécifiés dans ce diagramme.

Le BDD est similaire à la première page d'une notice de montage d'un meuble, indiquant la liste des éléments et des pièces à assembler avec leurs quantités respectives.

Par rapport à *UML*, le BDD de *SysML* redéfinit le diagramme de classe en remplaçant les classes par des blocs. La figure I.2 ci-dessous du diagramme de définition de bloc pour un véhicule.

I.4.1.2 Diagramme de bloc interne

Le diagramme de bloc interne (IBD) décrit la vue interne d'un bloc ou vue boîte blanche, et se base sur le BDD pour représenter l'assemblage final des blocs qui composent un bloc composite.

Les blocs composant le bloc principal et définit dans le BDD, ils sont instanciés en parties. Ces parties sont assemblées par des connecteurs, les reliant directement ou au travers de leurs ports (ports standards avec interfaces exposées et/ou ports de flux). L'exemple de véhicule illustré à la figure I.2 montre l'IBD du bloc véhicule dans le diagramme de définition des blocs.

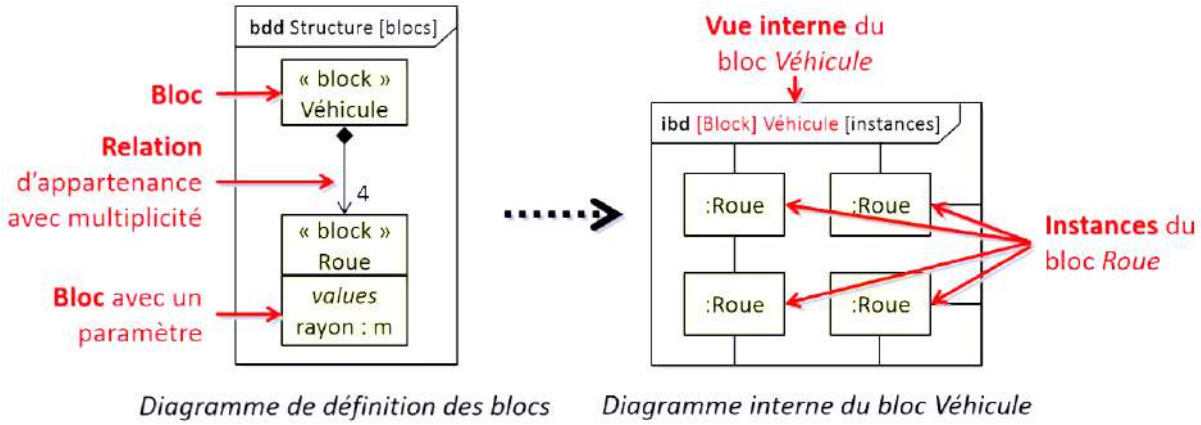


FIGURE I.2 – Diagramme de définition de bloc et leur diagramme interne d'un véhicule

Par rapport à *UML*2, l'IBD de *SysML* redéfinit le diagramme de structure composite en ajoutant entre autres les ports de flux. La figure I.2 présente deux diagrammes utilisés conjointement, le diagramme de définition des blocs et le diagramme interne d'un bloc. En *SysML*, les composants sont modélisés par des objets issus de classes.

Les objets sont des instances des blocs (modèles particularisés de composants), et le bloc est une classe de modèles de composants, du produit ou de son environnement, matériels et logiciels.

I.4.1.3 Diagramme de paquetage

Le diagramme de paquetage est la représentation graphique des relations existante entre les paquetages (ou espaces de noms) composant un système, et permet de regrouper sous une même appellation un ensemble d'éléments de modélisation tout comme avec *UML*. Un paquetage est susceptible de contenir n'importe quel élément de modélisation *SysML* et dans la pratique on utilise les paquetages pour regrouper au sein d'une même entité, un diagramme de cas d'utilisation, les diagrammes de séquence associés, le diagramme de classes et il apparaît alors comme un dossier dans une arborescence de fichiers. La figure I.3 présente un simple exemple de diagramme de paquetage.

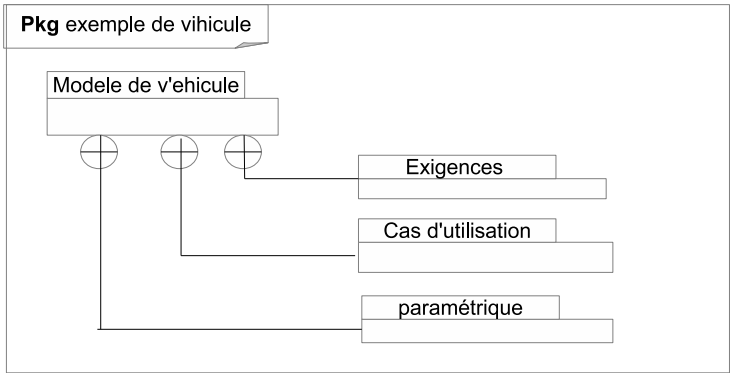


FIGURE I.3 – Exemple du diagramme de paquetage

Un paquetage est représenté par un dossier contenant un nom du paquetage peut être préfixé par le nom du paquetage qui le contient, et le contenu du paquetage peut être exposé. Les diagrammes de haut niveau représentent souvent le contenu d'un paquetage.

I.4.1.4 Diagramme paramétrique

Le diagramme paramétrique permet d'intégrer des analyses systèmes (performance, fiabilité, etc.) par des blocs de contraintes, ces derniers représentent une expression mathématique dont les paramètres peuvent faire référencer à des éléments du système, par exemple aux propriétés de blocs.

La figure I.4 montre le diagramme paramétrique basé sur l'exemple de véhicule précédent :

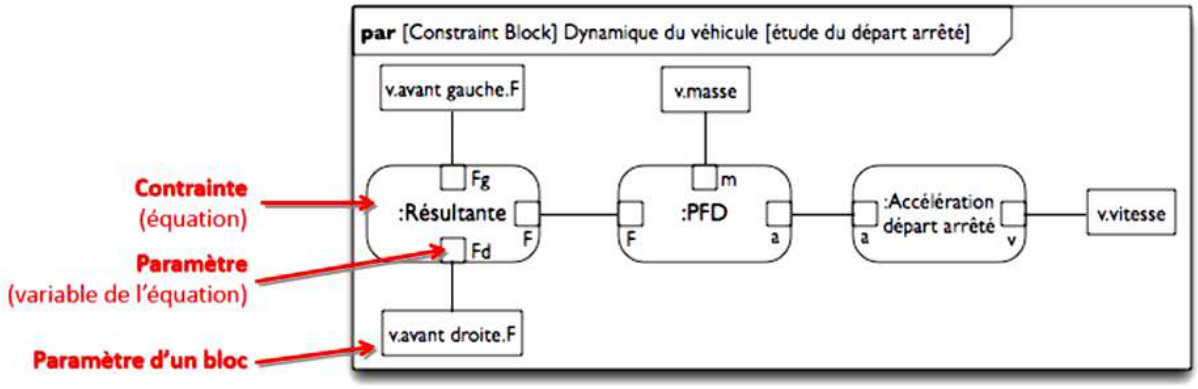


FIGURE I.4 – Diagramme paramétrique d'une véhicule

De façon similaire à la création du diagramme BDD, les blocs de contraintes sont définis dans un diagramme de classe.

I.4.2 Modélisation dynamique

La modélisation dynamique décrit comment les ressources modélisées dans les schémas structurels interagissent et comment ils exécutent les uns aux autres. Le schéma de comportement des ressources met en mouvement, contrairement à la vue structurel.

I.4.2.1 Diagramme de cas d'utilisations

Comme avec *UML*, on utilise les diagrammes de cas d'utilisations pour identifier les acteurs et les cas d'utilisations d'un point de vue d'utilisation du système, interactions acteurs/système. En *SysML*, les cas d'utilisations permettent de définir les frontières, le périmètre fonctionnel du système comme présenté dans la figure I.5.

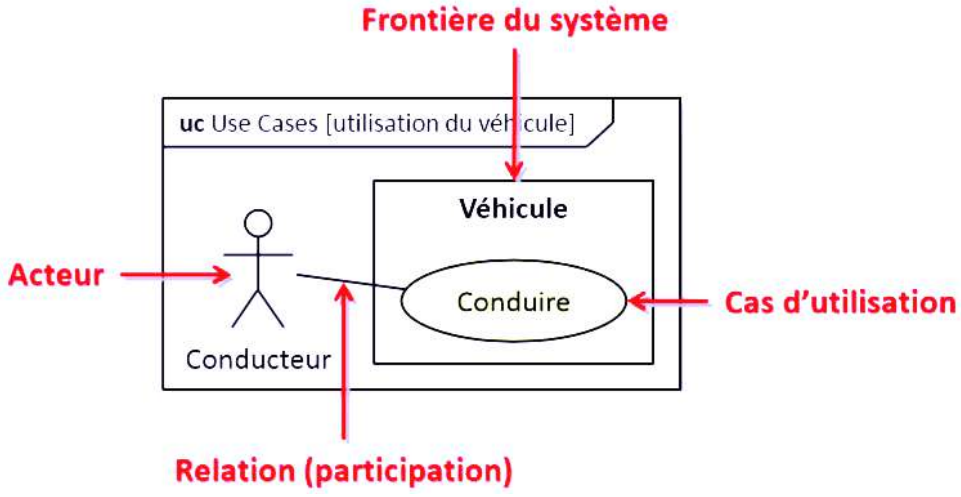


FIGURE I.5 – Exemple de diagramme de cas d'utilisation

I.4.2.2 Diagramme de séquence

Le diagramme de séquence représente les éléments intervenant dans un scénario ou un processus, ainsi que les messages échangés dans un ordre chronologique. Les éléments intervenant sont représentés par des lignes de vie (lifetime en anglais) pouvant être des instances de blocs du modèle. Cette instanciation de blocs établit un lien avec la modélisation structurelle du système.

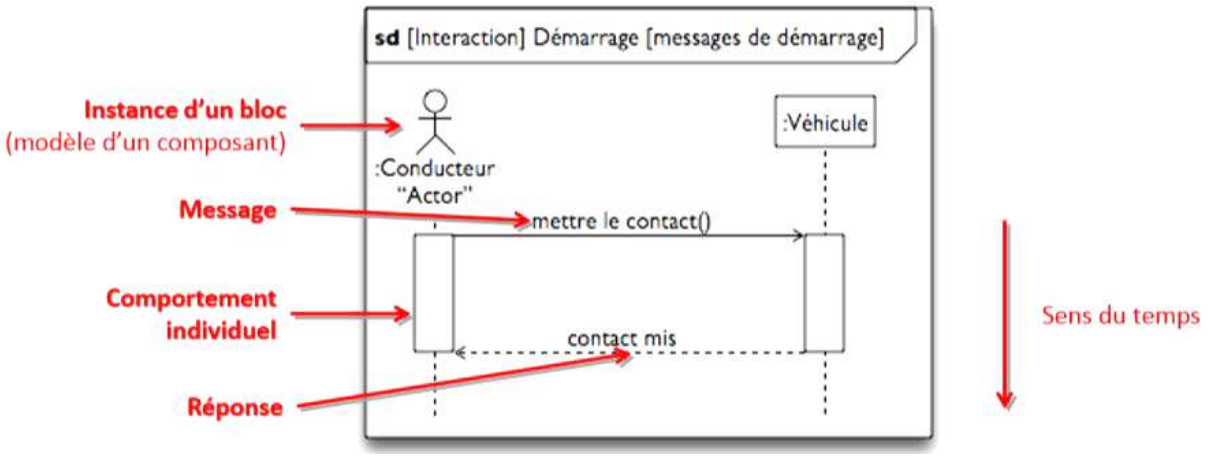


FIGURE I.6 – Diagramme de séquence d'une véhicule

La figure I.6 représente un exemple de diagramme de séquence d'un conducteur de véhicule. L'ensemble des propriétés du diagramme de séquence utilisées en *UML* sont également disponibles avec *SysML* : messages synchrones ou asynchrones, opérateurs (ex : alt, loop, opt, par), références vers d'autres diagrammes de séquence, etc[5].

I.4.2.3 Diagramme d'états

Le diagramme d'états est utilisé avec *SysML* de la même manière qu'avec *UML*, c'est-à-dire qu'il permet de représenter le cycle de vie auquel doivent se conformer toutes les instances d'un bloc donné, ce à travers la modélisation de tous les états possibles.

Seuls les blocs qui sont importants d'un point de vue métier, ou qui sont de nature complexe, devraient avoir un diagramme d'état.

La figure I.7 présente un exemple de modélisation par diagramme d'états. Les propriétés du diagramme d'état *UML* sont également disponibles avec *SysML* : conditions sur évènements, effets, activité durable, transitions, états composites, régions concurrentes, etc.

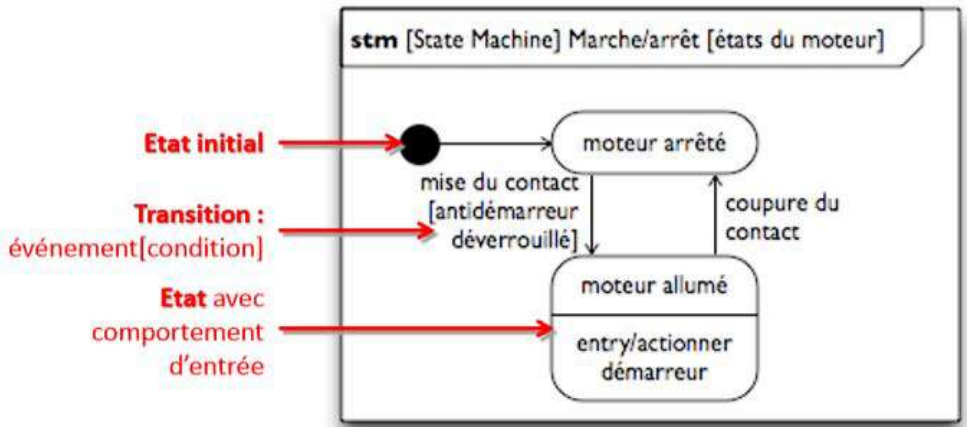


FIGURE I.7 – Exemple de diagramme d'état

I.4.2.4 Diagramme d'activité

Le diagramme d'activités est un diagramme comportemental (DA appelé Activity Diagram en anglais), permet de représenter le déroulement d'un processus sous la forme d'une activité correspondante à une décomposition séquentielle d'actions, aussi appelées tâches. Dans sa forme la plus restreinte, ce diagramme représente un algorigramme, c'est-à-dire un flux de contrôle (ce flux n'a rien à voir avec ceux présents dans le diagramme de blocs internes : il ne faut donc pas les confondre) [6].

Les éléments graphiques utilisés dans ce diagramme ressemblent fortement à ceux du diagramme d'états : chaque tâche est représentée par un rectangle aux coins arrondis et est ensuite reliée à une autre tâche par des transitions représentées par de simples flèches (voir figure I.8). Lorsqu'une tâche est terminée la suivante commence.

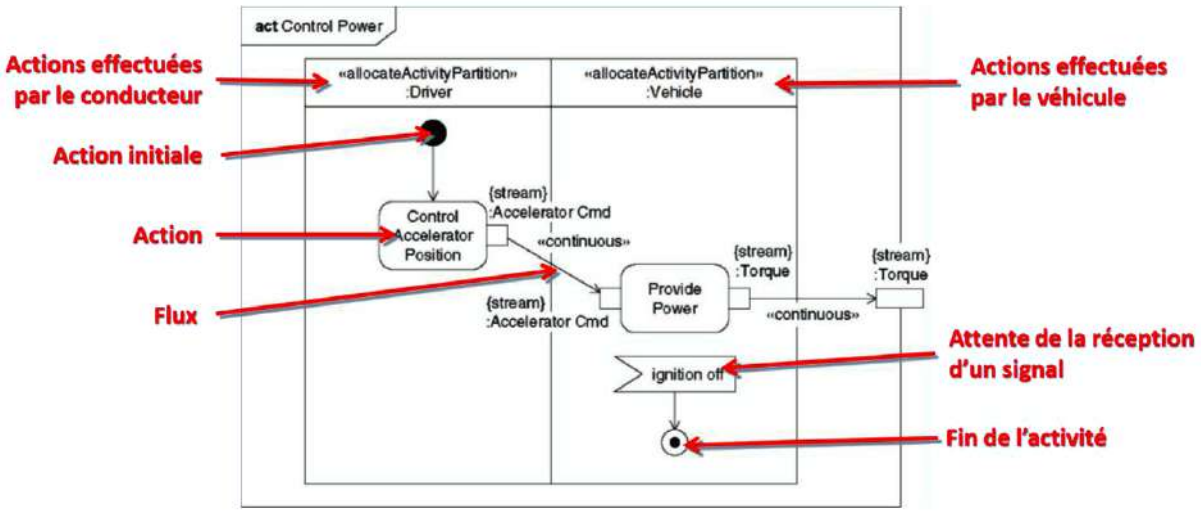


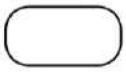
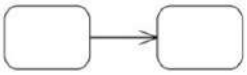
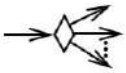
FIGURE I.8 – Diagramme d'activité d'un véhicule

Le diagramme d'activité est utilisé pour représenter les étapes d'un processus, impli-

quant en général les « input et output pins » qui correspondent respectivement au type d'élément requis en entrée d'une activité ou action, et à celui généré en sortie.

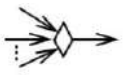
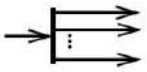
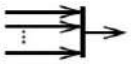
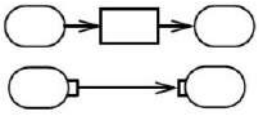
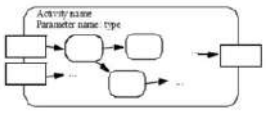
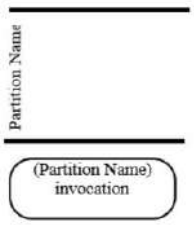
Si une action ou une activité correspond à l'opération d'un bloc, il est alors possible de vérifier que les types d'éléments définis en entrée et en sortie de cette activité soient cohérents avec la signature de l'opération du bloc ou de son interface.

TABLE I.4.1: Les éléments du diagramme d'activité conforme aux spécifications **OMG**

Nom de noeud	**Syntaxe concret**	**Description**
Initial Node		Ceci montre un point de départ ou la première activité de l'écoulement. Dénote par un cercle plein.
Final Node		L'extrémité du diagramme d'activité est montré par un symbole de boudine, également appelé comme activité finale.
Action		La petite unité de fonction executable au sein d'une activité est l'action, elle est représentée sous forme d'un rectangle arrondis, ce dernier contient le nom ou brève description de l'action, si une action est appel d'activité, alors un petit symbole de fourchette est apposé en bas, adroite.
Control flow		Passage instantané et automatique à l'action suivante. Et déclenchées par la fin d'une activité et provoquant le début immédiat d'une autre.
Decision node		Une alternative spécifie des chemins alternatifs, chacun basé sur la valeur d'une expression booléenne (condition de garde). Qui valide ou non le déclenchement d'une transition, les conditions de garde ne doivent pas se chevaucher et couvrir tous les cas possibles.

à suivre dans la page suivante

TABLE I.4.1 – *Suivant*

Nom de noeud	**Syntaxe concret**	**Description**
Merge Node		Un noeud de contrôle qui rassemble de multiples flux alternatives entrant et accepte un flux sortant unique. Merge ne doit pas être utilisé pour synchroniser les flux simultanés.
Fork Node		La décomposition du flux de contrôle en deux ou plusieurs flux de contrôles.
Join Node		La recomposition de flux de contrôle de deux ou plusieurs flux de contrôle en un seule, et permet de spécifier la synchronisation d'activités concurrentes, Les flux qui sortent doivent balancer les flux qui entrent dans la jonction correspondant.
ObjectFlow		Représente la participation d'un objet dans un flux de contrôle, et sont représentés par des relations de dépendance entre objets et états d'action ou d'activités.
Activity		Décrit l'exécution de fonctionnalités ou de comportements. elle est modélisée par plusieurs nœuds d'actions, les nœuds de contrôle et les nœuds d'objets.
Activity Partition		Afin d'organiser un diagramme d'activités selon les différents responsables des actions représentés, il est possible de définir des "couloires d'activités", qui sont en genre de paquet pour la responsabilité d'organisation des activités dans une classe.

Le tableau I.4.1 précédent présente les éléments graphiques de base du diagramme d'activité selon les spécification OMG de *SysML*[5].

Toutes les propriétés des diagrammes d'activités *UML* sont également disponibles avec *SysML*, ex : swimlanes, etc. *SysML* a rajouté quelques spécificités :

– Notion de contrôle pour activer ou désactiver les actions en cours (Control Value).

– Spécification de la nature du débit sur le flot : système continu ou discret.

– Définition de taux et de probabilité sur les flux de contrôle ou d'objets.

I.4.3 Modélisation des exigences

Le diagramme d'exigences est une extension *SysML*. Que ce soit pour l'ingénierie système ou uniquement pour des réalisations logicielles, les exigences sont couramment utilisées pour formaliser les prérequis du système, se traduisant par des fonctionnalités ou conditions qui doivent ou devraient être satisfaites par le système (selon les éventuelles priorités associées aux exigences). L'objectif de ce diagramme et de lier les solutions mises en œuvre sur le système avec les besoins définis dans le cahier des charges.

La figure V.8 représente l'exemple des exigences d'un véhicule modélisé par un diagramme d'exigences.

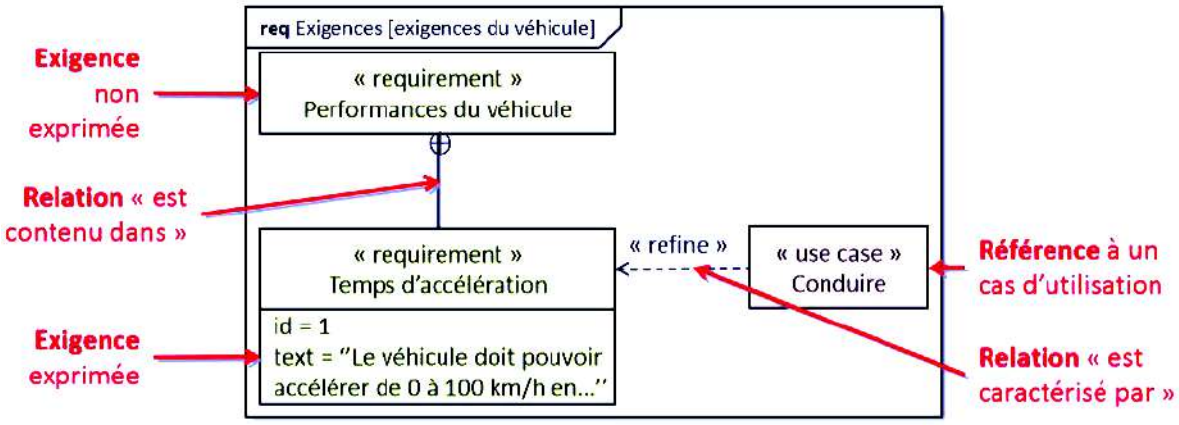


FIGURE I.9 – Un exemple de diagramme d'exigence

Les exigences ont pour objectif d'assurer l'adéquation de la solution (le système réalisé) avec les besoins. Les exigences peuvent être formalisées et catégorisées, par exemple différenciant les exigences fonctionnelles des exigences techniques (performance, fiabilité, ergonomie, etc.).

Les exigences sont également utilisées pour la modélisation, par la création d'associations entre les exigences et les cas d'utilisations, blocs ou tout type d'élément du modèle, établissant la traçabilité du modèle. *SysML* décrit les exigences et leur représentations, s'inspirant des fonctionnalités des outils actuellement disponibles sur le marché. Ainsi, *SysML* définit une représentation graphique et visuelle des exigences textuelles, leur organisation hiérarchique, et des associations avec les éléments du modèle.

SysML définit aussi de nouveaux types d'associations (liens de dépendance stéréotypés) :

– Derive : une ou plusieurs exigences dérivent d'une exigence.
– Satisfy : un ou plusieurs éléments du modèle permettent de satisfaire une exigence.
– Verify : un ou plusieurs éléments du modèle permettent de vérifier et valider une exigence.

– Refine : un ou plusieurs éléments du modèle redéfinissent une exigence.

SysML définit de nouveaux commentaires stéréotypés permettant d'associer une explication à des associations ou éléments du modèle :

– Problem : commentaire dont la description pose le problème ou le besoin.
– Rationale : commentaire dont la description indique la raison ou la justification par rapport à l'élément ou l'association associé.

Un exemple de représentation des exigences sur l'exemple de véhicule (Requirements Traceability) et met dans la figure I.10 :

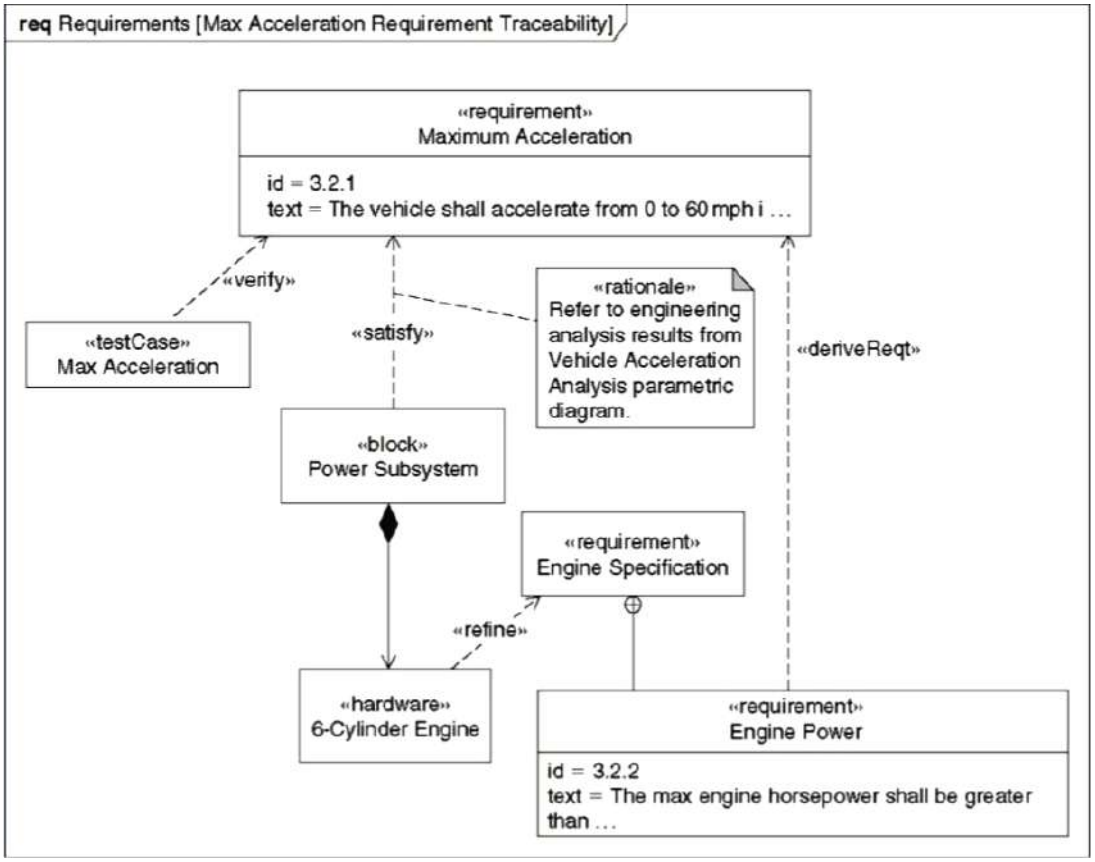


FIGURE I.10 – Diagramme des exigences d'un véhicule

Le concept d'allocation est repris du vocabulaire des ingénieurs systèmes, indiquant un ensemble d'éléments associés dans un environnement structuré. La modélisation système implique des tentatives d'allocations entre les éléments du système. Cet ensemble d'allocation est utilisé pour générer une matrice pour vérifier que les parties du système sont correctement intégrées. La création d'allocations permet de maintenir une cohérence entre les éléments du système, en particulier entre les modèles dynamiques et les modèles structurels.

I.5 Outils SysML

Le langage *SysML* a été intégré par de nombreux éditeurs d'outils commerciaux ou open source :

- Sparx Systems Enterprise Architect (plugin *SysML* ou version Ultimate requise).
- IBM Rational Software Modeler (plugin d'une société tierce disponible).
- MagicDraw (plugin *SysML* requis)
- Open source : POLARSYS, TOPCASED.

Dans ce travail, nous avons choisi d'utiliser l'environnement *POLARSYS*, qui est une boite à outils open source (environnement Eclipse).

I.6 Conclusion

SysML se présente comme un langage visant à supporter une approche *orientée modèle* dans les processus d'ingénierie système, il s'agit d'un langage de modélisation qui permet de supporter les phases de spécification, d'analyse, de conception et de vérification des systèmes complexes. Son expressivité, son pouvoir d'abstraction, sa modularité et son aptitude pour la description de larges spécifications, ainsi que le niveau relativement satisfaisant d'automatisation qu'on peut en tirer. *SysML* est donc un langage suffisamment flexible pour permettre son utilisation dans divers cas de développement, cet aspect positif est constamment mis en valeur. C'est sa grande force, mais en même temps sa plus grande faiblesse, il souffre principalement d'une faible sémantique formelle.

Chapitre II

Réseau de Petri pour la vérification formelle

II.1 Introduction

L'intégration des méthodes formelles [7] dans les activités de développement des systèm--es a confrontés un obstacle liée à la difficulté réelle de manipuler les concepts théoriques et des méthode d'analyse associer, ainsi que la capacité des développeurs d'exprimer les propriétés du système de façon aisée. En effet, les méthodes formelles reposent sur l'utilisation des langages formels [8] pour donner une spécification du système. Parmi les formalismes les plus utilisés, les réseaux de Petri, ils occupent une place prépondérante et sont parmi les outils ayant été le plus largement étudiés, si l'on considère la diversité des techniques automatiques de vérification qui leur sont associés. Ils offrent une structure très simple et constituent toujours le support de nombreuses études théoriques des systèmes concurrents et de leurs propriétés.

Dans ce chapitre, nous présentons le modèle formel que nous avons utilisé comme support de vérification, les réseaux de Petri. Ensuite, nous abordons l'utilisation des techniques de vérification fondées sur les modèles *Modèle-Checking* pour analyser des systèmes spécifiés en réseau de Petri. En particulier, nous présentons l'outil *CPNTools*, un outil utilisé pour l'édition, la simulation et l'analyse des réseaux de Petri colorés hiérarchiques.

II.2 Les Réseaux de Petri

Le formalisme des réseaux de Petri(RdP) représente un outil mathématique puissant dans le domaine de la modélisation et de la vérification des systèmes. En plus de leur force d'analyse, ils offres une représentation graphique simple qui aide à la modélisation des systèmes complexes.

Les RdP peuvent donc représenter aussi bien des processus informatiques (tâches) que des processus industriels (chaîne de commande de machines) mais aussi tous les systèmes caractérisés par leurs aspects parallèles, asynchrones, distribués, concurrents, non déterminis--tes et/ou stochastiques [9] [10].

Historiquement, le RdP est présenté par *Carl Adam Petri* dans sa thèse " Communication avec des Automates " en *Allemagne* à *Bonn* en 1962.

Un RdP est un graphe biparti orienté valué qui a deux types de nœuds : les places et

les transitions. L'arcs doit faire le lien entre deux types différents de nœuds c-à-d, il est orienté soit d'une place vers une transition ou soit d'une transition vers une place, il est étiqueté par une valeur (ou un poids), qui est un nombre entier positif. L'arc ayant k poids peut être interprété comme ensemble de k arcs parallèles. L'étiquette du poids égale à 1 est ignorée. Un marquage assigne un nombre entier à chaque place, si un marquage assigne un entier k à une place p, on dit que p est marqué par k jetons. On considère d'abord la structure fondamentale qui se compose de deux sortes d'objets (les places et les transitions) et des relations entre eux.

Graphiquement les places sont représentées par un cercle. Et les transitions par un trait ou un rectangle. Il convient aussi qu'un arc possède un poids avec une valeur entière, le poids devra être indiqué sur l'arc. Le marquage est représenté par des points noirs(jetons) à l'intérieur de la place. La figure II.1 représente un *RdP* avec l'ensemble de trois places et quatre transitions, reliées par huit arcs orientés.

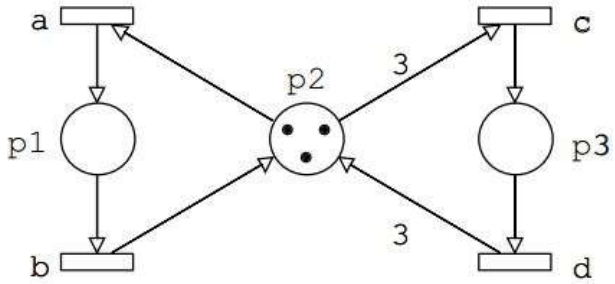


FIGURE II.1 – Exemple de Réseau de Petri

II.2.1 Définitions formelles

Un *RdP* est un 5-tuple, $PN = (P, T, F, W, M_0)$ où :

- P un ensemble fini de places .
- T un ensemble fini de transitions .
- $F \subseteq (P \times T) \cup (T \times P)$ ensemble d'arcs .
- $W : F \to N$ une fonction de poids.
- $M_0 : \mathrm{P} \to N$ le marquage initial.

$P \cap T = \emptyset$ et $P \cup T = \emptyset$

Un *RdP* (P, T, F, W) sans aucun marquage initiale spécifique est noté N. Et un *RdP* avec un marquage initial donné M_0 est noté par (N, M_0) [9].

II.2.2 Marquage initial

Le marquage initial d'un *RdP* est un vecteur représente le nombre de jetons dans chaque places p_i dans le *RdP* ($p_i \in P$) à l'état initial. Le vecteur M_0 désigne le marquage initial de *RdP* représenté dans la figure II.1.

$$M_0 = \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$$

II.2.3 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.

II.2.3.1 Définition

Soit un *RdP* P avec m places et n transitions. On appelle matrice des pré-conditions pre, la matrice mxn à coefficients dans N tel que $pre(i,j) = w(pi,tj)$, 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 définis la matrice nxm des post-conditions $post(i,j)$, tel que $post(i,j) = w(tj,pi)$ contient le nombre de marques déposées dans p_i lors du franchissement de la transition t_i. La matrice $C = post–pre$ est appelée matrice d'incidence du réseau P.

II.2.4 Dynamique des *RdP*

Les *RdP* sont principalement utilisés pour modéliser le comportement d'un système au cours du temps. Ce comportement est concrétisé par l'évolution du marquage (le déplacement des jetons de place en place). Ces mouvements sont régis par des règles.

II.2.4.1 Franchissement d'une transition

Une règle de franchissement est une simple relation de transition qui définit le changement d'état dans un réseau marqué lors de l'exécution d'une action. Soit un *RdP* N, t une transition de N, et M un marquage. On dit que la transition t est sensibilisée ou franchissable ou tirable pour le marquage M, si et seulement si :
$\forall\ p \in P\ M(p) \geq Pre(t,p)$ (i.e $M \geq W(t,\bullet)$).
On note par $M[ti$ que la transition t est sensibilisée par le marquage M.

Toute transition franchissable par un marquage M $M[ti$ peut être tirée (franchie) et son tir conduit à un nouveau marquage M' défini, pour tout p, par :

$$M'(p) = M(p) - Pre(t,p) + Post(t,p)$$

II.2.4.2 Séquence de franchissement

Une séquence de franchissement $\ll \sigma \gg$ est une suite de transition $(t_1, t_2, \ldots, t_n)$ qui permet, à partir d'un marquage $\ll M \gg$, de passer au marquage $\ll M' \gg$ par le franchissement successif des transitions définissant la séquence $M_0[\ \sigma > M_p$.

II.2.4.3 Marquage accessible

Le marquage d'un *RdP* à un instant donné est un vecteur dont la valeur de la I^{eme} composante est le nombre de marques dans la place P_i à cet instant. Le franchissement d'une transition conduit à un changement du marquage. Le passage du marquage M_k au marquage M_l par franchissement de la transition T_j est noté : M_k ($T_j \geq M_l$. Le nombre de marques dans la place P_i pour le marquage M_k est noté $M_k(P_i)$. À partir d'un Même marquage, il peut être possible de franchir plusieurs transitions, menant ainsi à des marquages différents. L'ensemble des marquages accessibles à partir du marquage M_0 est l'ensemble des marquages obtenus à partir de M_0 par franchissements successifs d'une ou plusieurs transition(s). Cet ensemble est noté $A(R\,;\,M_0)$.[9]

II.2.4.4 Graphe de marquages accessibles

On peut représenter l'ensemble de marquages accessibles par un graphe si ce dernier et fini. Le graphe de marquages a comme sommet l'ensemble de marquages accessibles $A(R,$ $M_0)$ un arc orienté relie deux sommets M_i et M_j s'il existe une transition t franchissable permettant de passer d'un marquage à un autre M_i [$t > M_j$ Les arcs du graphe sont étiqueté par les transitions correspondantes.
La figure II.2 représente le graphe de marquages accessibles correspond au *RdP* dans la figure II.1.

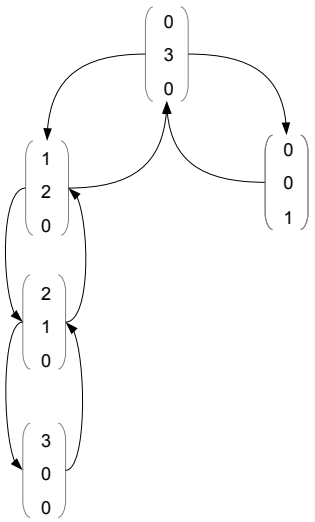


FIGURE II.2 – Graphe de Marquage

II.2.4.5 Arbre de couverture

L'arbre de couverture est une méthode alternative. Le mécanisme de construction est le même que pour le graphe des marquages accessibles. À ceci prêt que pour chaque nouveau marquage (nœud du graphe) ajouté, on vérifie s'il n'est pas supérieur à un marquage déjà

présent sur au moins une séquence entre M_0 et le nouveau marquage. Si tel est le cas tous les marquages de place supérieurs sont remplacés par ω. Ce symbole matérialise le fait que la place en question peut contenir autant de jetons que souhaité (elle est donc *non bornée*). Dans la suite, les places non bornées le demeurent naturellement et ceci quelles que soient les transitions franchies, ainsi le symbole ω ne disparaît jamais. Cette méthode produit le graphe de couverture.

II.2.5 Propriétés comportementales

Les propriétés génériques informent le développeur sur le comportement général du *RdP*. Celles-ci doivent être complétées par l'analyse des propriétés spécifiques du système modélisé.
La notion de vivacité est fondamentale dans la dynamique d'un *RdP* car elle exprime la possibilité pour toute transition d'être franchie à un instant de la vie du réseau [11] [12]. Dans le cas contraire, on dit qu'il y a un blocage, en d'autres termes, une partie du réseau est (ou bien est devenu) inactive [13]. En effet, si une transition est infranchissable, certains états du système ne seront jamais atteints, il y a donc peut être erreur de conception car le système devient incapable de réaliser certaines fonctionnalités pour lesquelles il a été conçu.

II.2.5.1 Bornitude

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. Pour vérifier si un *RdP* est borné on examine son arbre de couverture. Un *RdP* est borné pour un marquage initial M_0 si et seulement si l'arbre de couverture ne contient aucune composante à ω.

- **Place bornée :** Une place est dit k-borné ou simplement bornée si le nombre de jetons dans chaque place ne peut pas dépasser un nombre fini K, pour tout marquage accessible depuis le marquage initiale M_0.
  La place p $\in P$ est bornée si et seulement si :
  $$\forall\ M \in M_0[i, \exists\ k \in N \text{ tel que } M(p) \preceq k.$$
- **Réseau borné :** Un *RdP* est borné pour un marquage initial M_0 si toutes les places sont bornées pour M_0.
  On dit qu'un *RdP* est sain s'il est 1-borné.

II.2.5.2 Vivacité

La vérification de la vivacité correspond à vérifier qu'à tout instant le système conserve la possibilité de reproduire toutes les actions.

- **Vivacité d'un réseau :** Un *RdP* (N, M_0) est vivant (ou d'une manière équivalente M_0 est un marquage vivant de N) si, quelque soit le marquage obtenu du marquage initiale M_0, il est possible de franchir n'importe quelle transition dans le réseau par

progression à travers une séquence de franchissement. Cela signifie qu'un *RdP* vivant garantie l'absence d'impasse, quelque soit la séquence de franchissement choisie.

- **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. La vérification qu'une transition est vivante consiste à vérifier que celle-ci apparait dans toutes les composantes fortement connexes du graphe des marquages.
- **Pseudo-vivacité :** Un *RdP* (R, M_0) 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.

II.2.5.3 État de blocage

L'absence de blocage est une propriété du graphe des marquages accessibles. Il s'agit d'une propriété essentielle des *RdP*. Elle assure que le réseau pourra toujours progresser. Un réseau est sans blocage si l'ensembles des marquages directement accessibles de tout marquage accessible est non vide.

Un état $M \in M[i>$ est un état de blocage (état puits), si et seulement si aucune transition n'est franchissable.

II.2.5.4 Réversibilité et état d'accueil

Un *RdP* (N, M_0) est réversible si, pour tout marquage M dans $R(\mathrm{M}_0)$, M_0 est accessible depuis M. Dans un *RdP* réversible il est toujours possible de retourner en arrière vers l'état initiale. On défini un état d'accueil par un marquage M' est un état d'accueil si pour tout marquage M dans $\mathrm{R}(\mathrm{M}_0)$, M' est accessible depuis M.

II.2.6 Extensions des *RdP*

Les *RdP* étant relativement récents et disposant d'une représentation graphique attrayante, ils connaissent de nombreux développements. En effet, vu la complexité croissante des systèmes industriels et pour répondre aux besoins de modélisation toujours plus sévères, les chercheurs ont inventé de nouvelles formes de *RdP* plus performants et plus spécifiques [14]. Cependant cette évolution est trop dense pour être présentée ici. Les *RdP*s ont connu plusieurs extensions au fils des années. Les extensions les plus connues sont les *RdP* colorés [15], les *RdP* algébriques, les *RdP* Predicate/Transition [16], les G-Nets [17], Nous ne présenterons donc que les extensions qui nous intéressent par la qualité

de leurs réponses aux besoins des applications industrielles et leur adéquation à l'analyse des systèmes étudiés.

II.2.6.1 Réseaux de Petri colorés

Lorsque le nombre d'entités du système à modéliser est important, la taille du *RdP* devient rapidement énorme, et si les entités présentent des comportements similaires, l'usage des *RdP* colorés [15] permet de condenser le modèle. Les *RdP* colorés sont des *RdP* dans lesquels les jetons portent des couleurs. Une couleur est une information attachée à un jeton. Cette dernière permet de distinguer des jetons entre eux et peut être de type quelconque. Ainsi, les arcs ne sont pas seulement étiquetés par le nombre de jetons mais par leurs couleurs. En effet, les jetons appartiennent à des domaines de couleurs qui permettent de les distingués. Ainsi, à chaque place est associé un domaine de couleur qui permet de différencier les jetons que peut prendre cette place. Et à chaque transition aussi associé un domaine de couleur afin de choisir les couleurs de franchissement.
Les pré-conditions deviennent des applications qui associent à chaque couleur tirée les couleurs qui seront prélevées des places en entrée. Et les poste-conditions deviennent des applications qui associaient à chaque couleur tirées les couleurs qui seront déposées dans les places en sortie. Les réseaux colorés n'apportent pas de puissance de description supplémentaire par rapport aux *RdP* P/T, ils permettent juste une condensation de l'information c-à-d ils permettent simplement un abréviation de ces derniers.
L'opération de passage d'un *RdP* simple au *RdP* coloré est appelée opération de pliage et l'opération inverse est appelée dépliage.

II.2.6.1.1 Notations

La définition et l'analyse des *RdP* colorés, qui seront définis ultérieurement dans ce chapitre, sont basées sur la notion de multi-ensemble. Intuitivement un multi-ensemble est un ensemble dans lequel un élément peut apparaitre plusieurs fois.

Afin de définir plus précisément les *RdP* colorés. D'abord on définit des multi-ensembles.

- **Les multi-ensembles :**
  Soit A un ensemble fini non vide, un multi-ensemble μ sur A dans l'ensemble des entiers naturels N. On note $\mu = \sum_{x \in A}$ (x), où μ(x) désigne le nombre d'occurrences de x dans le multi-ensemble μ. où Bag(A) désigne l'ensemble des multi-ensembles sur A.

II.2.6.1.2 Définition formelle des *RdPC*

Comme présenté dans [18], [19], [20], un *RdP* coloré est un tuple (P, T, Pre, $Post$, M_0, C) Où :
- P est un ensemble fini de places.
- T est un ensemble fini de transitions disjoint de P.
- C est l'ensemble des couleurs, $C = c1, c2, c3, \ldots, c_n$ appelé type. A chaque place est as-

socié un type ou un ensemble de couleurs.
- Pre, $Post : P \times T \rightarrow IN$ où Pre, $Post$ sont respectivement des fonctions d'incidence avant, et des fonctions d'incidence arrière. Ces fonctions sont associées à chaque arc pour traduire la relation qui existe entre la couleur et la transition associée, celle choisie pour franchir cette transition (couleur de franchissement) et celle associée à la place correspondante (couleur du jeton).
- M_0 est un ensemble multiple appelé marquage initial du réseau.

II.2.6.1.3 Transition tirable

Une transition t est tirable ou franchissable depuis un marquage M si et seulement s'il existe une couleur $c \in C(t)$ telle que les jetons nécessaires pour la couleur c de franchissement sont disponibles dans M :

$$\forall \text{ p} \in P \; M(p) \geq pre(p,t)(c)$$

II.2.6.1.4 Franchissement d'une transition

Lorsqu'une transition t est franchie ou tirée depuis M pour une couleur $cinC(t)$, un nouveau marquage M' est atteint :

$\forall \text{ p} \in P \; M'(p) = M(p) + C(p,t)(c)$ ou $C = post$ - pre.

Le franchissement d'une transition t depuis un marquage M vers un marquage M' est noté $M \rightarrow M'$. On dit aussi que M' est directement accessible de M.

La figure II.3 présente un $RdPC$ avec deux palace. La première place $P1$ liée à deux domaines de couleur $< R >< B >$, et marquée avec un marquage $C1$. La deuxième place $P2$ liée au domaine $< R >$, et marqué avec $C2$. les arcs $(P1, T1)$ et $(T1, P2)$ sont étiquetés avec les fonctions $f1$ et $f2$. Au-dessous de la figure II.3, les domaines de couleurs et les fonctions sont bien présenté.

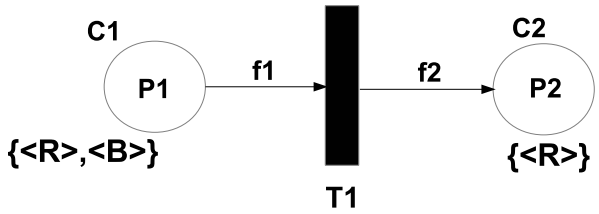


FIGURE II.3 – Exemple d'un RdP coloré

Avec :

<R> = Rouge.

<B> = Bleu.

<V> = Vert.

<N> = Noir.

C1 = <R>, <B>

C2 = <R>, <B>, <V>, <N>

f1 <R> = <R> ;f1 <B> =<B>

f2<R> =<V> ;f2<B> =<N>

II.2.6.1.5 Analyse des *RdPC*

Il est essentiel de constater qu'à un *RdPC* on peut associer un *RdP* ordinaire qui aura un comportement identique. Les propriétés du réseau coloré (borné,vivant,. . .) sont donc celles du réseau ordinaire associé. De ce fait, l'analyse du graphe des marquages accessibles des *RdPC* se fait de la même façon que celles des *RdP* simples.

II.2.6.2 Réseaux de Petri hiérarchiques

Le *RdP* hiérarchique (*RdPH*) est un modèle dans lequel une partie peut être représentée par une transition de substitution qui est une abstraction d'un autre modèle. C'est-à-dire, que cette partie est détaillée à part.
Le principe de *RdP* hiérarchiques [21], est de construire un modèle en combinant un certain nombre des réseaux appelés **page** en un seul réseau. L'objectif est de réduire la complexité d'un modèle en permettant la modélisation modulaire. Dans ce qui va suivre, nous allons définir la sémantique utilisée par [15]. Donc un *RdP* coloré hiérarchique (*RdPCH*) est un n-tuplet : $RdPCH = (S, SN, SA, PN, PT, PA)$ où :

- S est un ensemble fini non-vide de *pages* où chaque page représente un *RdP* coloré non-hiérarchique,
- SN est un ensemble de nœuds de substitution appartenant à l'ensemble T (ensemble de transitions),
- SA est une fonction d'assignation des pages définies à partir de SN dans S. Cela veut dire que le nom (ou l'assignation) du nœud de substitution correspond au nom (ou l'assignation) de la sous-page qui le représente par un *RdPC* non-hiérarchique.
- PN est un ensemble de nœuds appelés port appartenant à l'ensemble P.
- PT est le type de fonctions associé au nœud 'Port' des éléments PN. Ce type de fonctions peut être : in, out, i/o (in/out).
- PA est une fonction d'assignation du port qui définit la relation entre les nœuds, et entre les pages et les sous-pages.

  Le type d'interconnexion entre les pages et les sous pages s'effectue à partir des transitions. Une **transition de substitution** remplace une page complète.

Pour mieux illustrer cette définition, un exemple introduisant ces notions est représenté par la figure II.4

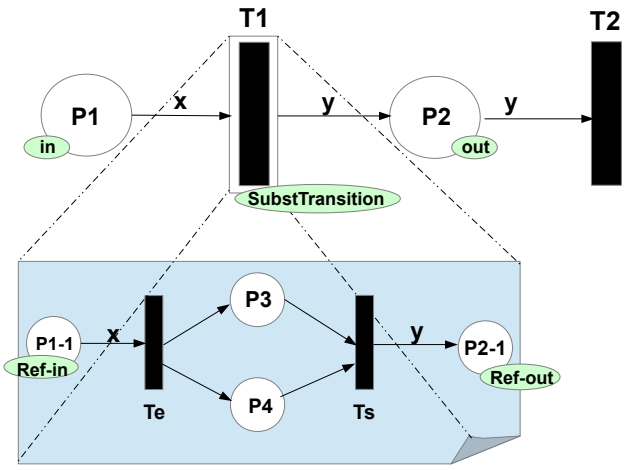


FIGURE II.4 – Réseau de Petri Hiérarchique

II.2.7 Outils d'analyse des RdP

L'aspect formel des *RdP* a encouragé les développeurs à mettre au point une multitude d'outils de simulation et de vérification des *RdP* selon la technique de vérification du modèle (*CPNTools*, CPNAMI, PROD, Petri Net Kernel, JPetriNet, Opera, ...etc). La plupart de ces outils présentent un environnement graphique d'édition des *RdP* avec la possibilité de simuler le modèle et d'analyser des propriétés génériques des *RdP*. Il faut avoir un outil efficace, qui nous aidera à rapprocher d'une vision plus appliquée. Dans notre cas nous choisissons l'outil *CPNTools*.

II.2.7.1 CPNTools

CPNTools [22] est à l'origine développés par le Groupe CPN à l'Université d'Aarhus de 2000 à 2010. Les principaux architectes de l'outil sont Kurt Jensen, Søren Christensen, Lars M.Kristensen, et Michael Westergaard. Depuis l'automne 2010, l'outil *CPNTools* est transféré au groupe AIS, l'Université de technologie d'Eindhoven, aux Pays-Bas.
CPNTools est un outil des *RdP* de haut niveau. Il prend en charge les *RdP* de base et les *RdPC* plus les *RdPH*. Il dispose d'un simulateur et un outil d'analyse de l'espace d'état. Il est l'un des applications les plus utilisées dans son domaine, il se compose de :

- *CPN editor* : permet d'éditer les réseaux colorés et de vérifier la syntaxe.
- *CPN simulator* : permet de simuler le comportement de réseau (les régles de franchissement, les transitions franchissables, etc).
- *CPN state space tool* : supporte la vérification sur l'espace d'états [1] (SS).

1. *Stat space* de l'anglais

CPNTools combine la capacité des *RdP* coloré et la capacité d'un langage fonctionnel (le langage Standard *ML* [23]). *CPNTools* fournit des primitives pour décrire les processus concourants, tandis que le langage fournit des primitives pour définir des types de données (ensemble de couleur) et des manipulations des données [24].

II.2.7.1.1 Support des couleurs par *CPNTools*

La déclaration des couleurs et des variables est effectuée à l'aide du langage *ML* basé sur le langage de programmation *ML*. Grâce au logiciel *CPNTools*, il est possible d'utiliser des couleurs complexes et des fonctions. Les couleurs simples existantes dans le *CPN ML*, sont : unité (couleur noir –sans couleur-), entier, booléen, string (chaîne de caractères) et énuméré.
Voila un exemple (figure II.5) tirée de [25] d'un *RdP*C édité par *CPNTools* modélise le système du diner des philosophes, et les déclarations suivent le langage fonctionnel Standard ML.

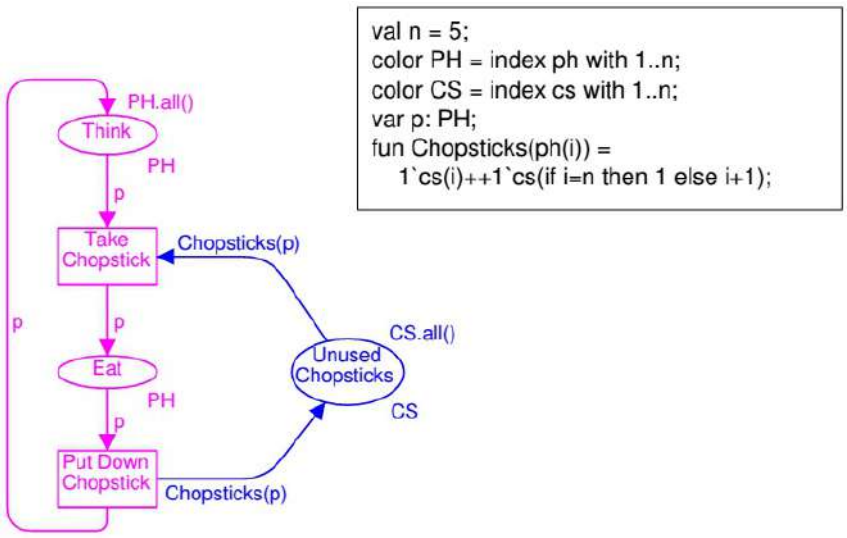


FIGURE II.5 – *RdPC* modélisé par *CPNTools*

II.3 Vérification

Le but de la vérification est d'assurer qu'un programme (ou système) satisfait un certain nombre de propriétés. 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" [26].

II.3.1 Approches de vérification formelle

Les méthodes de vérification formelles [27], sont des techniques permettant de raisonner rigoureusement à l'aide de logiques mathématiques sur des programmes informatiques ou des systèmes, afin de démontrer leur validité par rapport à une certaine spécification. Ces méthodes permettent d'obtenir une très forte assurance de l'absence de bug dans les logiciels, c'est-à-dire d'acquérir des niveaux d'évaluation d'assurance élevés, elles sont basées sur les sémantiques des programmes sur des descriptions mathématiques formelles du sens d'un programme donné par son code source (ou parfois, son code objet). Cependant, elles sont généralement coûteuses en ressources (humaines et matérielles) et actuellement réservées aux logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique. De façon générale, on distingue deux grandes catégories de techniques pour vérifier formellement la correction d'un système. La première catégorie regroupe les techniques de preuve [2] 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. La seconde 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.

II.3.1.1 Preuve de théorème

Pour appliquer la technique de preuve de théorème, nous devons exprimer sous forme de formules logiques le modèle et les propriétés à prouver. Ces formules peuvent être combinées entre elles en respectant des règles définies (un système formel) afin de déduire de nouvelles formules [28]. Les preuves de théorèmes se font ainsi par dérivation à partir d'un ensemble initial de propriétés connues sur le modèle. Toutefois, le problème qui persiste est l'automatisation totale des constructions de preuves. En pratique, les outils de preuve de théorème ne sont pas entièrement automatiques, ils ont besoin de l'intervention interactive de l'utilisateur et leur mode d'emploi nécessite une bonne expertise. D'un autre part, si ces preuves permettent d'établir la correction d'un modèle, elle se révèle peut pratique en cas d'erreurs, si une propriété n'est pas vérifier, il peut être difficile de trouver le comportement du système qui en est la cause.

II.3.1.2 Vérification basé sur les modèles (Model-Checking)

La vérification du modèle ou le *model-checking* est une technique de vérification basée sur l'exploration exhaustive de toutes les configurations ou états du système à la recherche de comportements déviants de la spécification. Ces comportements sont spécifiés à l'aide de propriétés exprimées sur le modèle. Le principe du model-checking [29][30] est de vérifier, étant donné une représentation M d'un système réel et une spécification φ, si

2. (theorem proving) de l'anglais

M est un modèle de la spécification ($M \models \varphi$?).
Un *model-checker* peut être vu comme une boite noire qui reçoit en entrée un système ainsi qu'une propriété exprimée sur ce système, et produit en retour une réponse indiquant si la propriété est vérifiée ou non. Les algorithmes mis en œuvre incluent généralement une construction de l'éspace d'état du système puis un parcours de cet espace à la recherche d'erreurs. Cet espace d'état est un graphe dirigé qui décrit toutes les évolutions possibles du système. Les nœuds de ce graphe sont les états du système et les arcs représentent les transitions entre ces états. Ainsi, si le système étudié est un programme, un nœud correspondra à un état de la mémoire (valeurs des variables, compteurs ordinaux des processus, ...), et les transitions seront les instructions du programme. Le model checker peut aussi détecter ces erreurs en même temps qu'il construit l'espace d'état et s'arreter si une telle erreur a été détectée. On parle alors de vérification à la volée.
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. La figure II.6 décrit le principe générale de *model-checking*.

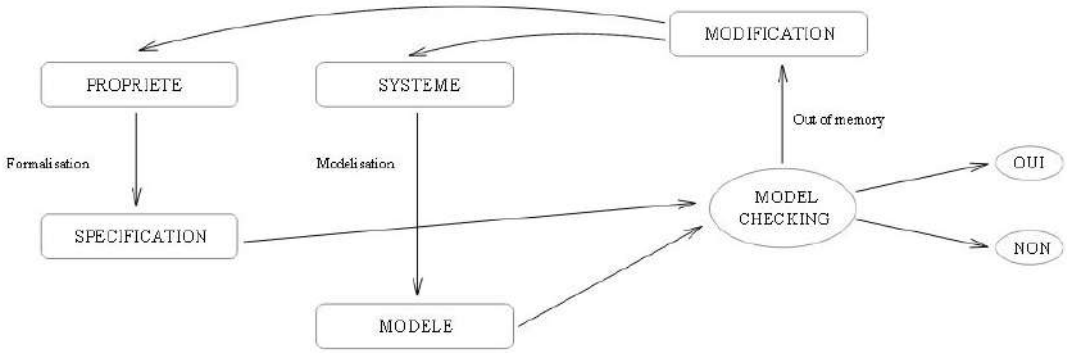


FIGURE II.6 – Principe de model-checking

II.3.1.2.1 Processus du model-checking

En pratique, le *model-checking* est un processus en trois phases :

1. Modéliser le système (système de transitions $\mathcal{M}$) et les spécifications (logique temporelle ϕ).
2. Vérifier si $\mathcal{M}$ satisfait ϕ ou non. Si non, retourner un contre-exemple.
3. Analyser les résultats obtenus.

II.3.1.2.2 Système de transitions et espace d'état

Pour vérifier toutes les exécutions possibles du systéme étudié, on représente son modèle

sous la forme d'un système de transitions a états, où chaque état correspond à un état du système à chaque pas d'excution et chaque transition à une évolution possible d'un état donné vers un autre selon l'action exécutée par le système. Ce système de transition peut être représenté par un graphe orienté ou automate, une machine de *Turing* comme il peut être généré à partir un *RdP*.

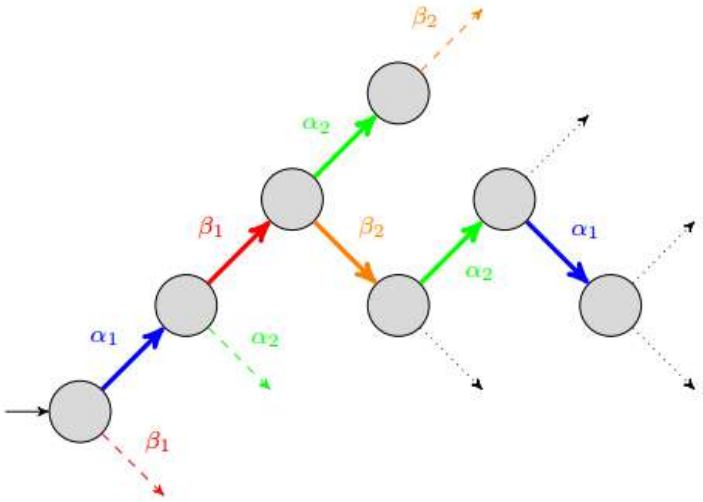


FIGURE II.7 – Système de transitions et espace d'états

La figure II.7 introduit un espace d'état qui est l'ensemble des états du système de transitions accessibles à partir d'un état initial en suivant les chemins du système de transitions.

II.3.2 Les propriétés à vérifier

Les propriétés d'un système sont souvent classées en deux catégories :

Sûreté et Vivacité :

- les propriétés de **sûreté** qui expriment que «quelque chose de mauvais n'arrive jamais».
- les propriétés de **vivacité** qui expriment que «quelque chose de bon est toujours possible».

II.3.3 Spécification des propriétés

Les comportements des systèmes modélisés sont les chemins d'exécution des systèmes de transitions. Les propriétés sur ces derniers peuvent être exprimées à l'aide de la logique temporelle (Temporal Logic). Il existe plusieurs langages de logique temporelle, les plus utilisés sont la logique temporelle linéaire LTL (Linear Temporal Logic), la logique temporelle arborescente CTL (Computational Temporal Logic) et plus particulièrement la logique $CTL*$.

II.3.3.1 Logiques temporelles

L'intérêt des logiques temporelles [31][32] est qu'elles permettent de spécifier des propriétés dans un langage logique le long des exécutions (par exemple d'un système de transitions). On n'a donc pas besoin de connaître la structure interne du système à vérifier pour spécifier des propriétés. La phase de spécification peut ainsi être réalisée distinctement de celle de modélisation du comportement du système. Les logiques temporelles permettent d'exprimer des propriétés dynamiques, faisant référence au temps discret. Pour spécifier des propriétés des exécutions on utilise une logique temporelle linéaire (l'état du système à l'instant suivant de celle de courant est complètement déterminé). Dans le cas de l'arbre des exécutions on utilise une logique temporelle arborescente (l'état du système à l'instant suivant de l'instant courant n'est pas unique).

II.3.3.1.1 LTL : Linear Temporal Logic

La logique temporelle linéaire [33] permet de spécifier des propriétés d'exécutions. On considère un ensemble fini de propriétés atomiques AP. La syntaxe de LTL est la suivante :

Définition (Formules de LTL)

- $\forall\ p \in AP$, $p \in LTL$.
- si p, $q \in LTL$, alors $p \vee q$, $\neg\ P \in LTL$.
- si p, $q \in LTL$, alors $\times\ p$, $p \cup q \in LTL$.

La signification intuitive des ces formules pour les modalités $\times p$, $p \cup q$ est la suivante : $\times\ p$ est vraie sur une séquence $\pi = p_0 p_1 p_2 .. \ . \ p_n .. \ .$ ssi la sous séquence $p_1 p_2 .. \ . \ p_n ...$ satisfait p. $p \cup\ q$est vraie sur π ssi p est vraie depuis le début de π jusqu'à un instant où le suffixe de la séquence satisfait q.

On note **tt** la valeur booléenne vraie et **ff** la valeur booléenne fausse.

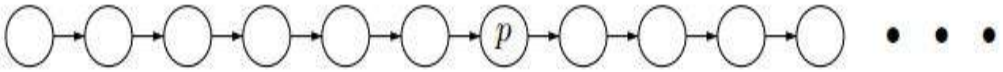
Sémantique de LTL

Soit $p = \mathrm{q}_0 q_1 \cdot \cdot \cdot \mathrm{q}_n \cdot \cdot \cdot$ une séquence d'états et L telle que $\forall\ i \geq 0$, $L(qi) \subseteq AP$. La relation $\ll p \models \phi \gg$ se lit $\ll$la séquence p satisfait $\phi \gg$. Cette relation est définie inductivement et donne la sémantique des formules de LTL :

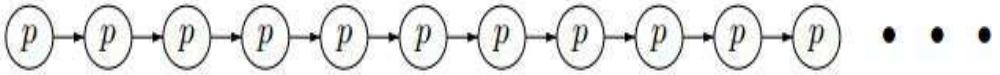
- $p \in AP$, $p \models p \longleftrightarrow p \in L(q0)$;
- $p \models p \vee q \longleftrightarrow p \models p$ ou $p \models q$;
- $p \models \neg\ p \longleftrightarrow p \models p$;
- $p \models \times\ p \longleftrightarrow p1 \models p$;
- $p \models p \cup q \longleftrightarrow \exists\ j \geq 0$, $p_j \models q$ et $(\forall\ k \ ¡\ j,\ p_k \models p)$.

Certaines formules de LTL sont souvent utilisées car elles correspondent à des propriétés standard et les abréviations suivantes sont souvent rencontrées :

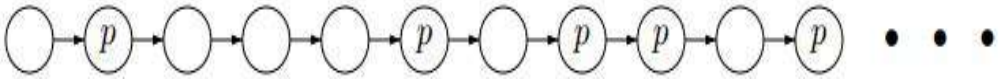
– fatalement (ou inévitablement) p : $\mathbf{F}p \equiv \mathbf{tt} \cup p$.



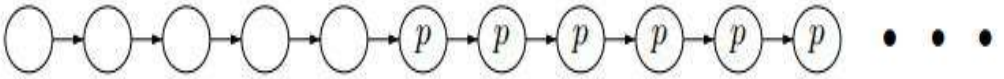
– toujours p : $\mathbf{G}$p $\equiv \neg\ \mathbf{F}\ \neg$ p (p est un invariant).



– infiniment souvent p : $\mathbf{GF}$p



– presque partout p : $\mathbf{FG}$p



II.3.3.1.2 CTL : Computational Tree Logic

La logique LTL ne permet pas d'exprimer des propriétés du type «il est possible dans le futur que...». Il faut pour cela avoir plusieurs futurs possibles dans le modèle de la logique. La logique temporelle arborescente CTL [31] [32] (Computation Tree Logic) permet d'exprimer des propriétés des arbres d'exécution (correspondant éventuellement à un système de transitions).

Définition (Formules de CTL)

Les formules de CTL sont les formules d'états définies inductivement par :

– $\forall\ p \in AP$, p est une formule d'état .
– si p, q sont des formules d'état, alors $p \vee q$ et $\neg\ p$ sont des formules d'état,
– si p est une formule de chemin, alors Ep et Ap sont des formules d'états,
– si p et q sont des formules d'état, alors $\times\ p$ et $p \cup q$ sont des formules de chemin.

On considère pour CTL des arbres infinis de propriétés obtenues à partir de l'arbre d'exécution d'un système de transitions étiqueté $S = (Q,\ \mathrm{s}_0,\ A,\ \longrightarrow,\ L)$: à l'arbre d'exécution $A(S)$ on associe l'arbre des propriétés a_0 étiqueté par les propriétés $L(q)$ des états q de chaque nœud de $A(S)$.

La sémantique de CTL est définie sur les arbres d'exécution. Un arbre d'exécution d'un système de transitions est complètement défini par son état d'origine et par la relation de transition $\longrightarrow$. C'est pourquoi la sémantique de CTL est définie sur les états d'un système de transitions (qui caractérise les arbres issus de ces états).

Sémantique de CTL

Soit q_0 un état de S. La sémantique de CTL est définie inductivement par :

– $p \in AP$, $q_0 \models p \longleftrightarrow p \in L(q_0)$;
– $q_0 \models p \vee q \longleftrightarrow q_0 \models p$ ou $q_0 \models q$;

- $q_0 \models \neg\, p \longleftrightarrow q_0 \models p$;
- $q_0 \models Ep \longleftrightarrow \exists \sigma = q_0 \cdots q_n \cdots$ telle que $\sigma \models p$;
- $q_0 \models Ap \longleftrightarrow \forall \sigma = q_0 \cdots q_n \cdots$ on a $\sigma \models p$;
- $\sigma \models \times$ p $\longleftrightarrow \sigma$ 1 $\models p$;
- $\sigma \models p \cup q \longleftrightarrow \exists\, j \geq 0$, σ j —$\models q$ et ($\forall\, k$ ¡ j, $\sigma\ k$ —$\models p$).

Quelques exemples de propriétés CTL sont introduits dans la figure II.8.

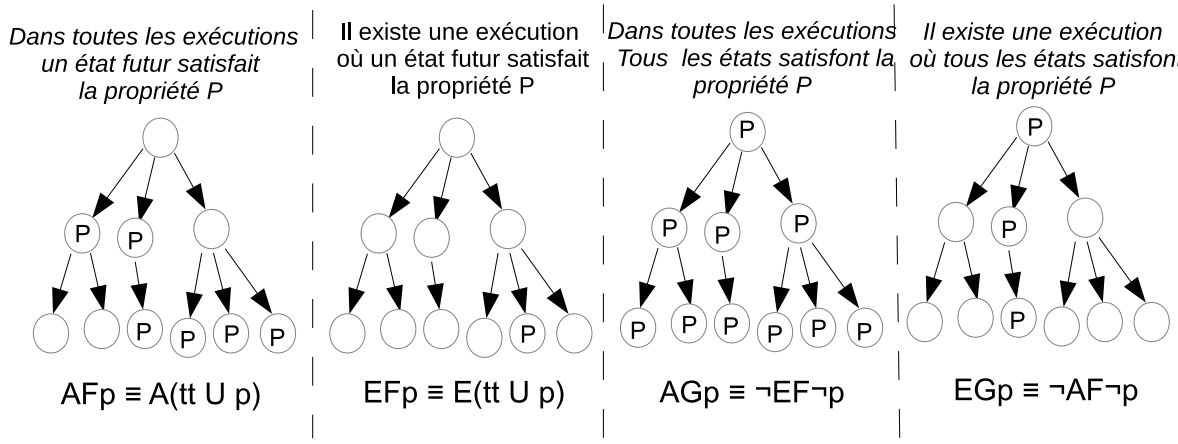


FIGURE II.8 – Abréviations CTL

II.3.3.1.3 LTL vs. CTL

L'une des différences entre CTL et LTL est que CTL ne permet pas d'exprimer l'équité. LTL par contre ne permet pas d'exprimer la notion de possibilité dans le futur. En fait LTL et CTL ne sont pas comparables.

II.3.3.1.4 CTL*

$CTL*$ est un extension de CTL qui contient aussi LTL.

Formules de CTL*

Les formules de $CTL*$ sont les formules d'états définies inductivement par :

- **(s1)** $\forall\, p \in AP$, p est une formule d'état,
- **(s2)** si p et q sont des formules d'état, alors $p \vee q$ et $\neg\, p$, sont des formules d'état,
- **(s3)** si p est une formule de chemin, alors Ep et Ap sont des formules d'état,
- **(p1)** si p est une formules d'état, alors p est une formule de chemin,
- **(p2)** si p et q sont des formules de chemin, alors $p \vee q$ et $\neg\, p$ sont des formules de chemin,
- **(p3)** si p et q sont des formules de chemin, alors $\times\, p$ et $p \cup q$ sont des formules de chemin.

La sémantique de $CTL*$ est une combinaison des sémantiques de LTL et CTL :

Sémantique de CTL*

La sémantique des formules de $CTL*$ est définie inductivement par :

- **(s1)** si $p \in AP$ est une formule d'état, $q_0 \models p \leftrightarrow p \in L(q_0)$,
- (s2) $q_0 \models p \vee q \leftrightarrow q_0 \models p$ ou $q_0 \models q$,
- **(s'2)** $q_0 \models \neg\, p \leftrightarrow q_{\,0} \models p$,
- (s3) $q_0 \models Ep \leftrightarrow \exists \sigma = q_{\,0}.\, q_n$ telle que $\sigma \models p$,
- **(s'3)** $q_0 \models Ap \leftrightarrow \forall\, \sigma = q \cdot_{\,0} \cdot\, q_n$ telle que $\sigma \models p$,
- (p1) $\sigma = q_0 \cdots q_n \cdots \models p \leftrightarrow q_{\,0} \models p$,
- **(p2)** $\sigma \models p \vee q \leftrightarrow \sigma \models p$ ou $\sigma \models q$,
- **(p3)** $\sigma \models \times\, p \leftrightarrow \sigma 1 \models p$,
- **(p03)** $\sigma \models p \cup q \leftrightarrow \exists\, j \geq 0, \sigma j \models q$ et ($\forall\, k < j, \sigma k \models p$).

II.3.4 Vérification par *CPNTools*

Pour vérifier le comportement du systèmes modélisé en $RdPCH$ en utilisant l'outil *CPNTools*, il faut générer l'espace d'états correspondant et spécifier les propriétés à vérifier sous forme des formules de logique temporelle. La logique supporté par *CPNTools* est *ASK-CTL*. En fin, vérifier ces propriétés sont valides ou non.

II.3.4.1 ASK-CTL

ASK-CTL est une variante de la logique CTL. Elle est interprétée sur les espaces d'état d'un RdP dans l'outil *CPNTools*. CTL s'étendre afin de prendre en compte, à la fois des informations d'état et de transition. *ASK-CTL* est supporté par un modele-checker implémenté dans l'outil *CPNTools* qui permet de prendre . Il prend une formule *ASK-CTL* comme argument, vérifie la formule sur l'espace d'état, et renvoie la valeur de vérité de la formule donnée.

La logique *ASK-CTL* et son *model-checker* sont implémentés en langage SML, et les requêtes sont formulées directement dans la syntaxe SML[34].

II.3.4.1.1 Syntaxe

Supposons un RdP N. La logique, qui est interprété sur l'espace d'états de N, a deux catégories de formules, formules d'états et formules de transition.

- Formules d'état :

  $A : := tt \mid \alpha \mid \neg A \mid A1 \vee A2$

  – $\mid < B >$

  – $\mid EU(\ A_1\ ,\ A_2\ )$

  – $\mid AU(\ A_1\ ,\ A_2\ )$

  Où tt est interprétée comme la valeur constante vrai, est une fonction de l'ensemble $[M \rightarrow IB]$, c'est à dire, α un mappage de fonction de marquage vers l'ensemble

des booléeans IB, et B est une formule de transition. EU et l'AU sont expliqués ci-dessous.

– Formule de transition :
  $B ::= \text{tt} \mid \beta \mid \neg B \mid B_1 \vee \text{B}_2$
  – $\mid < \text{A} >$
  – $\mid EU(\ B_1\ ;\ \text{B}_2\ )$
  – $\mid AU(\ B_1\ ;\ \text{B}_2\ )$
  Où β est une fonction de l'ensemble [BE $\rightarrow B$], à savoir, une fonction de mappage d'un éléments de liaison vers booléens, et A est une formule d'état.

La logique *ASK-CTL* contient l'opérateur temporelle U standard (jusqu'à) combiné avec les quantificateurs de chemin E (existe) et A (pour tous). Par exemple, l'opérateur $UE(\text{A}_1\ ;\ \text{A2})$ exprime l'existence d'un chemin a partir un marquage donnée vérifiant A_1 jusqu'à ce qu'un marquage est atteint à laquelle A_2 est vérifier.

À partir de ces opérateurs temporelle d'autres opérateurs sont définies. $Pos(A)$ signifie qu'il est possible de parvenir à une état où A est vérifier, il peut être définie comme $POS(A) = EXIST_UNTIL(TT, A)$. $Inv(A)$ signifie que A est vérifier dans chaque état joignable, il peut être définie comme $INV(A) = NOT(POS(NOT(A)))$. $Ev(A)$ signifie que tous les chemins, A est vérifier au sein d'un nombre fini d'étapes, il peut être définie comme $EV(A) = FORALL_UNTIL(TT, A)$.
L'ensemble des opérateurs *ASK-CTL* pour les formules d'état définie en syntaxe SML est présenté dans le listing II.1 , et celui de formules de transition est présenté par le listing II.2.

```
1 A ::= TT
2 | FF
3 | NOT(A)
4 | AND(A1,A2)
5 | OR(A1,A2)
6 | NF(<node expression>)
7 | EXIST_UNTIL(A1,A2)
8 | FORALL_UNTIL(A1,A2)
9 | POS(A)
10 | INV(A)
11 | EV(A)
12 | ALONG(A)
13 | MODAL(B)
14 | EXIST_MODAL(A1,B2)
15 | FORALL_MODAL(A1,B2)
16 | EXIST_NEXT(A)
17 | FORALL_NEXT(A)
```

Listing II.1 – Code SML des operateurs de formule d'etat

```
1 B ::= TT
2 | FF
3 | NOT(B)
4 | AND(B1,B2)
5 | OR(B1,B2)
6 | AF(<arc expression>)
7 | EXIST_UNTIL(B1,B2)
8 | FORALL_UNTIL(B1,B2)
9 | POS(B)
10 | INV(B)
11 | EV(B)
12 | ALONG(B)
13 | MODAL(A)
14 | EXIST_MODAL(B1,A2)
15 | FORALL_MODAL(B1,A2)
16 | EXIST_NEXT(B)
17 | FORALL_NEXT(B)
```

Listing II.2 – Code SML des operateurs de formule de transition

II.4 Conclusion

Dans ce chapitre, nous avons présenté le formalisme du réseaux de Petri puis les approches de vérification formelle, en focalisant sur l'approche de *model-checking* et son application sur les *RdP*. Cette méthode permet d'obtenir une très forte assurance de l'absence de bug dans les systèmes, c'est-à-dire d'acquérir des niveaux d'évaluation d'assurance élevés, de plus le *model-checking* a l'avantage d'être automatique.

À la fin de ce chapitre, nous avons présenté l'outil *CPNTools* qui est un outil puissant pour la simulation et la vérification des *RdP* grâce au *model-checker* qu'il contient, en plus la bibliothèque *ASK-CTL* qui implémente la logique CTL.

Chapitre III

Ingénierie Dirigée par les Modèles

III.1 Introduction

Ces dernières années, ont vu l'accroissement exponentiel de la complexité des systèmes informatiques, avec l'apparition de la problématique des lignes de produits logiciels, visant à minimiser simultanément les coûts de développement et le temps de mise sur le marché de nouveaux logiciels d'une même famille.

Chacun s'accorde à penser que l'ingénierie logicielle guidée par les modèles est l'avenir des applications. Bill Gates a dit que « Modéliser est le futur, et je pense que les sociétés qui travaillent dans ce domaine ont raison » [35]. L'IDM propose de modéliser les applications à un haut niveau d'abstraction et place le modèle au cœur du processus de conception, puis génère le code de l'application à partir des modèles.

Dans ce chapitre, nous allons présenter les notions et les concepts relatifs à l'ingénierie dirigée par les modèles. Une attention particulière sera mise sur la transformation de modèles et son importance dans le cadre d'une ingénierie dirigée par les modèles. Enfin, nous décrirons le langage ATL (Atlas Transformation Langage), qui est un langage de transformation de modèle.

III.2 L'Ingénierie Dirigée par les Modèles

L'Ingénierie Dirigée par les Modèles (IDM), ou Model-driven engineering (MDE), fait partie des domaines de l'informatique et plus particulièrement de l'ingénierie système, elle fournit les outils et les concepts nécessaires pour créer ou modifier des langages de modélisation. Ce domaine relativement récent à fait son apparition dans les années 80 avec un premier outil qui supportait les concepts de l'IDM, "the Computer-Aided Software Engineering" (CASE). La complexité grandissante des systèmes à modéliser a conduit les développeurs à manipuler des concepts de plus haut niveau. L'OMG a fini par s'intéresser à l'IDM, ils ont proposé en 1997 le standard MOF (Meta-Object Facility). On parle également parfois de MDA (Model Driven Architecture) qui est une marque déposée de l'OMG qui reprend et restreint les concepts de l'IDM [36].

L'IDM peut être vue comme une famille d'approche, qui se développent à la fois dans les laboratoires de recherches et chez les industriels impliqués dans les grands projets de développement logiciels. Cette approche vise non seulement à favoriser un « génie

logiciel » plus proche des métiers en autorisant une appréhension des applications selon différents points de vue (modèles) exprimés séparément Mais elle intègre également comme fondamentales de la composition et la mise en cohérence de ces perspectives. De plus, elle se veut productive en automatisant la prise en charge des outils relatifs à la validation des modèles, les transformations et les générations de codes.

III.2.1 La Modélisation

Un modèle est une simplification d'un système construit dans une intention particulière, il doit pouvoir répondre à des questions en lieu et place du système modélisé. Le modèle est une description (d'une partie) d'un système écrit dans un langage bien défini. La notion de modèle dans l'IDM fait explicitement référence à la notion de formalisme ou de langage de modélisation bien défini, ce dernier est défini par une syntaxe abstraite, concrète et 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 [35].

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 certaine de ses propriétés [37].

III.2.2 La Méta-Modélisation

Un méta-modèle est un modèle qui définit le langage d'expression d'un modèle, c'est-à-dire le langage de modélisation [38]. En pratique, un méta-modèle permet de capitaliser un domaine de connaissance, il devient naturellement le cœur d'un outillage visant à systématiser certaines étapes de développement. La notion de méta-modèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire.

Selon [35], un méta-modèle définit la structure que doit avoir tout modèle conforme à ce méta-modèle. Autrement dit, tout modèle doit respecter la structure définie par son méta-modèle. Par exemple, le méta-modèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc.

La figure III.1 illustre les trois niveaux modèle, méta-modèle et monde réel. Un modèle est dit conforme à un méta-modèle si chacun de ses éléments (objets et relations) est un instance d'un élément du méta-modèle, et s'il respecte l'ensemble des propriétés (ex. contraintes d'invariant) exprimées sur le méta-modèle.



FIGURE III.1 – Relations entre monde réel, modèle, méta-modèle

Un **méta-modèle** est simplement une description d'un langage de modélisation, il apporte les règles syntaxiques qui permettent de l'utiliser (d'écrire un modèle dans ce langage) et le sens de chaque item présent dans ce langage.

La figure III.2 inspirée de [36] représente la relation entre le système et le modèle, ainsi que, la relation entre le modèle et le méta-modèle.



FIGURE III.2 – Les relations de bases dans l'IDM.

Dans la figure III.2, la relation "représente" dénote qu'un modèle est une représentation d'un système, tandis-que la relation "instance de" dénote qu'un modèle est conforme à un méta-modèle si ce modèle appartient à l'ensemble modélisé par ce méta-modèle.

III.3 L'Architecture Dirigée par les Modèles (MDA)

L'MDA est une approche de développement proposée et soutenue par l'OMG [38]. 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 sur les modèles et leurs transformations.

MDA défini deux façons différentes de représenter les modèles, soit sous forme de documents textuels, soit sous forme d'objets de programmation. La représentation textuelle est plus adaptée au stockage des modèles sur disque dur ou aux échanges de modèles

entre applications tandis que la représentation objet est plus adaptée à la manipulation informatique (transformation, exécution, validation, etc.).

D'après [35], MDA consiste à élaborer les modèles du logique métier des systèmes de façon indépendante des plates-formes d'exécution puis à transformer ces modèles automatiquement vers des modèles dépendants des plates-formes. La complexité des plates-formes n'apparaît plus dans les modèles du logique métier, mais se retrouve dans la transformation. Les avantages de MDA sont donc la pérennisation du logique métier de l'entreprise grâce à l'élaboration de modèles, la productivité de ce logique métier grâce à l'automatisation des transformations de modèles et la prise en compte des plates-formes d'exécution grâce à l'intégration de celles-ci dans les transformations des modèles.

III.3.1 Transformation et 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 [39] :

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 III.3 établie par [40] donne une vue générale d'un processus MDA appelé couramment cycle de développement en Y en faisant apparaître les différents niveaux d'abstraction associés aux modèles.

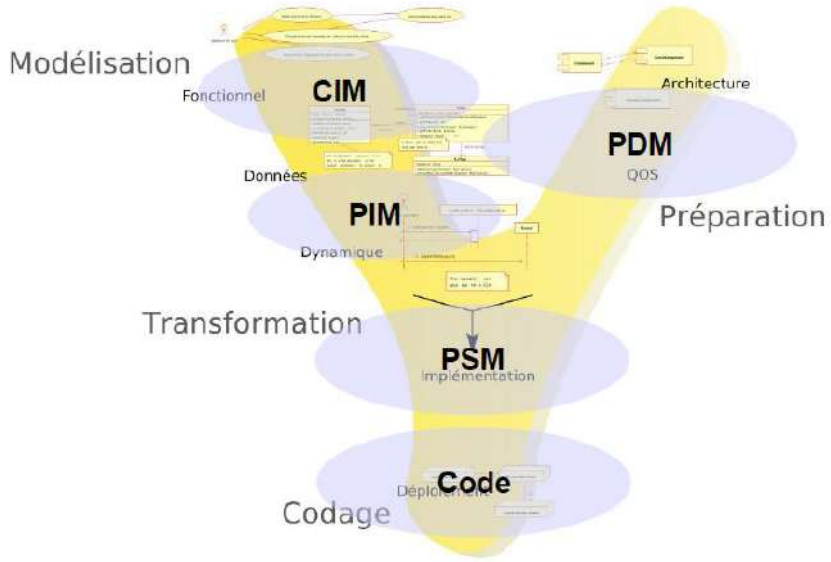


FIGURE III.3 – Le processus en Y de l'approche MDA

III.3.2 Standards de l'OMG

L'approche MDA a le mérite d'être élevée au rang de standards de l'OMG (voir la figure III.4), dont notamment OCL, MOF, XMI et CWM De [38].

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

MOF (Meta-Object Facility) 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) un standard d'échange de métadonnées.

CWM (Common Warehouse Metamodel) 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.

III.4 La 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 (voir figure III.5).

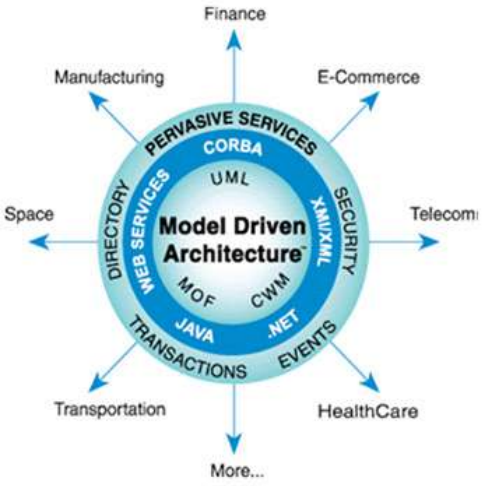


FIGURE III.4 – Les standards de l'Architecture Dirigée par les Modèles

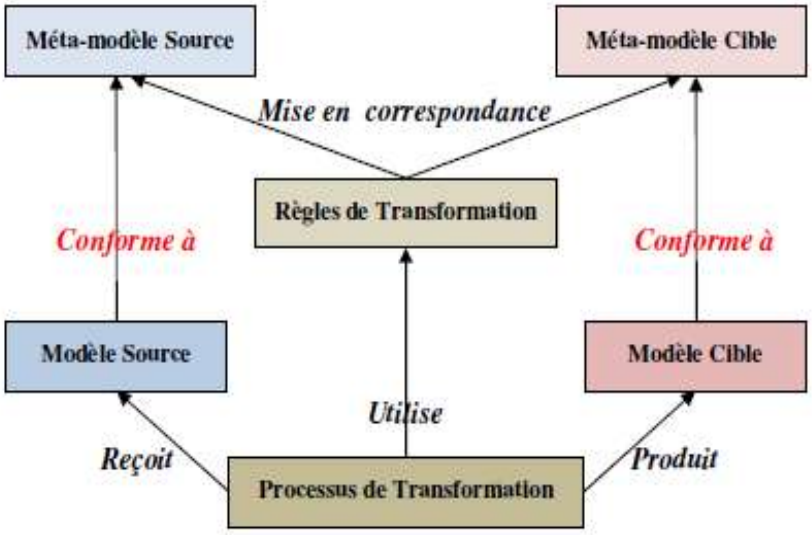


FIGURE III.5 – Concepts de base de la transformation de modèles

III.4.1 Techniques de transformations

Il y a trois grandes catégories de techniques de transformation d'après [35], l'approche déclarative, impérative et hybride.

III.4.1.1 Approche déclarative

Recherche de certains patrons (d'éléments et de leurs relations) dans le modèle source. Chaque patron trouvé est remplacé dans le modèle cible par une nouvelle structure d'élément, l'écriture de la transformation assez simple mais ne permet pas toujours d'exprimer toutes les transformations facilement.

III.4.1.2 Approche impérative

Proche des langages de programmation usuels, on parcourt le modèle source dans un certain ordre et on génère le modèle cible lors de ce parcours, l'écriture de la transformation peut être plus lourde mais permet de toutes les définir, notamment les cas algorithmiquement complexes.

III.4.1.3 Approche hybride

À la fois déclarative et impérative. La plupart des approches déclaratives offrent de l'impératif en complément car plus adapté dans certains cas.

III.4.2 Types de transformation

Un des concepts les plus importants de l'IDM est la transformation de modèle c'est à dire de partir d'une modèle "source" et aller vers un modèle "cible". L'IDM distingue deux types de transformation (voir figure III.6).

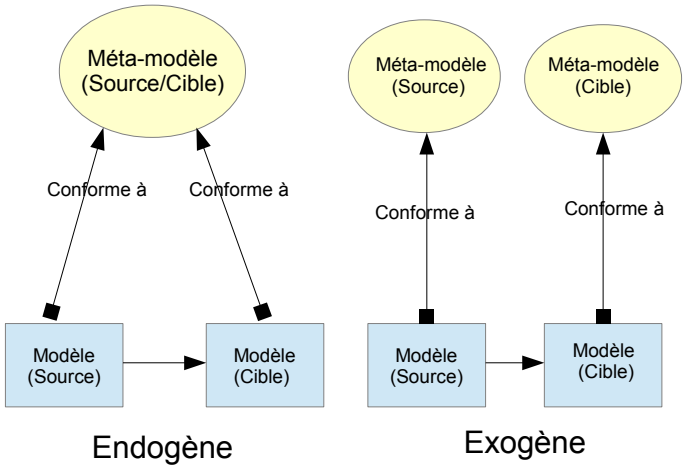


FIGURE III.6 – Les types de transformation

- La transformation Endogène : Le modèle source et le modèle cible sont conformes au même méta-modèle (e.g. : d'un diagramme SysML vers un diagramme SysML).
- La transformation Exogène : Le modèle source et le modèle cible sont conformes à des méta-modèles différents (ex : d'un SysML vers un RdP).

Il y a alors possibilité d'automatiser la transformation. Pour l'écriture des règles de transformation au niveau méta-modèle, l'OMG a défini le standard QVT (Query/View/Transformation). À ce jour les outils qui supportent ce standard ne sont pas suffisamment matures et ne sont pas encore adaptés à des applications réelles. Il existe de nombreux langages pour effectuer les transformations de modèle. Notre choix s'est porté sur le langage ATL qui est intégré à la boîte à outils POLARSYS.

III.5 Language de transformation ATL

Le langage ATL a été intégrés dans la plate-forme de modélisation *EMF*. Dans cette section on va introduire la plate-forme de modélisation EMF, ensuite, on défini le langage de transformation ATL.

III.5.1 La plate-forme de modélisation EMF

L'EMF a été conçu pour simplifier le chargement, la manipulation et la sauvegarde de modèles au sein de l'environnement Eclipse. Il repose sur un formalisme de description de méta-modèles nommé Ecore. Ce formalisme est un sous-ensemble de la norme EMOF (Essential MOF), elle-même étant un sous-ensemble de MOF2. Un aperçu d'un fichier Ecore ouvert avec son éditeur est donné par la figure III.7.

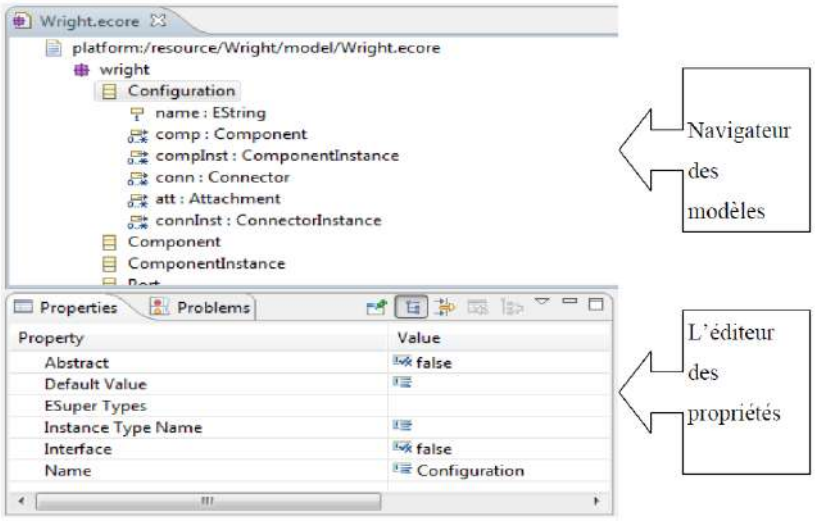


FIGURE III.7 – Un aperçu d'un fichier Ecore ouvert avec EMF

L'arbre de navigation est utilisé pour naviguer et créer de nouveaux éléments Ecore. La deuxième ligne définit le nom du paquetage. L'éditeur des propriétés est utilisé pour modifier des éléments Ecore.

III.5.2 Atlas Transformation Langage

ATL est un langage de transformation de modèles au niveau des méta-modèles plus ou moins inspiré par le standard QVT de l'OMG. Il est développé depuis 2003 au sein de l'Université de Nantes par les groupes (INRIA et LINA) dans le cadre de IDM. ATL fait partie de la section M2M (Model-To-Model) d'Eclipse, il est donc recommandé en tant qu'outille de transformation de modèle à modèle. De plus, il fournit des moyens pour produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources, il a pour vocation de permettre l'expression des règles de transformation de modèle et de pouvoir l'exécuter. ATL se compose d'un langage de transformation, un compilateur, une

machine virtuelle et un IDE s'appuyant sur Eclipse pour but de faciliter le développement de transformation.

ATL possède la caractéristique d'être un langage hybride. Ce caractère vient du fait que c'est un langage de transformation à la fois déclaratif et impératif. Ainsi il est possible d'écrire des règles de correspondances entre deux méta-modèles (matched rules déclaratives) mais aussi de faire appel à certaines règles au sein d'autre règles (called rules impératives). ATL intègre le langage OCL qui permet d'écrire sous forme de code des pré et/ou post conditions pour une opération et notamment d'insérer certaines contraintes à l'application d'une règle.[35]

Ainsi, la transformation de modèle via des règles de transformations écrites au niveau méta-modèle, peut être appliquée à notre problématique en utilisant ATL comme langage de transformation et les outils développés autour de ce langage. Cela nécessite alors l'exploitation d'un ou plusieurs méta-modèles adéquats.

III.5.3 Programmation d'un module ATL

Un fichier de module ATL commence par la section d'en-tête et peuvent contient des importations ainsi que des helpers, et se termine par les règles, la figure III.8 représente la structure du langage ATL. Le module d'ATL est stockée dans un fichier avec l'extension "atl". Voir le manuel de l'utilisateur ATL [41] pour plus de détails.

- Si un identifiant est en conflit avec un mot-clé, il doit être marqué avec des apostrophes.
- L'analyseur d'ATL est sensible à la casse. Cela concerne les noms de fichiers autant que le code source elle-même.
- Les commentaires sont signalés par des tirets doubles "–".

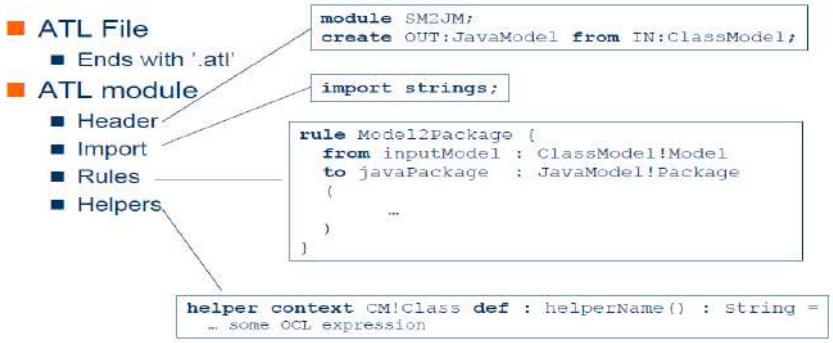


FIGURE III.8 – La structure du langage ATL

III.5.3.1 La section d'en-tête

La section d'en-tête (header en anglais) définit le nom du module de transformation ainsi que le nom des variables correspondantes aux méta-modèles sources et cibles. Elle

encode également le mode d'exécution du module qui peut être soit le mode normal (défini par le mot clef *from*) ou le mode de raffinage (défini par le mot clef *refining*). La syntaxe de la section d'en-tête est définie comme suit :

```
1 module Source2Cible;
2 create OUT : MMCible from IN : MMSource1, IN1 : MMSource2;
```

Le mot clé *module* définit le nom du module. Le mot-clé *create* introduit la déclaration des modèles sources et cibles.

III.5.3.2 La section d'importation

La section d'importation déclare ce que les bibliothèques doivent être importées. Le code source *ATL* suivant représente l'en-tête d'importation de la bibliothèque Strings Par exemple, :

```
1 uses strings;
```

Le mot clé *uses* déclare les bibliothèques qui doivent être importés. Il peut y avoir plusieurs importations mais il faut pas déclarer les importations qui ne sont pas nécessaires. La bibliothèque *string.atl* contient de nombreuses fonctions utiles pour les chaines de caractères, mais pas nécessaire pour l'exemple de *Source2Cible*.

III.5.3.3 Les helpers

Les helpers peuvent être utilisés pour définir des variables et des fonctions (globales) et ce sont des expressions OCL. Ils peuvent appeler les uns les autres (récursion est possible) ou ils peuvent être appelés à partir des règles. En général, ils servent à définir des pièces répétitives de code en un seul endroit. Les fonctions *helper* commence par le mot-clé *helper* et se termine par un point-virgule. Entre les deux est l'expression OCL, comme présente l'exemple suivant.

```
1 helper def: objectIdType : Relational!Type =
2   Class!DataType.allInstances()->
3   select(e | e.name = 'Integer')->first();
```

Listing III.1 – Structure de helper dans le langage ATL

La notion de terme contexte est similaire à OCL, elle peut être comparée à un paramètre d'entrée d'une méthode. Une variable de contexte est définie à l'aide de chemin d'expression ATL *metamodele*!*element*, il est accessible via la variable lui-même. Si le contexte n'est pas spécifié, le module lui-même il prend comme contexte. Le nom de la fonction est introduit avec *def* : est suivie par des crochets. Le type de retour est entre un colon et un signe égal qui sert de point de départ pour la mise en œuvre de la fonction, cette dernière se termine par un point-virgule

III.5.3.4 Les règles

Les règles (rules en anglais) décrivent la transformation d'un modèle source à un autre modèle cible en rapportant un méta-modèle. Il est introduit par le mot-clé *rule* qui est suivi par le nom de la règle, ce nom doit être unique dans le module. La mise en œuvre est entourée par des accolades.

Dans la configuration de la source (la partie *from*), le type du modèle source qui va être transformé doit être déclaré. Cette partie est constituée du mot-clé *from* dans la déclaration des variables et éventuellement d'un filtre, ce dernier est une expression OCL limiter la règle à des éléments du modèle source satisfaire certaines contraintes. Le filtre est écrit derrière la déclaration de variable et entouré par des parenthèses.

Dans la partie cible *to* se trouve la déclaration des éléments du modèle cible ou doit être transformé. Le motif de cette partie déclare les détails de la transformation. Un élément de motif cible est constituée d'une déclaration de variable et une séquence de liaisons. Ces liaisons sont constituées principalement de constructions fléchées à gauche. Habituellement, un attribut du modèle cible (sur le côté gauche de la flèche) reçoit une valeur de retour d'une expression OCL (sur le côté droit de la flèche) qui est basé sur la au variable. En ce sens, la partie droite de la flèche peut être constitué d'un attribut de la variable ou dans un appel à un assistant fonction (qui est une expression OCL).

```
1 rule NomDeLaRegle
2   from
3     i: NomMMSource!NomMetaClasseSource
4   to
5     o: NomMMCible!Meta-ClasseCible
6   Attribut1 <- i.AttributA,
7   Attribut2 <- i.AttributB+i.AttributA,
```

Listing III.2 – Structure d'une règle dans le langage ATL

Les affectations sont séparées par des virgules ",". La dernière ligne dans un bloc de déclarations ne se termine pas par une virgule, et l'attribution de valeurs est réalisée avec la flèche gauche opérateur " $<-$".

III.5.3.5 Les modes d'exécution des modules

Le moteur d'exécution ATL définit deux modes d'exécution pour les différents modules ATL.

- Le mode normal : c'est le mode d'exécution par défaut, il faut spécifier explicitement la manière dont les éléments de modèle cible doivent être générés à partir des éléments du modèle source. Ce mode est spécifié par le mot clef *from* dans l'en-tête. Il est utilisé dans le cas d'une transformation **exogène**
- Le mode de raffinage : dans ce mode d'exécution la conception de la transformation qui vise à dupliquer le modèle source avec seulement quelques modifications. Concevoir cette transformation dans le mode d'exécution par défaut nécessite la

spécification des règles, aussi bien pour les éléments qui vont être modifiés, que ceux ne vont d'être modifiés. Ce mode est spécifié par le mot clef $refining$ dans l'en-tête. Il est utilisé dans le cas d'une transformation **endogène**

III.6 Conclusion

Dans ce chapitre, nous avons défini les principaux concepts du domaine de l'ingénierie des modèles en mettant en évidence le rôle central des modèles. Puis, 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 et au développement des systèmes complexes.

Chapitre IV

Méthodologie

IV.1 Introduction

Dans le cas de conception des systèmes ayant des fonctions critiques, il est nécessaire de vérifier le plus tôt possible que les choix de conception du système sont cohérents avec les objectifs visés. Pour ce faire, une première étape de modélisation du système est nécessaire. Pour cette étape, nous proposons *SysML* comme langage de spécification. Ensuite, la deuxième étape consiste à vérifier que la conception est conforme aux exigences. Pour cette deuxième étape, nous proposons de transformer les diagrammes d'activités en *RdPCH* pour permettre la vérification des exigences *SysML*. Après, la translations de ces derniers en formules de logique temporel. La solution peut donner des résultats très satisfaisants si les besoins des utilisateurs sont rigoureusement et complètement capturés. Mais, l'opération est compliqué puisque les développeurs n'utilise pas le même vocabulaire sur le comportement du système, qui est l'aspect le plus essentiel à prendre en compte. À travers ce chapitre, nous expliquons la démarche globale que nous proposons pour pouvoir vérifier des exigences sur les *DA*s.

IV.2 Besoin de formalisation des spécification SysML

Certains organismes s'intéressent de plus en plus à la normalisation, c'est le cas de l'OMG. Cette association qui est à l'origine de langage de modélisation comme *UML*, a défini le langage *SysML*[1] qui se veut orienté système. Cette notation *SysML* est un langage de modélisation pour les systèmes très expressif, car elle est riche en terme de ces différents modèles et diagrammes. En revanche, ce langage est semi-formel, si on parle par exemple de DA, et parfois totalement informel lors de l'expression des exigences par le langage naturel au niveau de diagramme d'exigence. En conséquence, et afin de pouvoir vérifier des spécifications *SysML*, il est nécessaire de passer à des spécifications formelles. Comme présenté dans la figure IV.1, un DA peut être lié à une exigence *SysML* par la relation $<< verify >>$, ce qui signifie que le DA sera utilisé pour vérifier cette exigence.

La figure IV.1 représente la relation $<< Verify >>$ entre l'exigence *SysML* *"Requirement_1"* et le DA "*SysAct*", qui désigne que : "*SysAct*" permet de déterminer

1. System Modeling Language

si *"Requirement_1"* est réalisée ou non. Alors, dans notre cas, nous somme intéressés par cette relation de vérification.

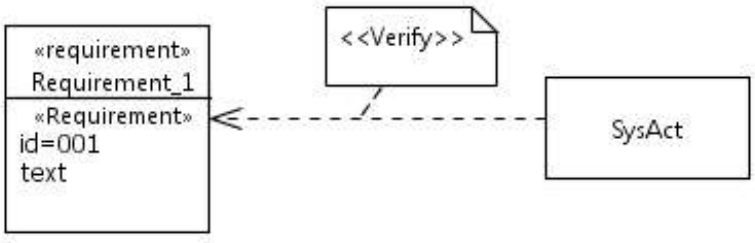


FIGURE IV.1 – Exigence *SysML* édité avec Papyrus

IV.3 Démarche générale

La démarche que nous avons proposé est bien présenté par la figure IV.2, elle prend comme entrées un DA qui doit être valide syntaxiquement, et un DE qui comporte une ou plusieurs exigences liées aux *DA*. La première étape consiste à formaliser les exigences *SysML* qui sont décrits en langage naturel. Pour ce faire, nous utilisons le langage *ActRL*[2] proposé dans [42] qui peut être utilisé par les concepteurs *SysML* pour exprimer des exigences. Par la suite, un chapitre complet sera consacré pour détailler ce langage, ainsi que son implémentation qui nous donne un outils permettant l'édition des expressions *ActRL*. La deuxième étape consiste à passer d'une spécification *SysML*(DA et DE "*ActRL*") vers une spécifications formelle. Cette étape se base sur une approche IDM pour passer d'un modèle à un autre.

La première transformation concerne le DA, elle permet de le translater en *RdPCH*, elle est implémentée à l'aide de langage de transformation ATL. Le *RdPCH* dérivé sera en format *PNML*[43]. À fin de permettre son analyse par *CPNTools*, nous proposons de générer à partir du *RdPCH* en format *PNML* un *RdPCH* en format adaptée au *CPNTools*, (voire le coté droit de la figure IV.2). En parallèle, nous abordons la transformation des expressions *ActRL* vers des formules de la logique *ASK-CTL* [44], une variante de la logique CTL implémentée dans *CPNTools*.

Enfin, lorsque toutes les transformations sont achevées, on peut entamer la vérification des expressions *ASK-CTL* sur l'espace d'état généré depuis le *RdPCH* à l'aide de l'outil *CPNTools*. Nous obtenons le résultat de vérification. Touts ces étapes seront détaillées dans les deux prochains chapitres V VI.

2. Activity requirements langage

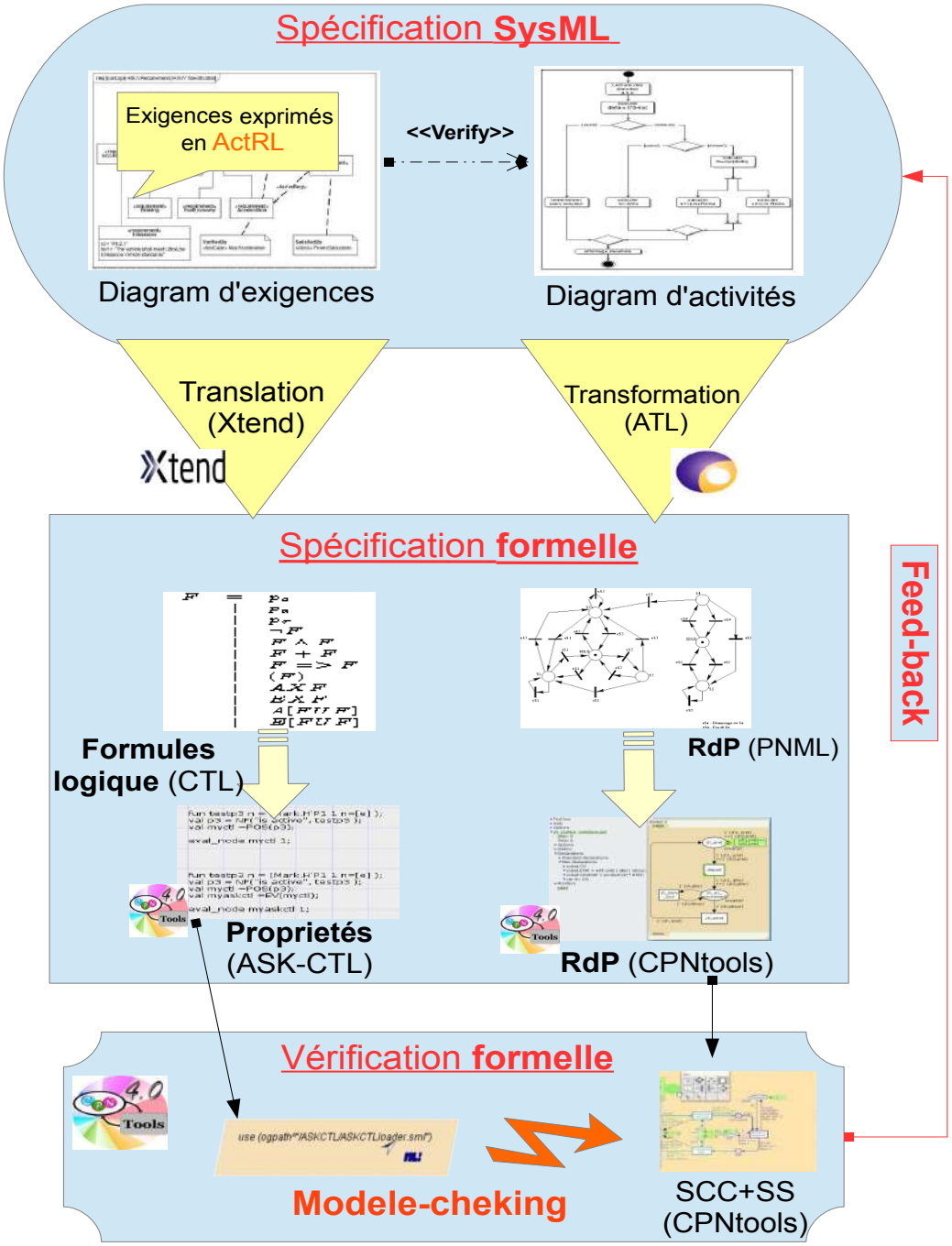


FIGURE IV.2 – Démarche générale

IV.4 Conclusion

Dans ce chapitre, nous avons présenté par une vue globale la démarche générale pour la vérification des exigences *SysML* sur les *DA*s. Nous avons cité les différentes étapes de notre démarche en expliquant brièvement les différentes translations à effectuer pour aboutir à l'étape de vérification. Dans ce qui suit, nous allons détailler ces différentes étapes.

Chapitre V

Langage ActRL pour l'expression des exigences SysML

V.1 Introduction

Dans ce chapitre, nous décrirons le langage ActRL, un langage qui permet d'exprimer des exigences qui concernent les diagrammes d'activité. Dans la première partie de ce chapitre, nous présentons le principe et la grammaire de ce langage. Puis, nous présentons l'outil que nous avons proposé pour permettre l'édition des expressions ActRL.

V.2 Langage ActRL

Les Formalismes comme la logique temporelle linéaire (LTL) et la logique arborescente (CTL) ont reçu beaucoup d'attention dans la communauté des méthodes formelles pour décrire les propriétés temporelles des systèmes. Cependant, la plupart des ingénieurs ne connaissent pas ces langages formels. De plus, ils nécessitent beaucoup d'efforts afin de maitriser ces langages. ActRL est un langage proposé en [42], il permet d'exprimer et de spécifier les exigences SysML à vérifier sur des diagrammes d'activité.

V.2.1 Grammaire du langage

Dwyer [45] a proposé une approche basée sur les patrons (patterns). Cette approche utilise des patrons spécifiques, à un niveau d'abstraction plus élevé. L'idée principale est que la propriété temporelle sera une combinaison d'un patron et d'un scope. Ce dernier est la partie qui décrit le chemin d'exécution du système sur lequel le patron sera applicable.

Patterns : il y a huit patterns qui sont organisées sous une classification sémantique, dans notre grammaire on s'intéresse au cinq suivants (côté gauche de la figure V.1).

Les paternes d'occurrence sont :

- Universalité (universality) : un état est permanent.
- Absence (absence) : un état ne se produit jamais.
- Existence (existence) : un état se produit au moins une fois.

Les paternes d'ordre sont :

- Précédence (precedence) : un état P est toujours précédée par un état Q.
- Réponse (response) : un état P est toujours suivi par un état Q.

Scopes : il y a cinq types de champs (côté droit de la figure V.1) :

- globalement (globally) : couvre toute l'exécution.
- Avant (before) Q : couvre l'exécution du système jusqu'à la première occurrence de Q.
- Après (after) Q : couvre l'exécution du système après la première occurrence de Q.
- entre (between) Q et R : couvre des intervalles de temps de l'exécution du système à partir d'une occurrence de Q à la prochaine occurrence de R.
- Après (after) Q jusqu'à (until) R : est même que la précédente dans laquelle R peuvent ne pas se produire.

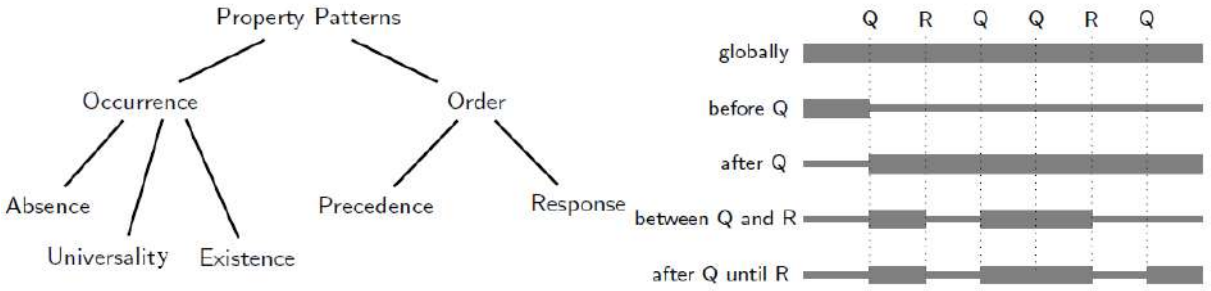


FIGURE V.1 – Les patterns et scopes de Dwyer

La grammaire du langage ActRL a été définie selon les règles EBNF (Extended Backus-Naur Form) dans [42]. Dans ces règles, les non-terminaux désignés entre <> sont des expressions de prédicat qui concerne les élément du DA. La formulation de ces expressions sera détaillée dans la suite de ce chapitre. Les règles EBNF sont proposé comme suit :

```
1 <AcTRL>::= <pattern> <scope>
2     <pattern>::= always <Pred-exp>
3       | never <Pred-exp>
4       | eventually <Pred-exp>
5       | <Pred-exp>  precededing <Pred-exp>
6       | <Pred-exp> following <Pred-exp>
7     <scopes> ::= globally
8       | before <Pred-exp>
9       | after <Pred-exp>
10      | between <Pred-exp> and <Pred-exp>
11      | after <Pred-exp> until <Pred-exp>
```

Listing V.1 – Les règles EBNF du langage ActRL

V.2.2 Expression Prédicat

Dans cette section, nous présentons l'ensemble des expressions de prédicat qui peuvent caractériser les éléments d'un diagramme d'activité au cours de son exécution. La formulation de ces expressions de prédicats sont proposée dans [42]. Elles sont comme suit :

1. Si $actionName$ est une action de l'activité $ActivityName$, alors $[ActivityName].[actionName].isActive()$ est une expression de prédicat valide. Sa valeur est $True$ (vrai) sur un état donné si actionName est en exécution.
2. Si $actionName$ est une action de l'activité $ActivityName$ et $ctlfName$ un flux de contrôle entrant de $actionName$, alors $[actionName].incoming[ctlfName].isNotEmpty()$ est une expression de prédicat valide. La même expression peut être définie pour un flux de contrôle sortant.
3. Si $actionName$ est une action de l'activité $ActivityName$, et $PinName$ est sa pin d'entrée, alors $[actionName].input[PinName].isNotEmpty()$ est une expression de prédicat valide. La même expression peut être définie pour un pin sortant.
4. Toutes les expressions OCL booléennes sur les objets manipulés dans l'activité $ActivityName$ sont des expressions de prédicat valides.
5. Toutes les expressions booléennes sur les variables locales utilisées dans $ActivityName$ sont des expressions de prédicat valides.
6. Si $actExp$ est une expression de prédicat valide, alors $not(actExp)$ est expression de prédicat valide.
7. Si $actExp1$, $actExp2$ sont des expressions valides, alors $(actExp1)et(actExp2)$ et $(actExp1)ou(actExp2)$ sont des expressions valides.

V.3 Édition des expressions ActRL

Comme une étape préliminaire à ce travail, nous avons proposé de formaliser les exigences SysML, et nous avons développé un outil pour faciliter l'édition des exigences qui concernent les diagrammes d'activités. Dans cette section, nous allons présenter l'implémen--tation de l'éditeur des exigences ActRL avec l'outil XText.

V.3.1 Language spécifique au domaine (DSL)

Un langage spécifique au domaine (en anglais « Domain Specific Language » ou DSL) fournit une notation orientée à ce domaine d'application. Il est basé sur les concepts et les fonctionnalités du domaine en question. En tant que tel, un langage dédié est un moyen efficace pour décrire et générer des programmes dans un domaine spécifique. Il permet au programmeur de s'assurer de construire un langage approprié et que ce langage répond aux exigences spécifiques du domaine ciblé. [46].

V.3.2 Vue d'ensemble de l'outil XText

XText permet le développement des grammaires DSLs et d'autres langages textuels en utilisant une grammaire qui ressemble au langage EBNF. XText est une composante de TMF (Textual Modeling Framework) intégré dans le framework de modélisation d'Eclipse

EMF. Il permet de générer un méta-modèle Ecore, un analyseur syntaxique (parseur) basé sur le générateur ANTLR (ANother Tool for Language Recognition) ou Java et un éditeur de texte sous la plate-forme Eclipse afin de fournir un environnement de développement intégré IDE spécifique langage. La figure à ceV.2 fournit une vue d'ensemble de l'outil XText. La forme textuelle du DSL constitue le point de départ. La grammaire du DSL est le point d'entrée de l'outil. XText permet de considérer la forme textuelle du DSL comme un modèle conforme à la grammaire d'entrée, donc au méta-modèle généré.

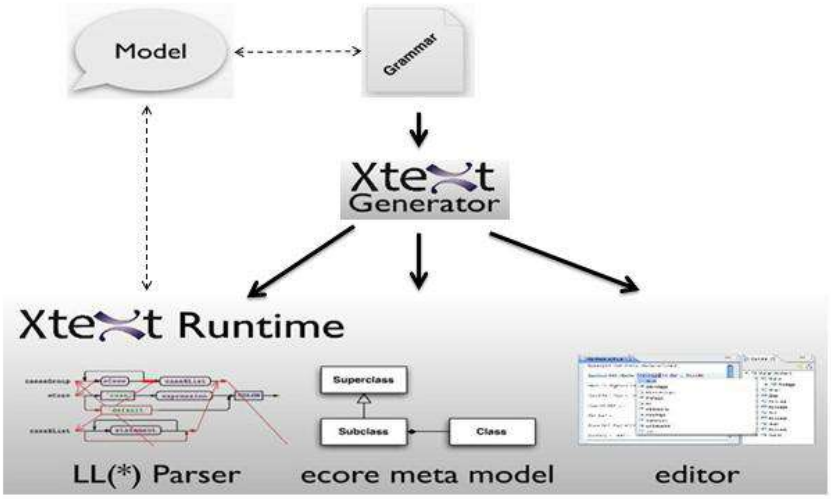


FIGURE V.2 – Démarche générale de fonctionnement de l'outil xtext

V.3.3 Grammaire ActRL avec XText

Pour définir la grammaire de ActRL dans XText, nous créons un fichier d'extension *.xtext* qui va contenir l'ensemble des règles permettant de générer l'analyseur syntaxique et l'outil d'édition des expression ActRL. Ce fichier doit débuter par les deux lignes suivantes :

```
1 grammar org.xtext.example.actrl.ActRL with org.eclipse.xtext.common.
    Terminals
2 generate actRL "http://www.xtext.org/example/actrl/ActRL"
```

Listing V.2 – Les deux lignes de début de ficher Xtext

La première ligne déclare l'identificateur du modèle et la base des déclarations, et la deuxième ligne est la directive de création du méta-modèle de notre DSL généré avec son emplacement.

V.3.3.1 Importation de méta-modèle

Les expressions du langage ActRL sont formulés sur les diagrammes d'activités pour permettre un aide contextuel lors de l'édition d'une expression ActRL, il est nécessaire

de pouvoir la lier à son contexte qui est le DA, pour cela nous devons importer le méta-modèle UML pour obtenir les entités du DA. L'importation de méta-modèle UML et celui de son dépendance ce fait comme suit :

```
1 import "http://www.eclipse.org/uml2/4.0.0/UML" as uml
2 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

Listing V.3 – La partie d'importation

V.3.3.2 La règle Model

Une grammaire avec XText est définie par un ensemble de règles. Une règle décrit l'utilisation d'un ensemble de règles terminales et non terminales. Les règles telles qu'ID sont des règles terminales qui représente les mots clés. Nous définissons la grammaire ActRL par l'ensemble des règles ci-dessous.

Dans une grammaire, la première règle (dans notre cas *Model*) s'appelle la règle d'entrée. Elle est toujours utilisé comme règle de démarrage et peut être vu comme une règle mère de toutes les autre règles. Pour la grammaire ActRL, la règle modèle est le contenneur de toutes les autres règles, zéro ou plusieurs règles ActRL. les deux règles model et ActRL sont définis dans les listings suivants :

```
1 Model:
2 elements+=ActRL*
3 ;
```

```
1 ActRL:
2     "req" ID=[uml::Class|FQN]":" (p=pattern  s=scope)
3 ;
```

Listing V.4 – La déclaration des deux règles Model et ActRL

La règle *ActRL* signifiée qu'une expression ActRL commence toujours par le mot clé *req*, ensuite le nom de l'exigence, puis la règle *pattern* suivi par la règle *scope*. Pour comprendre plus la grammaire, l'EMF fournit une représentation graphique (View) appelée "XText Syntax Graph", La figure V.3 est la représentation graphique des règles *Model* et *Actrl*.

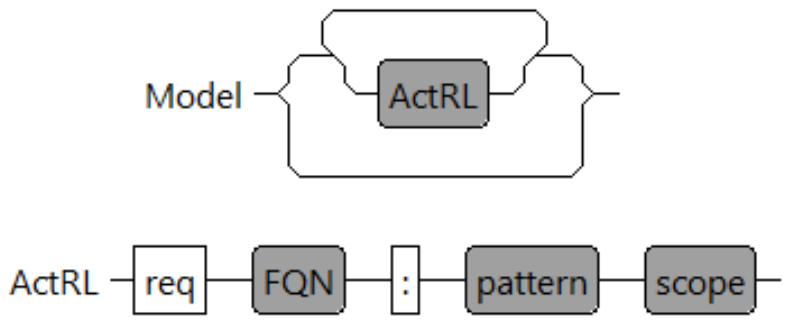


FIGURE V.3 – La représentation graphique des règles *Model* et *Actrl*

V.3.3.3 Les règles pattern et scope

Les deux règles *pattern* et *scope* contiennent les mots clés de grammaire (par exemple *always*, *never* .. etc). L'opérateur d'affectation « = » dénote qu'il s'agit d'une règle terminale de type chaine de caractères. Les deux règles sont déclarée comme suit :

```
1 pattern:
2     ((("always"|"never"|"eventually")(predex=predex))|((predex1=predex("
      preceding"|"following")predex0=predex)) )
3 ;
4 scope:
5     ((("globally")|("before"|"after") predex=predex)|('between'predex1=
      predex'and'predex0=predex)|('after'predex3=predex'until'predex2=predex))
      )
6 ;
```

Listing V.5 – La déclaration des règles pattern et scope

La représentation graphique de la règle *pattern* est présentée par la figure V.4 :

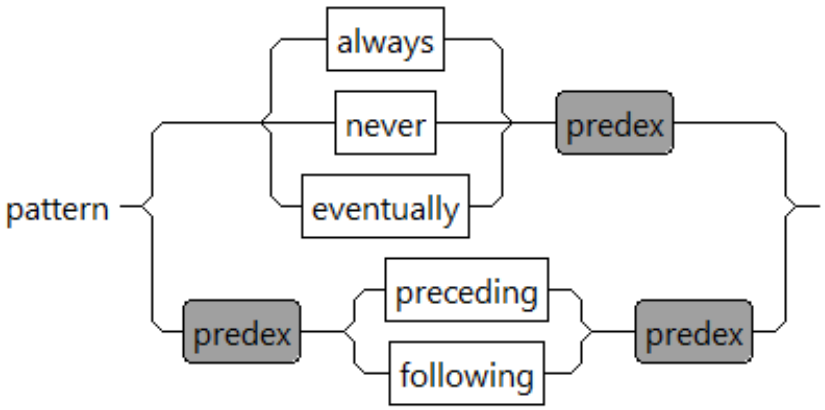


FIGURE V.4 – La représentation graphique de la règle pattern

La règle *pattern* indique via l'opérateur d'alternance « | » qu'un type peut être commencé soit par un des mots clés (*always* ou *never* ou *eventually*) suivi par la fonctionnalité nommée *predex* qui est une référence croisée d'un identificateur qui est analysé par une règle appelée *predex*. La valeur retournée par l'appel à *predex* est attribué (=) au nom de la fonction. La représentation de *scope* est dans la figure V.5 :

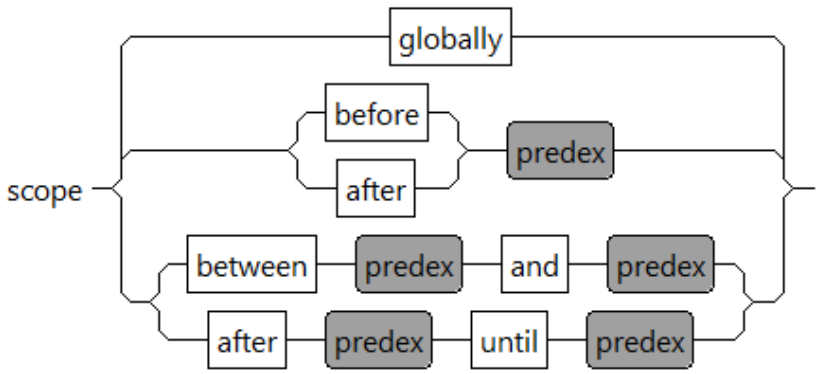


FIGURE V.5 – La représentation graphique de la règle scope

V.3.3.4 La règle PredEx :

La règle *predex* est la règle où les expressions de prédicat sont déclarer. Comme il est difficile de citer toutes les règles pour la définition des expression des prédicats, nous présentons ici qu'une partie de ces règles.

```
predex :
  '{'exp0=not (('and'|'or') exp1+=not )*'}'
;
not:
  '(' (('not(' exp ')')|(exp))  ')'
;
exp:
  //predicate expressions
    ('['ref1=[uml::Action|FQN]']''.''isActive()')|
    ('['ref2=[uml::Action|FQN]']''.'('incoming'|'outgoing') '['ref3=[uml::
    ControlFlow|FQN]']''.''isNotEmpty()')|
    ('['ref5=[uml::Action|FQN]']''.'('input'|'output')      '['ref4=[uml::
    InputPin|FQN]']'    '.''isNotEmpty()')|

    (ref6=[uml::LiteralBoolean|rule] )
;
```

Listing V.6 – La déclaration des règles des expressions de prédicat

La règle *predex* représente la définition des expressions de prédicat en utilisant les opérateurs NOT, OR et AND. La règle *exp* permet de définir une expression de prédicat atomique sur les éléments du DA, cette règle se base sur le référencement du méta-modèle du DA, afin de pouvoir utiliser les éléments du DA sur lequel on veut exprimer notre exigence. La figure V.6 donne une visualisation des règles des expressions de prédicat.

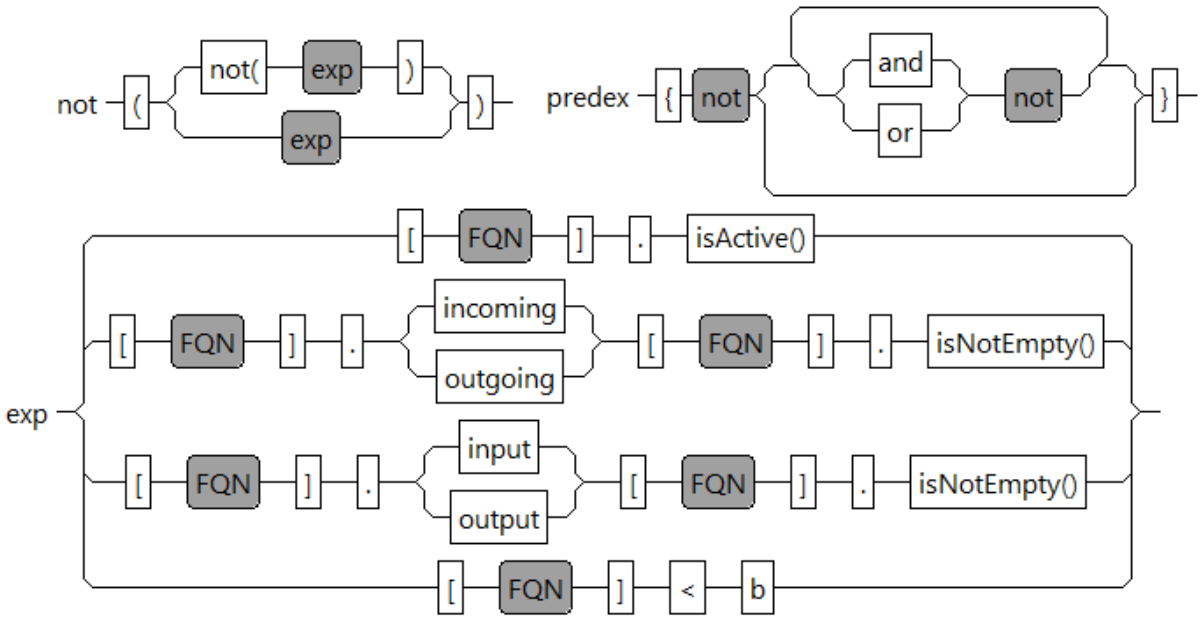


FIGURE V.6 – La représentation graphique des éléments des expressions de prédicat

Après avoir généré la grammaire, nous pouvons avoir un éditeur pour notre DSL ActRL, qui permet une coloration syntaxique des mots clés et une validation du code en temps réel. La figure V.7 donne un aperçu de l'exemple présenté avec l'éditeur ActRL.

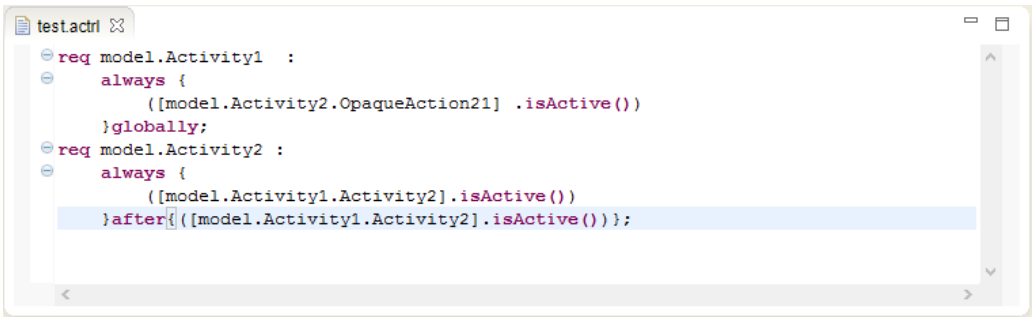


FIGURE V.7 – Un aperçus sur exemple de modèle dans l'éditeur ActRL

V.4 Intégration des expressions Actrl au diagrammes d'exigences

Les utilisateurs de l'éditeur ActRL peuvent écrire de nombreuses expressions (exigences), après la terminaison de l'écriture et du corrections de toutes les erreurs syntaxiques possibles et la sauvegarde du fichier qui contient tout les expressions, chaque exigence va être injecté dans l'exigence appropriés dans le diagramme d'exigence grâce au parseur qui peut prendre les exigences et parzer le fichier de DE pour les mettre dedans.

Pour comprendre plus cette tache, la figure V.8 explique par un exemple comment chaque exigence va être injectée au niveau du diagramme d'exigence.

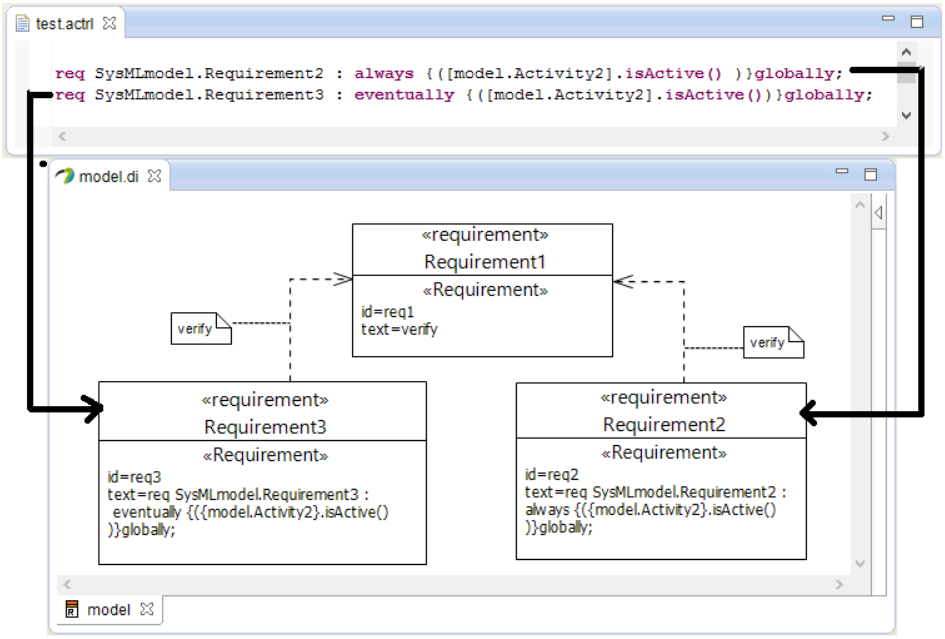


FIGURE V.8 – Un exemple d'intégration des exigence écrites par ActRL dans le DE

V.5 Conclusion

Dans ce chapitre, nous avons présenté le langage ActRL qui est spécifique pour l'expression des exigences SysML sur les diagrammes d'activités. Ainsi, nous avons implémenté un outil qui permet l'édition des exigences SysML en langage ActRL. cet outil est développé à travers le framework XText fournie avec EMF. Cette étape est une phase préliminaire permettant d'automatiser par la suite la vérification des exigences SysML sur les diagrammes d'activités en passant vers des spécification formelle.

Chapitre VI

Translation des spécifications SysML en spécifications formelle en vue de leur vérification

VI.1 Introduction

Comme nous l'avons cité dans le chapitre IV, ce chapitre consiste à détailler la translation des spécifications *SysML* en spécifications formelles. Les *DA* seront translatés en réseaux de Petri, et les exigences *SysML* en formules de logique *CTL*. Le but de cette transformation est de permettre la vérification formelles des exigences *SysML* sur les *DA*s.

VI.2 Transformation des DA vers les RdP

Notre démarche de transformation des *DA*s en réseaux de Petri est décrite par la figure VI.1, elle se base sur l'approche *IDM* (schéma CIM-PIM-PSM et l'utilisation d'outils de transformation des modèles). Dans la suite de cette section, nous allons détailler cette démarche de transformation en mentionnant les éléments de base et en détaillant les règles de transformation. Enfin, nous proposons une adaptation des modèles *RdP* issus de la transformation pour pouvoir exploiter l'outil *CPNTools* dans le processus de vérification.

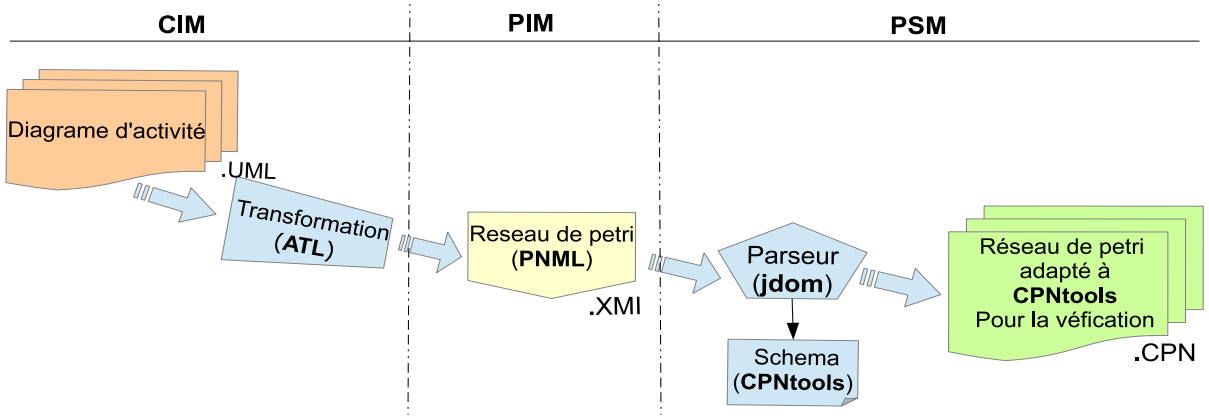


Figure VI.1 – Démarche générale de transformation de DA vers RdP

VI.2.1 Méta-modèles

La première étape dans la transformation que nous proposons consiste à définir un méta-modèle pour les *RdP* et un méta-modèle pour les *DA*s. Pour arriver a implémenter les règles de transformations d'un modèle vers un autre, il faut d'abord bien citer les méta-modèles cibles et sources. Comme nous avons deux formalismes différents, nous somme dans le cas d'une transformation exogène (Voir chapitre III).

VI.2.1.1 Métamodèle source (diagramme d'activité)

Comme nous avons vue dans le chapitre I, le *DA* est composé de nœuds et d'arcs. Ces nœuds sont divisés en deux parties, les nœuds de contrôle et les nœuds d'activités. Chacun de ces nœuds est ensuite spécialisé et peut grâce aux arguments associés avoir un comportement sensiblement différent, plus de détails sur le *DA* sont présentés dans la spécification *OMG* [47].

Le méta-modèle du *DA* (*MMDA*) que nous allons utilisé est extrait de la spécification *SysML* et *UML* [48] de l'OMG. En effet, l'OMG spécifie le *DA SysML* en ce basant sur celui d'*UML*. Il faut donc exploiter des informations des deux spécification. La figure VI.2 présente une traduction en diagramme de classe *UML* du *MMDA* (**simplifié**) présent dans la spécification *OMG*.

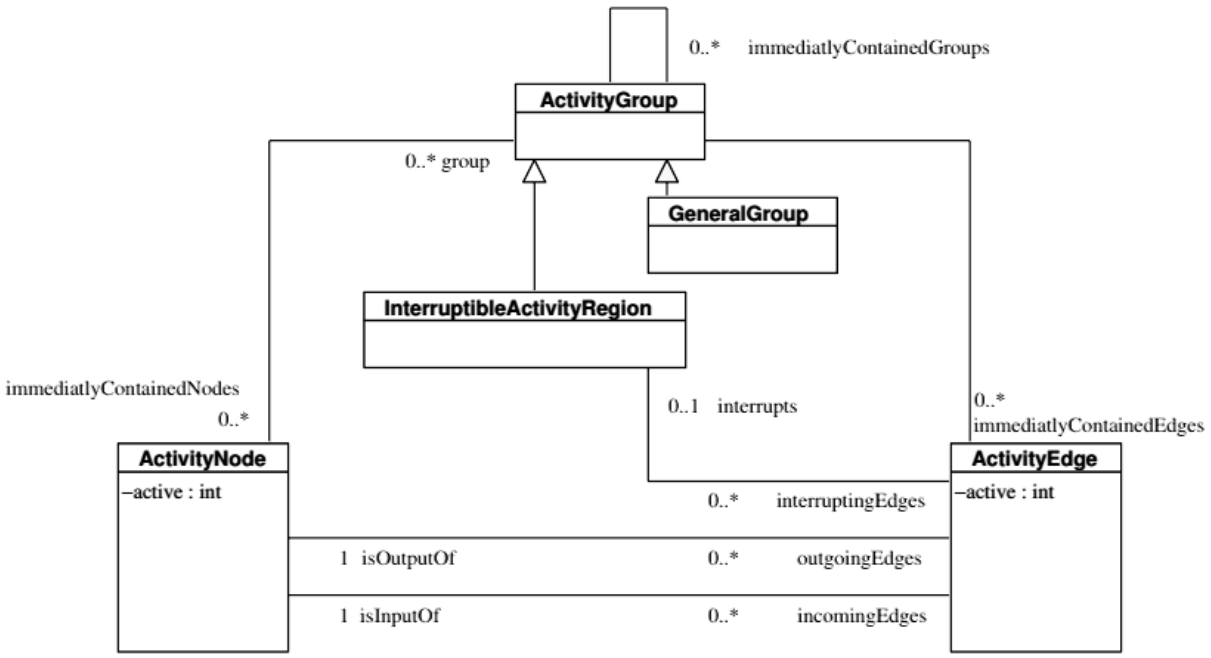


FIGURE VI.2 – La partie supérieure de méta-modèle AD

Le méta-modèle *MMDA* a été implémenté dans la boite à outils *POLARSYS* avec *EMF*(voir figure VI.7). Nous avons jugé plus intéressant d'utiliser cette implémentation

au lieu de créer notre propre méta-modèle. De plus, après vérification, nous avons constaté que ce *MMDA* est conforme à la spécification *OMG*.

VI.2.1.1.1 Description de méta-modèle

Dans ce qui suit, nous allons décrire le *MMDA*. Présenter l'ensemble des blocs de ce méta-modèle reviendrait à la spécification *OMG*. Ce méta-modèle ne se veut pas exhaustif mais représente la majeure partie des concepts présents dans le *DA*, la figure VI.2 représente la partie supérieure de *MMDA*.

Une action peut avoir des flux d'activité entrants et sortants qui spécifient les flux de contrôle et les flux de données vers et depuis d'autres nœuds. La figure VI.3 présente la méta-classe (classe du niveau méta-modèle) "ActivityEdge" : une méta-classe possédant deux liens d'héritage avec les méta-classes "ObjectFlow" et "ControlFlow".

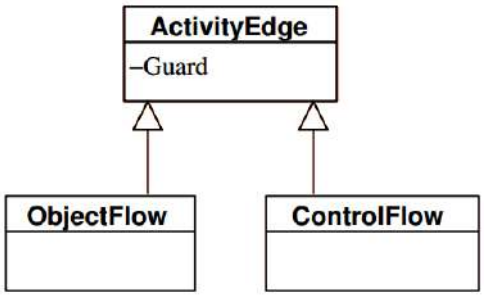


FIGURE VI.3 – Les flux d'activité(ActivityEdge)

Comme il est dit précédemment, les *DA*s sont composés de nœuds. Ces nœuds sont de contrôle, d'objet ou d'action. Cette description a été appliquée pour l'ensemble du méta-modèle présenté en figure VI.4

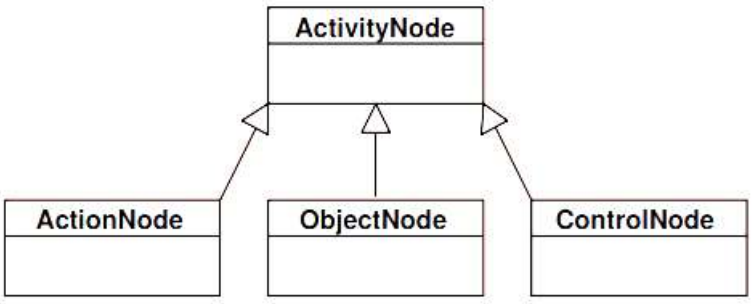


FIGURE VI.4 – Les nœuds d'activity(ActivityNode)

Il existe plusieurs types de nœuds de contrôle : nœud initial (initial node), nœud de fin d'activité (final node), nœud de fin de flot (flow final), nœud de décision (decision node), nœud de fusion (merge node), nœud de bifurcation (fork node), nœud d'union "synchronisation" (join node). La partie de méta-modèle qui décrit ces nœuds est représentés par le diagramme donné par la figure VI.5.

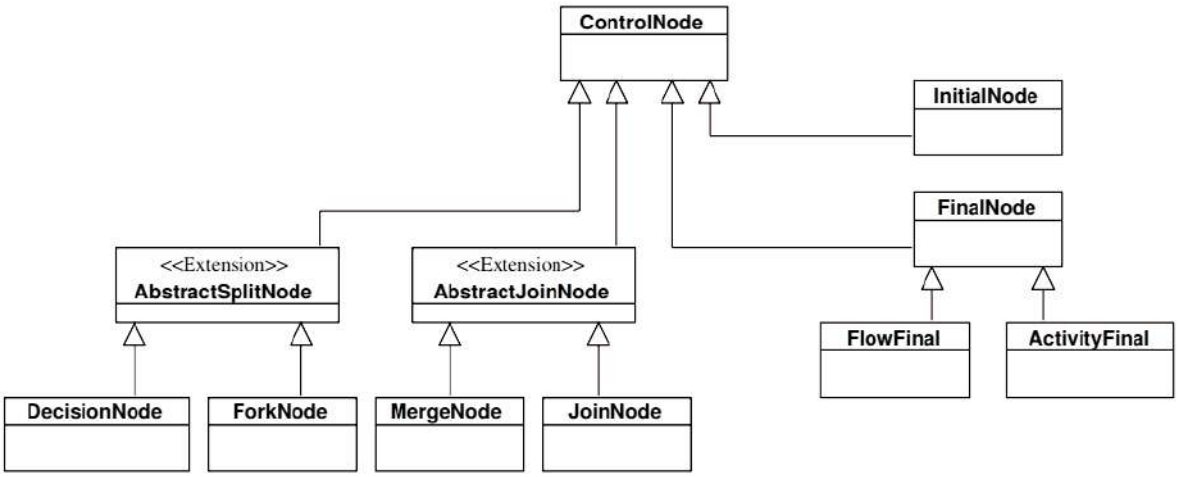


FIGURE VI.5 – Les nœuds de contrôle(ControlNode)

Un nœud d'objet est un nœud d'activité qui indique une instance d'un classificateur particulier, peut-être dans un état particulier ou peut être disponible à un point particulier de l'activité. Les nœuds d'objet peuvent être utilisés de différentes manières selon l'endroit où les objets sont écoulés.

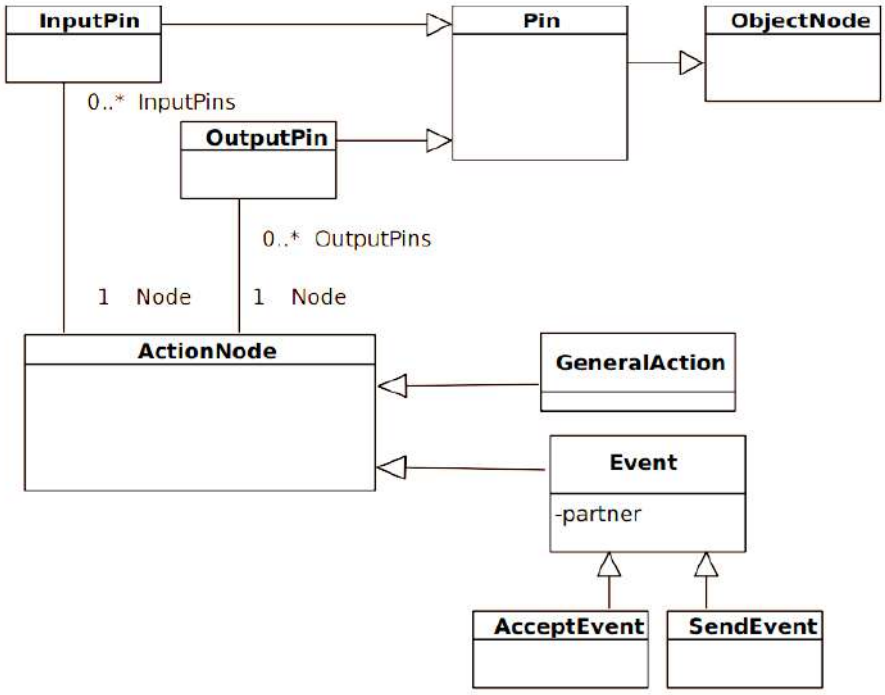


FIGURE VI.6 – Les nœuds d'action et d'objet (ActionNode,ObjectNode)

Une action représente une étape unique dans une activité, c'est celle qui n'est pas décomposée au sein de l'activité. Une activité représente un comportement qui est composé

d'éléments individuels qui sont des actions. Une action est donc simple du point de vue de l'activité contenante, en tant que pièce de structure à l'intérieur d'un modèle d'activité, elle est un élément discret unique. Comme une spécification du comportement à effectuer, elle peut invoquer le comportement référencés qui est qui peut être complexe. En conséquence, une activité définit un comportement qui peut être réutilisé dans de nombreux endroits, alors une instance d'une action est utilisé qu'une seule fois à un moment donné à une activité. Cette dernière partie correspond au méta-modèle est représenté par le diagramme dans la figure VI.6.

Parmi les intérêts du choix de méta-modèle intégré dans **EMF** est l'existence d'un editeur graphique de diagramme d'activité dans **POLARSYS** qui se base sur **PAPYRUS**[49], cet dernier permet de créer des modèles instanciées du *MMDA*.

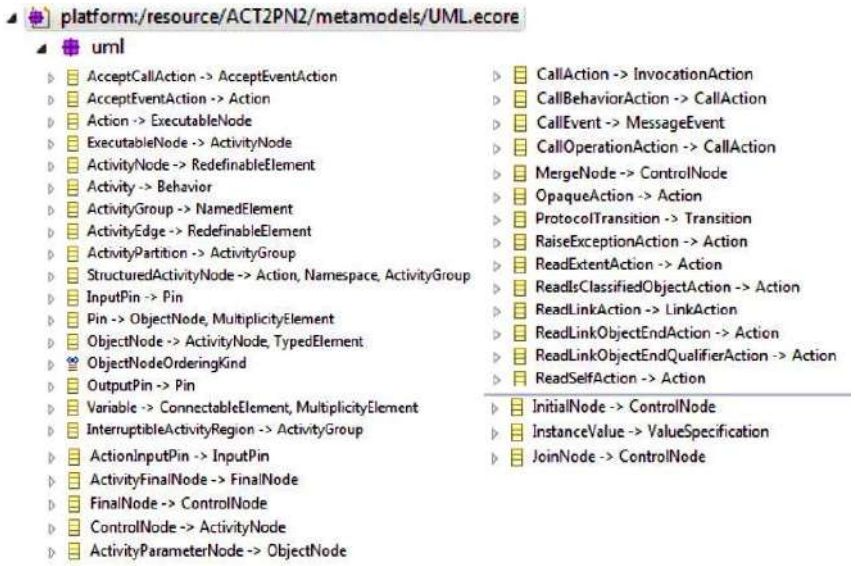


FIGURE VI.7 – MMDA sous extension (Ecore)

La figure VI.7 est une visualisation du *MMDA* à partir d'EMF sous l'environnement *Eclipse*.

VI.2.1.2 Métamodèle cible (réseaux de Petri)

Maintenant que le méta-modèle source a été défini il faut définir le méta-modèle cible. Il serait plus intéressant d'établir un méta-modèle qui engendre un modèle totalement indépendant des plateformes voire figure VI.1. Une transformation pourra par la suite effectuer pour adaptée le réseau de Petri résultant à une plateforme spécifique (dans notre cas l'outil *CPNTools*).

VI.2.1.2.1 PNML

La communauté des réseaux de Petri a cherché longtemps des moyens d'échanger des modèles des réseaux de Petri sans ambiguïté, basé sur un format d'échange convenue sur

un standard. Toutefois, ils ont rapidement réalisé que ce serait encore mieux si ce format est compatible avec une définition formelle convenue sur des réseaux de Petri. Clairement indiqué, le format doit être la syntaxe concrète de la syntaxe abstraite spécifiée de cette définition formelle.

PNML[1] [43] est une proposition d'un format d'échange basé sur XML pour les réseaux de Petri. À l'origine, *PNML* était destiné à servir comme un format de fichier pour la version Java de noyau de réseau de Petri. Mais, il est avéré que actuellement plusieurs autres groupes développent un format d'échange *XML* encore. Ainsi, *PNML* est seule contribution à la discussion en cours et les efforts de normalisation d'un format basé sur *XML*. Le langage *PNML* a été conçu selon les techniques de méta-modélisation de *DSL* couramment mises en œuvre dans *MDA* [50].

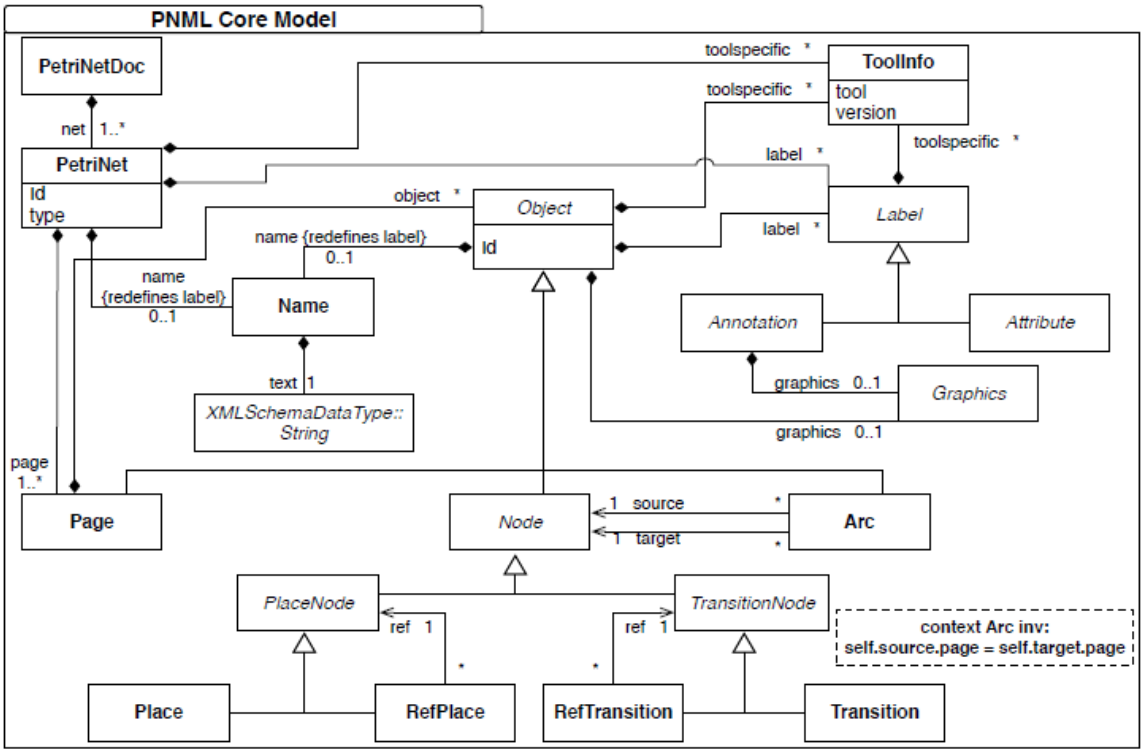


FIGURE VI.8 – Le *MMPNML* pnmlcoremodel extrait de *PNML framwork*

VI.2.1.2.2 Description de méta-modèle

Le méta-modèle PNML(MMPNML) représenté par la figure VI.8, résume les relation d'associations, de composition et d'héritage entre les méta-classes des différents éléments conforme au *PNML*. Ces Méta-classes sont décrites dans ce qui suit :

- **PetriNet** : est la Méta-classe racine qui représente un réseau de Petri. Elle a un attribut *name* de type *String* qu'il prend le nom du réseau de Petri. Elle peut contenir une ou plusieurs *Pages*.

1. Petri Net Markup Language

- **Page** : permet de contenir un ensemble des *Objets* (*Nœuds* et *Arcs*). Elle assure la structuration et le découpage par composition/décomposition hiérarchique du modèle.
- **Object** : représente les nœuds et les arcs de *RdP*, un *objet* possède un identificateur unique *ID* de type *String* pour référencer chaque objet.
- **Node** : est une classe qui hérite de celle de *Object*, elle représente un nœud d'un réseau de Petri. Ce nœud peut être une place (*Place*) ou une transition (*Transition*).
- **Place** : l'une des deux classes qui hérite de la classe *Node*. Chaque place peut avoir zéro ou plusieurs arc sortants (*OutArc*), et zéro ou plusieurs arc entrants (*InArc*).
- **Transition** : elle représente les transitions, elle est la deuxième classe qui hérite de la classe *Node*. Chaque transition peut avoir zéro ou plusieurs arc sortants (*OutArc*), et zéro ou plusieurs arc entrants (*InArc*).
- **Arc** : est l'objet qui relie les nœuds d'un *RdP*, parmi ses attributs :
  * **Source** : indique le nœud source soit une seul(1) place ou une seul(1) transition.
  * **Target** : indique un et un seul nœud cible.
- **RefPlace** : Permet de référencer une seul place, mais une Place peut référencer plusieurs **RefPlaces**, cette dernière a les mêmes attributs qu'une place en plus de l'attribut *ref* qui défini la relation de référencement.

Pour des raison d'hiérarchisation des *RdP*, il faut faire apparaitre la notion de **transition de substitution** qui fait référence à une **Page**, alors nous apportons des modifications sur ce méta-modèle au niveau de la méta-classe **Transition** en ajoutant l'attribut **type** qui permet de faire la distinction entre une transition et une "transition de substitution", l'attribut **type** prend la valeur *"SubsTransition"* si la transition et une transition de substitution.

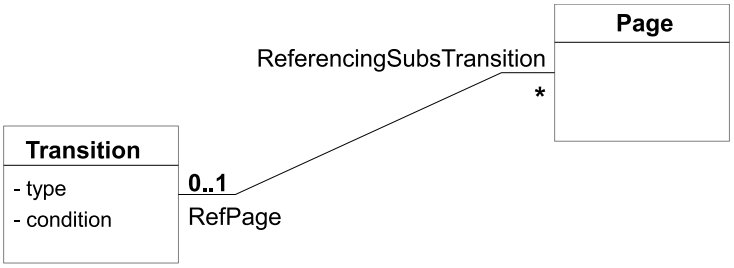


FIGURE VI.9 – Les modifications effectué sur MMPNML

En plus, nous ajoutons la relation **RefPage** qui permet à une transition de substitution de référencer vers une et une seul **Page**.

Autre ajout dans la méta-classe **Transition** concerne l'attribut *condition*, pour spécifier des conditions de franchissement. Les modifications effectué au niveau de *MMPNML* sont représenté dans la figure VI.9.

Dans notre travail, nous avons introduit des modifications sur le méta-modèle *pnmlcoremodele* intégré dans *PNMLFramework* [51] et nous avons utilisé le méta-modèle

résultant comme méta-modèle cible dans notre transformation. Le méta-modèle cible sous son extension *Ecore* est représenté dans la figure VI.10.

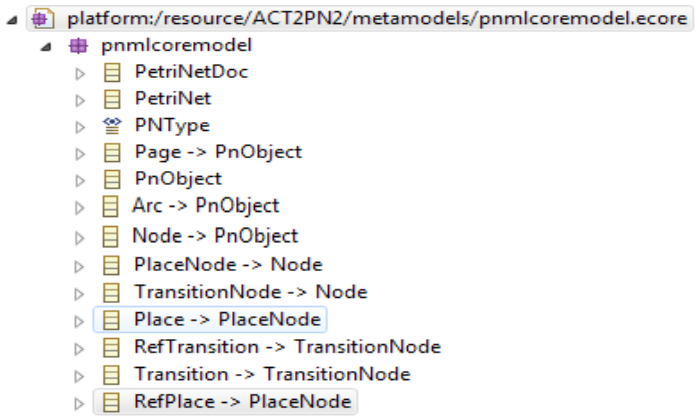


FIGURE VI.10 – *MMPNML* sous sa extesion ECORE

VI.2.2 Processus de transformation

Notre objectif est d'automatiser la transformation des *DA* vers les RdP. La transformation consiste à trouver les correspondances entre les concepts des deux méta-modèles source et cible. Les *DA*s sont supposés être syntaxiquement correct.

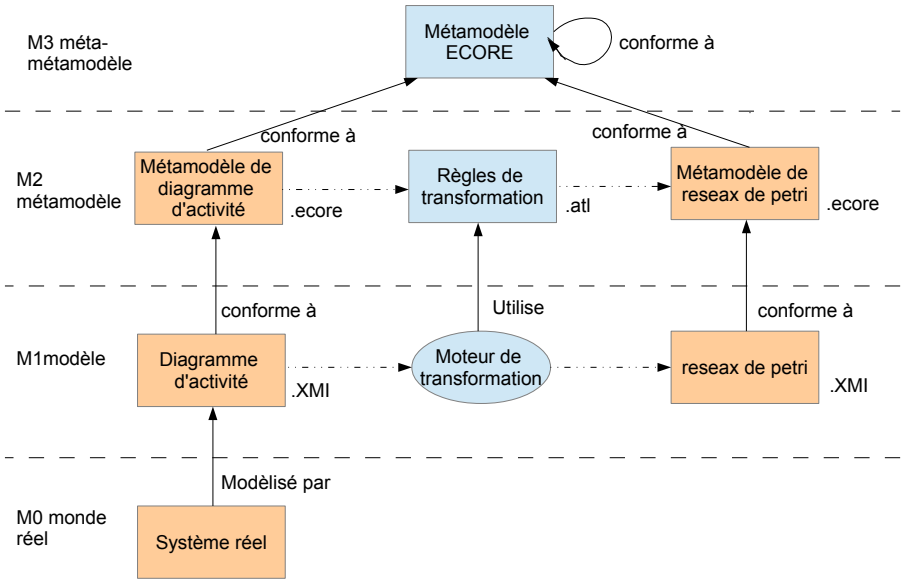


FIGURE VI.11 – Processus transformation ATL des DAs vers RDPs

Le processus de transformation est réalisée par l'application des règles. Une règle de transformation consiste à transformer un concept énoncé dans le méta-modèle source à un concept correspondant dans le méta-modèle cible. Le principe générale de transformation des *DA* vers *RdP* à travers l'outil *ATL* est représenté par la figure VI.11.

VI.2.2.1 Règles de transformation

Une transformation totale des DAs vers les *RdP*s demanderait des travaux plus conséq--uents sans toutefois poser des problèmes techniques majeurs, car le formalisme des *DA*s est riche. Pour cela, il existe un nombre d'études fournissant des informations détaillées sur les différentes propositions des règles de transformation DA vers RDP [52][53][54] [55]. Il est essentiel d'avoir une vision au niveau du modèle des résultats attendus. Ceci afin d'assurer une conservation des propriétés, et du comportement des *DA*s dans le *RdP* équivalent. Nous nous somme inspiré de ces travaux pour définir nos règles de transformation.

VI.2.2.1.1 Transformation des entités de diagramme d'activité

Cette partie va établir les concepts de la transformation entre quelques entités du *DA* (noeud/arc) et un ou plusieurs blocs équivalents *RdP* en donnant l'implémentation *ATL*. Ces blocs devront respecter le comportement (sémantique) du *DA* préconisé par l'OMG.

À première vue, un réseau de Petri peut sembler proche d'un *DA*. Leurs représentations graphiques et leurs comportements (déplacement de jetons dans le "réseau", composition des graphes : nœud, arc...) peut donner l'impression d'avoir deux techniques de modélisation très proches. Nous définissons ci-après les règles de transformation pour les éléments du *DA*.

- **Modèle et Activité :**
  Un *DA* est un modèle, ce modèle va être transformé en un *RdP* qui va avoir le même nom que le modèle. Le *DA* peut être composé de plusieurs activités, chaque activité sera transformé en une page du *RdP*. Cette transformation est représentée dans la figure VI.12. Comme résultat, nous obtenons un *RdP* hiérarchique dont chaque page représente une sous activité du *DA*.

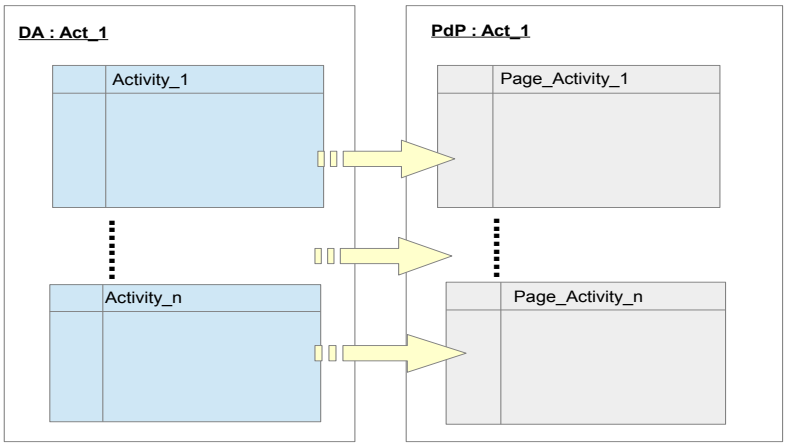


Figure VI.12 – Transformation de "Model"

Dans ce qui suit, le code *ATL* de la règle "*PetriNet*" est présenté par le listing VI.1, et celui de la règle "*Activity*" se situe dans la partie annexe VII.5 par le listing 3 :

PetriNet

```
rule PetriNet {
  from
    model: MMuml!Model
  to
    pn: MMpn!PetriNet (
      id <- model.name,
      pages <- model.packagedElement -> collect(e | thisModule.
    resolveTemp(e,
          'page'))
    )
}
```

Listing VI.1 – Le code *ATL* de la règle "*PetriNet*"

Le détail concernant la syntaxe pour l'implémentation des règles *ATL* est présenté dans le chapitre III, au niveau de la section (ref) qui concerne le langage *ATL*.

– **Action(opaque action) :**
Comme montre la figure(VI.13), une action sera translatée en deux transitions, deux arcs et une place. La première transition marque le début de l'exécution de l'action, la deuxième marque la fin de son exécution, et la place indique si l'action est en exécution.

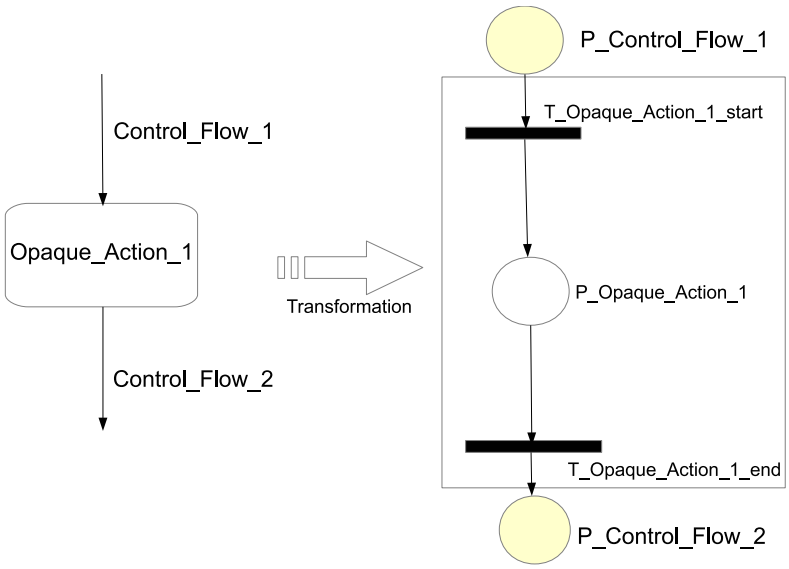


FIGURE VI.13 – Transformation de "opaque action"

Le listing 4 dans la partie annexe VII.5 présente l'implémentation par code *ATL* la règle "*OpaqueAction*" :

– **Action d'appel d'activité (Call behavior action)**
L'objet "Call behavior action" permet d'invoquer une autre activité depuis l'activité qui le contient. La transformation de cet objet donne une hiérarchie au sein du *RdP*. La figure VI.14 résume cette transformation. L'objet *Call behavior action* se transforme vers une transition de substitution(TS) contenu par la page de l'activité *appelante*, en plus la TS va référencer la page qui est translatée à partir de l'activité *appelée*. Cette transformation engendre aussi des places de référence "$Refplace$" de type "in"/"out" pour faire le lien entre les place de la page mère et la page fille.

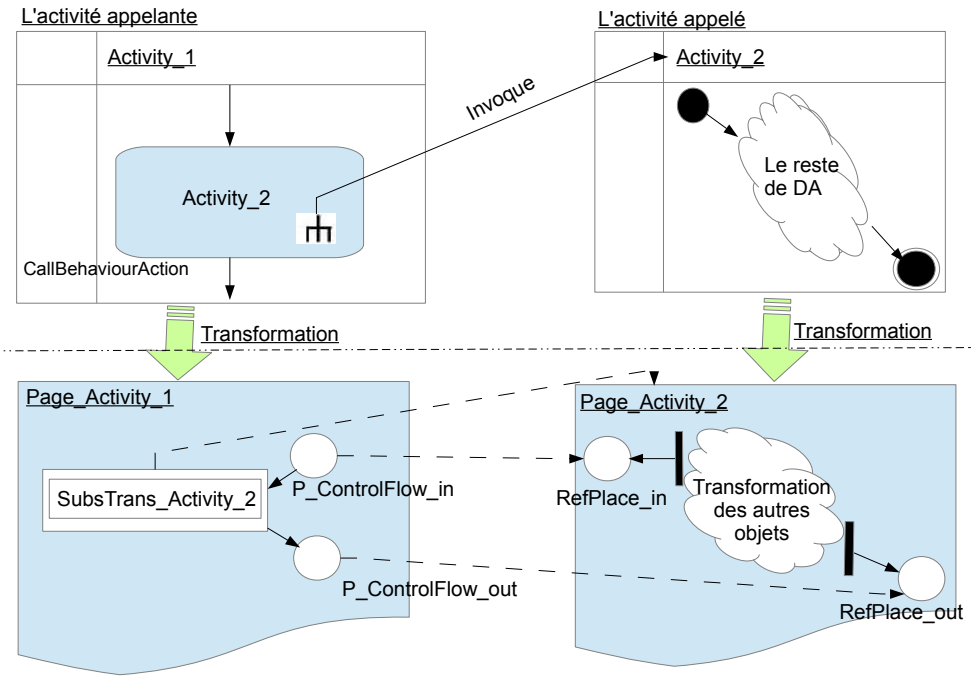


FIGURE VI.14 – Transformation de "Call behavior action"

Nous implémentons deux *helper*s pour l'aide au développement des règles ATL.
le listing 1 dans la partie annexe VII.5 présente le code ATL de *helper* qui retourne l'objet qui commence par l'activité appelé.
Le listing 2 présente le code ATL de l'autre *helper* qui retourne l'objet qui se termine par l'activité appelé.
Le listing 6 présente le code ATL qui correspond à la règle permettant de transformer des "Call behavior action"s. La règle ATL *Call behavior action* fait appel aux deux *helper*s, $getBeginActivity()$ sur la ligne 13 et $getEndActivity()$ sur la ligne 29 dans le listing 6.

– **Flux de control (Controlflow) :**
C'est l'objet qui relie les nœuds de contrôle ou/et les nœuds d'action, sa transformation en blocs *RdP* équivalent donne un arcs(in) vient d'une transition associée à un nœuds transformé, suivie par une place qui est la cible de l'arc(in), en suite un

arc(out) qui se dirige vers une transition associé à un objet déjà translaté. La figure VI.15 montre cette transformation.

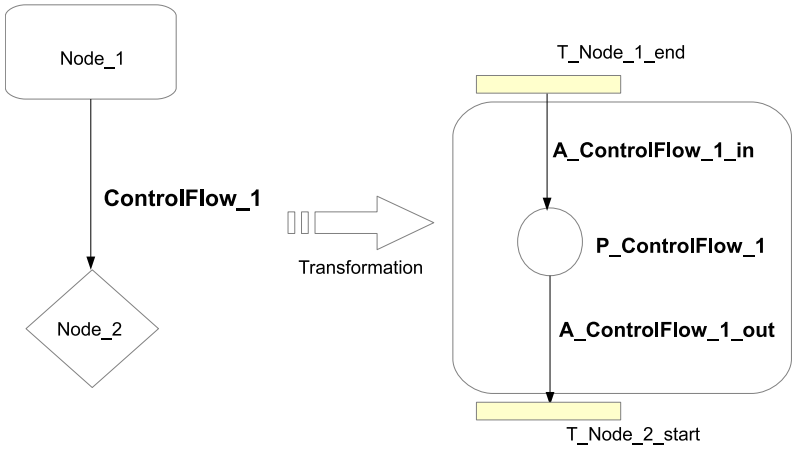
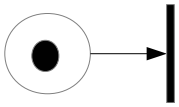
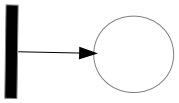
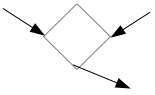
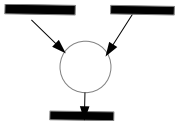


FIGURE VI.15 – Transformation de "Controlflow"

Le code *ATL* de la règle *Controlflow* est représenté par le listing 5 dans la partie annexe VII.5.

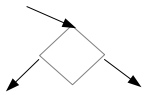
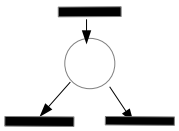
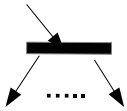
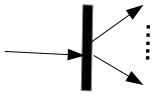
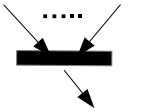
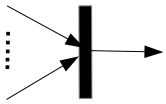
– **Quelques transformations des autres objets de DA :**
Le reste des règles de transformation des autres objets de base de DA sont décrit par la table VI.2.1 :

TABLE VI.2.1: Table contient le reste des règles de transformations

Objets SysML	Blocs *RdP*	Solution
Initiale Node 		Création d'une place marqué avec un marquage initial avec une transition permet de fournir a chaque arc sortant un jeton.
Finale Node 		Création d'une transition pour chaque arc entrant, afin d'activer la place finale, donc démarqué l'ensemble d'activités.
Merge Node 		Création d'une transition pour chaque entrée du Merge et l'ajout d'une place qui rdistribue directement chaque jeton en sortie.

à suivre dans la page suivante

TABLE VI.2.1 – *Suivant*

Objets SysML	Blocs *RdP*	Solution
Decision Node 		Création des transitions de sortie en fonction du nombre de sortie de décision.
Fork Node 		Création d'une transition qui permet de redistribuer autant de jetons que d'arcs sortant de façon synchrone.
Join Node 		Création d'une transition qui a les même propriétés de synchronisation dans les RDPs.

Le code *ATL* correspond au reste des règles de transformation sont décrites dans la partie annexe VII.5.

VI.2.2.2 Adaptation au schéma *CPNTools* Pour la vérification

L'objectif de transformation des *DA*s vers *RdP*s est la vérification du comportements des systèmes modélisés, en se basant sur des méthodes de vérification formelles applicables sue les *RdP*s. Cependant, la vérification peut se faire en utilisant diverses techniques et outils. Pour cela, nous avons choisi un outil puissant dans ce domaine, *CPNTools*, ce dernier comporte un *model-checker* qui aide à la vérification. Mais Le format *PNML* du *RdP* résultant depuis la transformation *ATL* que nous avons proposé n'est pas compatible avec *CPNTools*, pour cela et afin que notre démarche soit réalisable dans la pratique il faut adapter le format *PNML* avec le schéma spécifique compréhensible par l'outil *CPNTools*.

VI.2.2.2.1 Schéma CPN

Le format XML pour les fichiers d'extension *.cpn* est décrit en utilisant un document DTD. Pour un fichier *.cpn* soit bien validé par *CPNTools*, les éléments indispensables qu'il doivent présenté dans le document descriptif sont : des pages, des places, des transitions et des arcs, plus la cohérence de *RdP*. Il n'est pas nécessaire de s'inquiéter sur les options graphiques corresponds à chaque nœud(élément) de *RdP*, il peut également être possible d'ignorer les paramètres liés aux configurations d'un fichier(*.cpn*). Les éléments dans ce schéma sont représentés d'une façon structurée.
La structure d'un fichier *.cpn* est conforme au schéma *CPNTools* de base présenté par le listing VI.2 suivant :

```
1<file>
2<workspaceElements>
```

```
3   <generator .../>
4   <cpnet>
5     <globbox>
6       ...
7     </globbox>
8     <page id="...">
9       <pageattr name="...">
10      <place id="...">
11        <text>...</text> <!-- nom -->
12        <marking .../>
13        <type> <!-- Domaines de couleurs -->
14          <text>...</text>
15        </type>
16        <initmark> <!--  markage initial -->
17          <!--  type -->
18        </initmark>
19      </place>
20      <trans id="..." explicit="false">
21        <posattr .../> <!-- position -->
22        <text>...</text> <!-- nom -->
23        <subst subpage="..." portsock="..."> <!-- optionel : transition de
      substitution; -->
24          <subpageinfo id="..." name="..."> <!-- la page a reference-->
25          </subpageinfo>
26        </subst>
27  ...
28        <cond> <!-- guard(condition de franchissement) -->
29        </cond>
30  ....
31      </trans>
32      <arc id="..." orientation="TtoP|PtoT|BOTHDIR" order="...">
33        <transend idref="..."/> <!-- id de transition de l'arc -->
34        <placeend idref="..."/> <!-- id de place de cette arc -->
35         ...
36        </code>
37      </arc>
38    </page>
39    <page id="...">
40  ...<!-- Le RdP peut contenir plusieurs pages -->
41    </page>
42   </cpnet>
43 </workspaceElements>
44 </file>
```

Listing VI.2 – schéma *CPNTools* de base

VI.2.2.2.2 Correspondance *PNML/CPNTools*

La grande différence qui apparu entre le schéma *CPN* et celui de *PNML* et la syn-

taxe, mais globalement il y a des petites trucs sur le plan structurelle de chacun des deux schéma. Par la suite, nous définisserons la correspondance entre les éléments de ces deux schémas.
D'abord, le fichier sous extension *.cpn* se commence par une entête générale qui comporte des informations sur l'encodage des caractères, l'espace des noms XML, le fichier DTD qui valide le fichier CPN, la version de l'outil *CPNTools* utilisé et des autres informations générales. Et se termine par un pied qui est plus ou moins dynamique, ce dernier comporte des informations sur le positionnement des fenêtres graphiques et des palettes de manipulation des différentes fonctionnalité offert par cet outil.
Ensuite, sur le corps du fichier sera construit par projection des éléments du fichier *PNML* vers des éléments CPN.

Alors, quelques types d'éléments *PNML* soit précédés par l'expression de cette espace de nom.

Projection PNML/CPN

Voici les éléments de base de *PNML* et leurs équivalents de *CPNTools* après transformation. Chaque élément *PNML* est bien expliqué dans la section VI.2.1.2.2 :

- **PetriNet** :
  La figure VI.16 représente la transformation de l'élément *PetriNet* en code CPN, L'élément sous *PNML* est la racine de corps du *RdP* et donne l'ID du *RdP*.

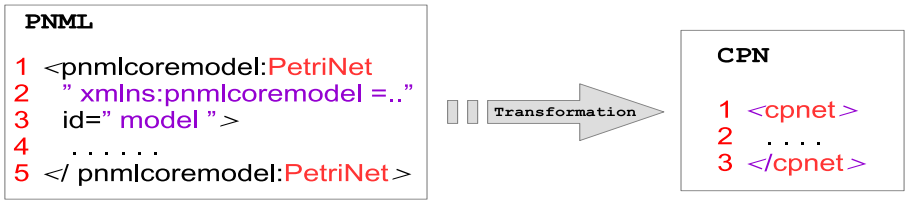


FIGURE VI.16 – Transformation de "PetriNet"

- **Pages**

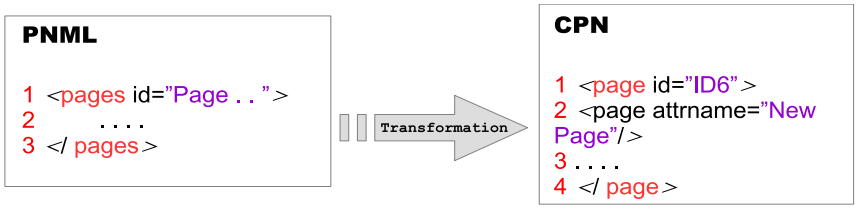


FIGURE VI.17 – Transformation de "Page"

La deuxième ligne dans le listing CPN dans la figure VI.17 représente l'élément qui donne le nom de la page.

– **Arc**

La figure VI.18 représente la transformation de l'élément *Arc* en code *CPN* correspondant. L'attribut "*orientation*" sur la première ligne de listing *CPN* varie entre "*PtoT*" qui signifie que le sens de l'arc est d'une place vers une transition, et "*TtoP*" qui signifie que la direction de l'arc est d'une transition vers une place. La ligne numéro 2 et 3 représentent respectivement les deux éléments extrémités de l'arc.

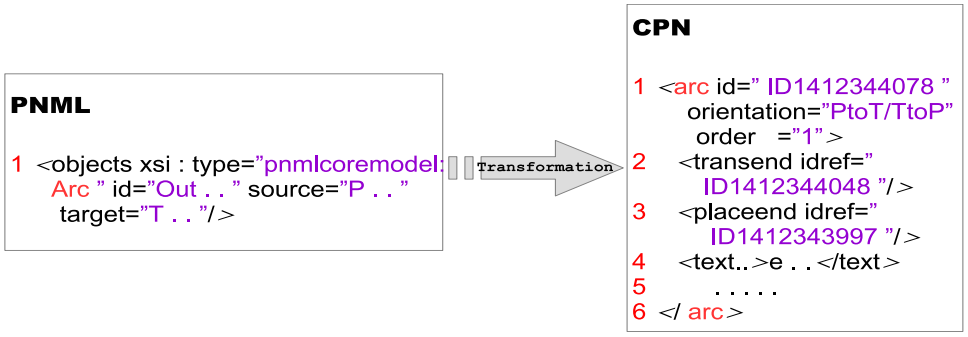


FIGURE VI.18 – Transformation de "Arc"

– **Place**

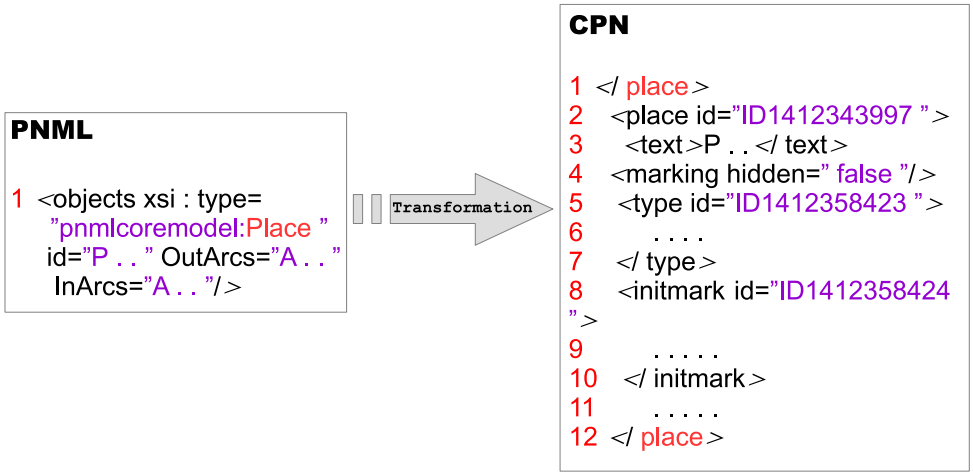


FIGURE VI.19 – Transformation de "Place"

Les éléments qui se trouvent depuis la quatrième ligne jusqu'au dixième ligne dans le listing *CPN* présenté sur la figure VI.19, donnent des informations sur le type et l'état de marquage initiale des places.

– **Transition**

Dans le listing *CPN* qui est présenté dans la figure VI.20, les éléments qui se situe depuis la troisième jusqu'au septième ligne sont optionnelle, sauf s'il y à une hiérarchie dans le *RdP*, c-à-d dans la page du *RdP*, une transition de substitution

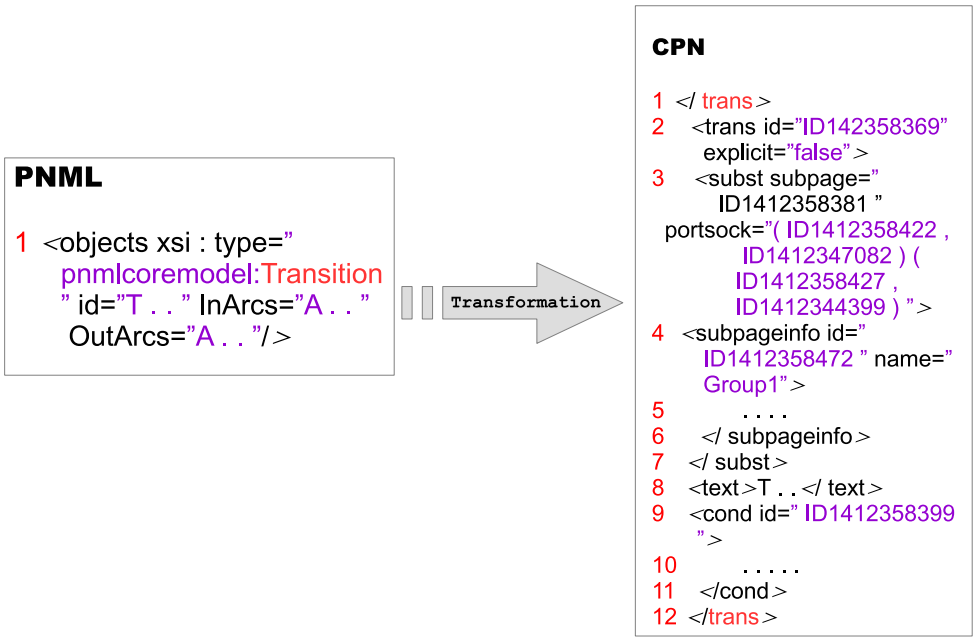


FIGURE VI.20 – Transformation de "Transition"

apparue. Ces éléments devient indispensables pour faire le lien entre la page fille et la transition de substitution. L'attribut "*portsock*" permet de faire le référencement entre les places et les places de référence. L'élément $< cond >$ sur la neuvième ligne est aussi optionnelle sauf s'il y à une condition sur le franchissement de transition.

– **RefPlace**

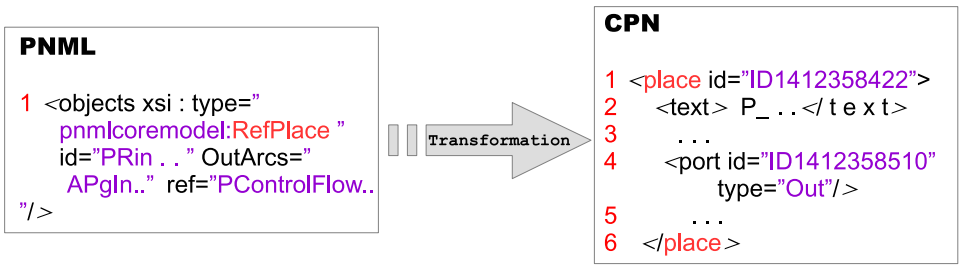


FIGURE VI.21 – Transformation de "RefPlace"

La structure de cette élément est la même que celui de l'élément *place*, sauf s'il y à une spécificité sur cette élément, comme présente la figure VI.21 et plus précisément sur la quatrième ligne dans le listing CPN, l'élément $< port >$ qui fait la différence entre une place de référence est une place, cette élément est caractérisé par l'attribut *type* qui peut être "*in*" qui signifie que la place de référence est d'entrée, ou "*out*" qui signifie que la place de référence est de sortie".

En générale l'élément $< text >$ ... $< /text >$ qui se trouve dans la plupart des éléments de *RdP* indique le *nom* de l'élément. L'ID de chaque élément de *RdP* sous schéma *CPNTools* doit respecter la forme "ID"+chiffre"(par exemple $id =$"$ID173$") pour

que le modèle soit validé par l'outil.

VI.3 Translation des expressions ActRL vers la logique ASK-CTL

Jusqu'à présent, nous avons définie comment translater les *DA* en *RdPCH* conforme au format *CPNTools*. Afin de pouvoir exploiter Ces *RdPCH* compatibles avec l'outil *CPNTools* pour vérifier les exigences *SysML*, il est nécessaire de translater ces derniers en formules de logique temporel. Dans cette section nous allons décrire comment translater les expressions *ActRL* (voire chapitre V) en formules de logique *ASK-CTL*, une extension de *CTL* implémentée dans *CPNTools*. Le passage du *ActRL* vers *ASK-CTL* es réalise en deux étapes, en passant du *ActRL* à *CTL* puis de *CTL* à *ASK-CTL*. La figure VI.3.2 montre les étapes de ce passage.



FIGURE VI.22 – Transforamations des expression ActRL vers ASK-CTL

VI.3.1 De ActRL vers CTL

Comme *ASK-CTL* est une extension de *CTL*, nous proposons de passer tout d'abord de *ActRL* à *CTL*. Pour ce passage, nous exploitons les translation définies dans [1] et nous définissions la table VI.3.2 pour passer de *ActRL* à *CTL*. Par la suite, nous passons du *CTL* à *ASK-CTL*.

La plupart des translations utilisent l'opérateur (W : *weak until*), cet opérateur est lié à l'opérateur (U :*until*) par les équivalences suivantes :

$$\text{A[x } W \text{ y]} = \text{!E[!y } U \text{ (!x and !y)]}$$
$$\text{E[x } U \text{ y]} = \text{!A[!y } W \text{ (!x and !y)]}$$

Cet opérateur rend souvent les correspondances plus faciles à comprendre.

TABLE VI.3.2: Translation des expressions *ActRL* en CTL extrait de [1]

Patterns	Scopes	Propriétés CTL
absence	Globally	AG(!P)
	(*) Before R	A[(!P | AG(!R)) W R]
	(*) After Q	AG(Q -> AG(!P))

à suivre dans la page suivante

TABLE VI.3.2 – *Suivant*

Patterns	Scopes	Propriétés CTL
	(*) Between Q and R	AG(Q *and* !R -> A[(!P | AG(!R)) W R])
	(*) After Q until R	AG(Q *and* !R -> A[!P W R])
existance	Globally	AF(P)
	(*) Before R	A[!R W (P and !R)]
	(*) After Q	A[!Q W (Q and AF(P))]
	(*) Between Q and R	AG(Q and !R -> A[!R W (P and !R)])
	(*) After Q until R	AG(Q and !R -> A[!R U (P and !R)])
universallity	Globally	AG(P)
	(*) Before R	A[(P | AG(!R)) W R]
	(*) After Q	AG(Q -> AG(P))
	(*) Between Q and R	AG(Q and !R ->A[(P | AG(!R)) W R])
	(*) After Q until R	AG(Q and !R -> A[P W R])
Precedence	Globally	A[!P W S]
	(*) Before R	A[(!P | AG(!R)) W (S | R)]
	(*) After Q	A[!Q W (Q and A[!P W S])]
	(*) Between Q and R	AG(Q and !R -> A[(!P | AG(!R)) W (S | R)])
	(*) After Q until R	AG(Q and !R -> A[!P W (S | R)])
Response	Globally	AG(P -> AF(S))
	(*) Before R	A[((P -> A[!R U (S and !R)]) | AG(!R)) W R]
	(*) After Q	A[!Q W (Q and AG(P -> AF(S))]
	(*) Between Q and R	AG(Q and !R -> A[((P -> A[!R U (S and !R)])
	(*) After Q until R	AG(Q and !R -> A[(P -> A[!R U (S and !R)]) W R])

La table VI.3.2 présente les vingt cinq cas possibles étudies dans ce contexte extrait de [1] et décrit les correspondances pour les propriétés des patterns dans la logique *CTL*.

VI.3.2 De CTL vers ASKCTL

Nous proposons la translation des cinq possibilités des expressions *ActRL* parmi la totalité des cas possibles en formules *ASK-CTL*. La table VI.3.3 présente la translations *ActRL* vers *CTL* vers *ASK-CTL* des cinq cas étudiés.

TABLE VI.3.3: De ActRL vers CTL vers ASC-CTL

ActRL	CTL	ASC-CTL
always p globally	AG(P)	INV(p)
never p globally	AG(!P)	INV(NOT(p))
eventually p globally	AF(P)	EV(p)
always p after q	AG(Q -> AG(P))	INV(OR(NOT(q),INV(p)))
never p after q	AG(Q -> AG(!P))	INV(OR(NOT(q),INV(NOT(p))))

La translation des expression *ActRL* consiste aussi à translater les expressions de prédicat(voire chapitre V). D'abord, il faut implémenter des fonctions qui permet de requêter du *RdP* en terme de leur propriétés pour connaitre l'état du *RdP* à un moment donné depuis l'espace d'état généré.

La communauté *CPNTools* fournie un manuel spécifique pour l'utilisation de l'outil *State Space* (*SS*) fourni avec *CPNTools*, il aide aussi à la maîtrise pour l'analyse de l'espace d'état [56]. En conséquence, nous pouvons investir à travers cette avantage pour formuler nos propre requêtes. L'implémentation des requêtes nécessitent d'écrire du code *ML*. Le listing VI.3 représente l'implémentation en code *ML* de la fonction *isActive*() que nous proposons, cette fonction est utilisé dans la plupart des expressions des prédicat dans les exigences *ActRL*. Elle prend en paramètre la place qu'on va testé.

isActive()

```
1 fun testPlace n =(Mark.Page'Act_1 1 n=[marquage] );
2 val P = NF("is active", testPlace );
```

Listing VI.3 – Le code ML de la fonction *isActive*()

Le listing VI.4 représente le code *ML* d'une formule *ASK-CTL* qui est le résultat de translation d'une expression *ActRL* de forme (*always Act_1.isActive*() *globally*). Cette formule $ASK - CTL$ utilise le résultat de translation de l'expression de prédicat *Act_1.isActive*() présentée dans le listing VI.3.

ASK-CTL

```
1 val myaskctl =INV(P);
2 eval_node myaskctl 1;
```

Listing VI.4 – Une formule ASK-CTL ecrite en code ML

À fin de rendre la translation des expressions *ActRL* en *ASK-CTL* automatique. *Xtend*[57] fournie avec *Xtext* est un dialecte souple et expressive de Java. Nous avons

translaté les expressions *ActRL* en formules *ASC-CTL* qui sont écrites en code *ML*, à l'aide de *Xtend*. Á travers *Xtend* on peut manipuler d'une manière souple le contenue des exigences *ActRL* exprimés à l'aide de l'éditeur que nous avons implémenté. Si la translation *ActRL* en *ASK-CTL* est bien terminé, on génère un fichier *.sml* contenant de(s) fonction(s) en code *ML*, ce fichier sera importé et utilisé dans le fichier contenant le RDP *CPNTools*.

VI.4 Vérification

Une fois les transformations cités précédemment donne les résultat attendus (un *RdP* sous format *CPNTools*, et des formules *ASK-CTL* en code *ML*), nous pouvons entamer la phase de vérification, cette dernière se faite par le biais de l'outil *CPNTools*. La figure VI.23 présente l'opération de vérification des propriétés sur le modèle avec le modèle-checker(*CPNTools*). Dans ce qui suit, nous illustrons les étapes impliquées pour la vérification.

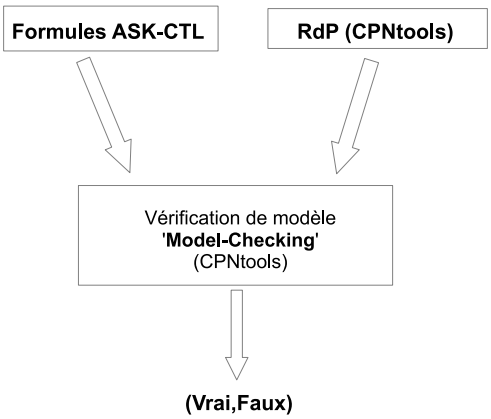


FIGURE VI.23 – Vérification de modèle

1. **Calcul de l'espace d'état :**
   Avant que nous pouvons calculer un espace d'état de *RdP*, nous devons invoquer l'outil de l'espace d'état, puis, nous générons l'espace d'état (*SS*) et le graphe des composantes fortement connexes (*SCC*).
2. **Invocation de la bibliothèque *ASK-CTL* :**
   Pour invoquer la bibliothèque *ASK-CTL* on utilise l'instruction suivante :

```
1  use ( ogpath ^ "ASKCTL/ASKCTLloader.sml")
```

3. **Sélection d'un(e) fonction/fichier** *ML*
   On sélectionne une fonction *ML* comporte une formule *ASK-CTL* que nous voulons la vérifier parmi l'ensemble des formules *ASK-CTL*. Ou, on importe directement un fichier qui contient toutes fonction *ML*.

4. **Évaluation de la formule :**
   Elle renvoie une valeur vrai ($true$) ou faux ($False$). Si le vérificateur renvoie «vrai», implique que la propriété est confirmée pour le modèle ou d'autre terme le modèle répond au exigence.

VI.5 Conclusion

Dans ce chapitre, nous avons définie une translation automatique des spécifications *SysML* composé d'un *DA* et d'un *DE* en spécifications formelles constitue d'un *RdP* et des formules de logique temporelle. L'outil *CPNTools* est utilisé pour achever la vérification des exigences *SysML* sur les *DA*. Les exigences *SysML* ont été translatées en logique *ASK-CTL*, une extension de *CTL* implémenté dans *CPNTools*. Nous avons vue que ses transformations permet l'exploitation de l'outil *CPNTools*, son module de génération d'espace d'état et le modèle-checker *ASK-CTL* pour réaliser la vérification.

Chapitre VII

Étude de cas

VII.1 Introduction

Pour illustrer notre méthodologie, nous présentons une étude de cas concernant un système de protection contre les incendies dans une salle informatique. Ce système de protection contient un système automatique de détection d'incendie conçu pour détecter le feu dans ses premiers stades. Ainsi que des alarmes internes et externes et un système d'extinction d'incendie. Quand une incendie est confirmée dans une salle informatique, le centre de détection active le dispositif d'actionnement automatique pour l'extinction du feu automatiquement.

VII.2 Les spécifications SysML

Selon la méthodologie proposée dans le chapitre IV, le but de cette étape est de produire un diagramme d'activité et un diagramme d'exigence *SysML* pour le système de protection contre les incendies.

VII.2.1 Diagramme d'activité

Nous spécifions le comportement du système de protection contre les incendie par le diagramme d'activité présenté dans la figure VII.1. Ce diagramme a été édité en utilisant l'editeur des diagrammes *PAPYRUS* intégré dans *POLARSYS*.

Lorsqu'une incendie se produit, elle sera détectée par le système de détection d'incendie. Le diagramme contient le call behaviour action $detecter_le_feu$ qui fait appel à une autre activité représenté par un sous diagramme d'activité nommé $detecter_le_feu$, il est donnée par la figure VII.2. L'activité $detecter_le_feu$ permet de détecter la chaleur par les détecteurs d'incendie, ensuite, elle détermine si une incendie est survenue.

Après la fin de l'activité appelés par le call behaviour action $detecter_le_feu$, dans l'activité principale $proteger_du_feu$, deux cas sont modélisés, le feu détectée et confirmée ou bien le feu n'est pas détecté. Dans le cas de détection et de confirmation du feu, le système déclenche les alarmes internes et externes, en même temps l'action *extinction* est activé. Et Pour le second cas, aucune action ne va être exécute.

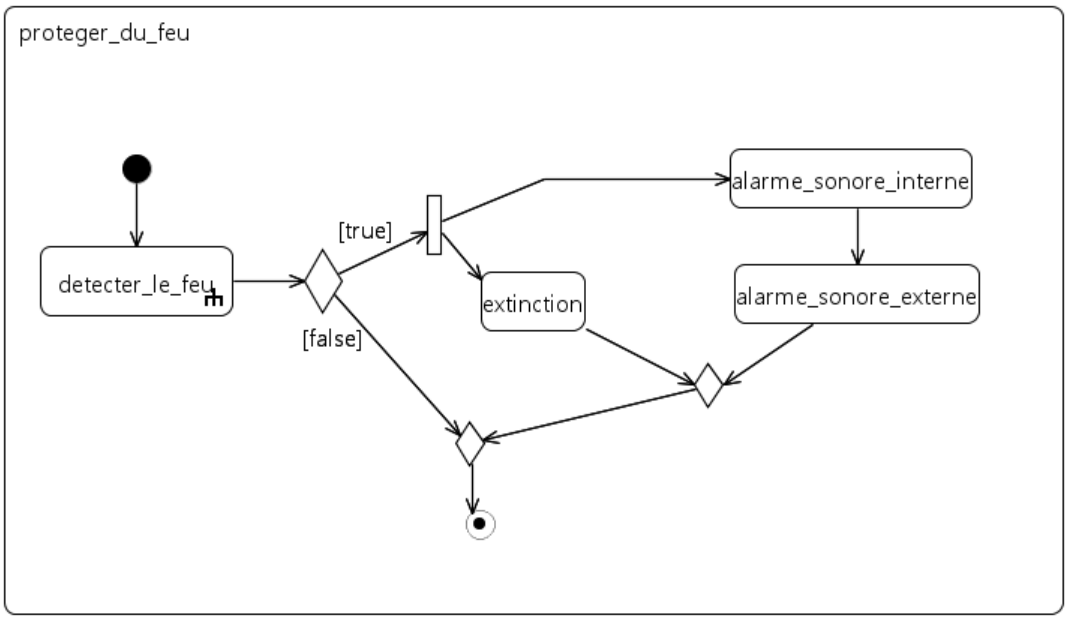


FIGURE VII.1 – L'activité principale *proteger_du_feu*

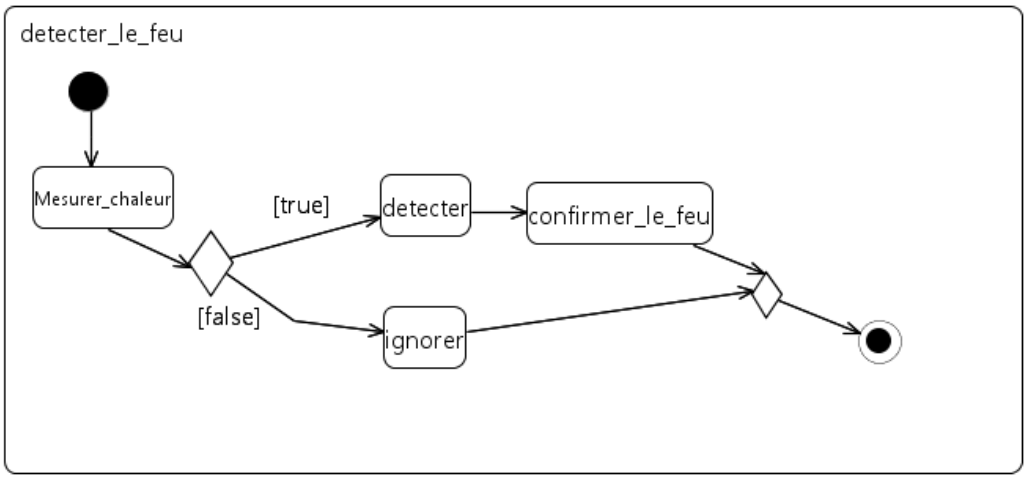


FIGURE VII.2 – L'activité appelé *detecter_le_feu*

VII.2.2 Diagramme d'exigence

Les exigences du système de protection contre les incendies sont modélisé dans un diagramme d'exigence $SysML$. Dans ce contexte, nous considérons seulement un ensemble d'exigences. Comme montre la figure VII.3.

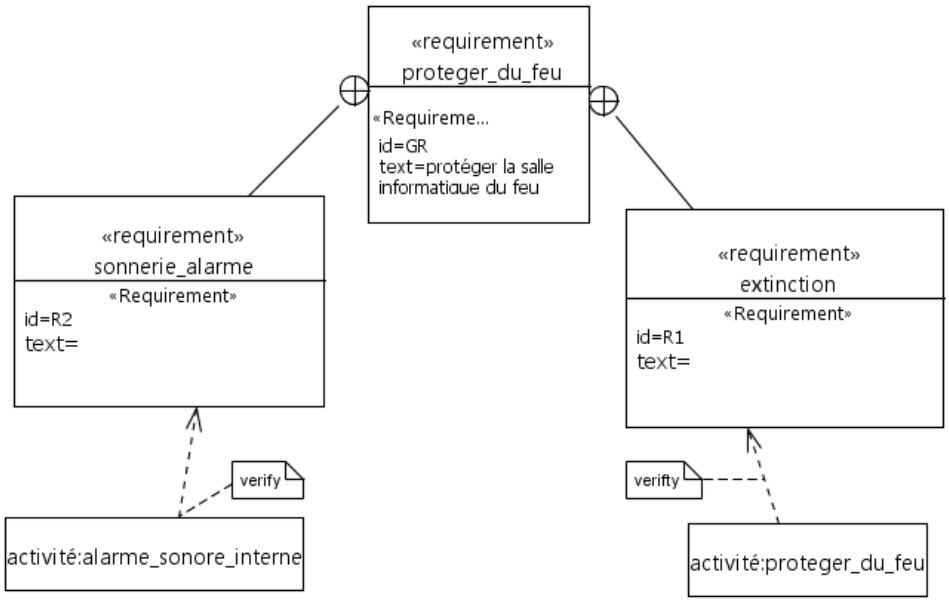


FIGURE VII.3 – Le diagramme d'exigence pour le système de protection contre les incendies

L'exigence principale du système est de protéger une salle d'ordinateurs contre les incendies. Cette exigence est composée de deux exigences ($R1$, $R2$). L'exigence $R1$ concerne la suppression d'incendie, elle vérifie que l'activité de suppression d'incendie ne jamais produit après que la détection d'incendie n'est pas confirmée. Cette exigence est vérifiés par l'activité $proteger_du_feu$ (voir Figure VII.3). L'exigence $R2$ est d'assurer le déclenchement des alarmes lorsque la détection d'une incendie est confirmée, elle décrit que éventuellement, l'alarme interne sera déclenché globalement. L'exigence $R2$ est vérifiée par l'action $alarme_sonore_interne$.

Les exigences qui ont la relation $<< verify >>$ sont édité avec l'éditeur des exigences $ActRL$ que nous avons proposé dans le chapitre V. Ceci est illustré par la figure VII.4.

Lorsque les exigences sont validé par notre éditeur, ces expressions sont intégrées dans le DE, à l'aide de l'application fournie avec une interface qui facilite l'opération, (étape 9 dans la figure VII.6), pour obtenir le DE final Comme représenté dans la figure VII.5.

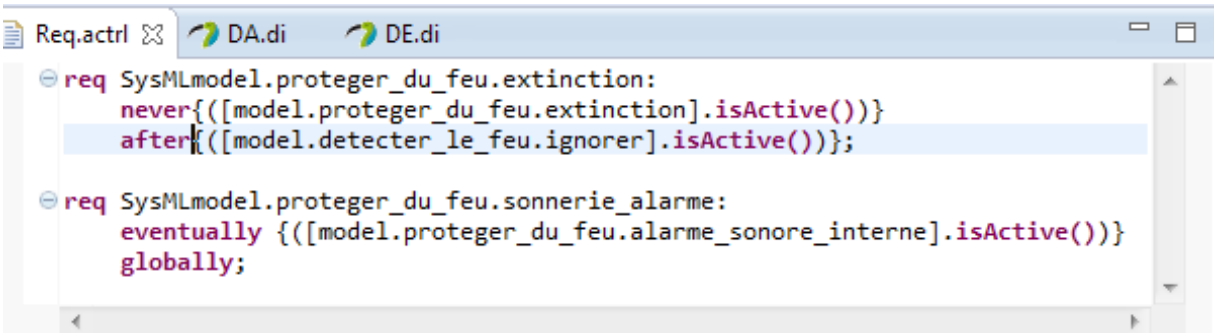


FIGURE VII.4 – L'expression des exigences avec l'editeur ActRL

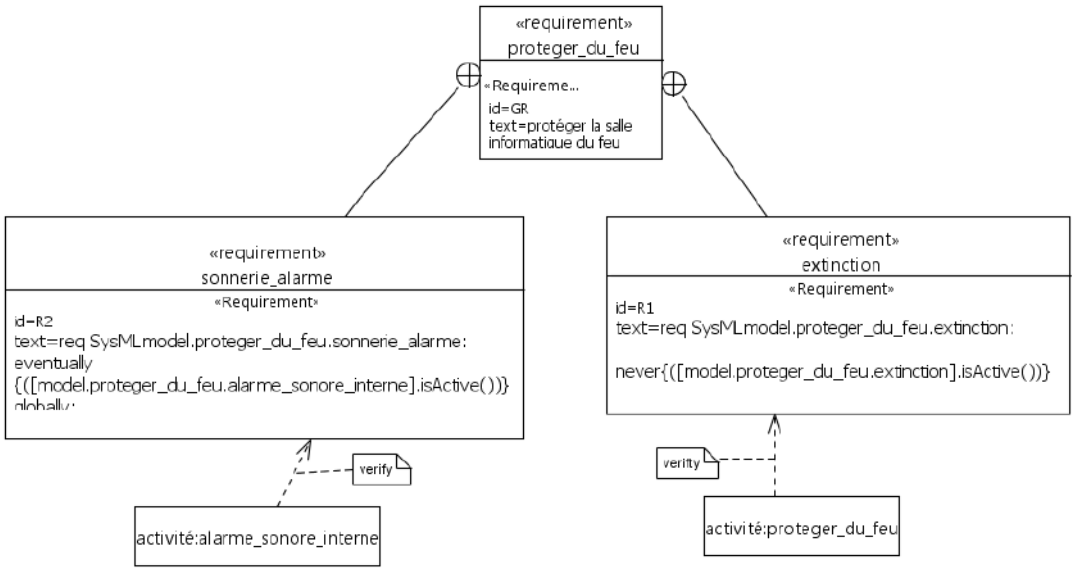


FIGURE VII.5 – Le DE après l'intégration des expressions extrait de l'editeur *ActRL*

VII.3 Transformation des spécifications SysML en langages formelles

La deuxième étape de notre méthodologie consiste à transformé les modèles précédents en des langages formelles. Premièrement, on va translater le diagramme d'activité en réseau de Petri. Ensuite, on fait la translation des exigences $ActRL$ en $ASK - CTL$.

VII.3.1 Transformation de DA en RdP

Dans cette section on va présenté le résultat de transformation du diagramme d'activité (voir la figure VII.1) en réseau de Petri. Pour cela, nous utilisons l'interface graphique que nous avons criée pour faciliter les procédures de transformation. l'interface est présentée dans la figure VII.6. Premièrement, on choisi le fichier du diagramme d'activité, son chemin apparue dans le champ numéroté par 1, ensuite on donne le chemin de fichier RdP sous format $PNML$ numéroté par 2, et on lance la transformation par le bouton numéro 3, puis une boite de message indique que la transformation est bien terminé.

À partir du fichier $PNML$ résultant depuis la première transformation, on commence une transformation de ce fichier en format CPN, on choisi l'emplacement du fichier résultant de l'extension *cpn*, et la valeur du marquage initial dans le champ 4 et 5 respectivement, et on lance la transformation par le bouton 6. Le résultat est présenté par la figure VII.7, il sagit d'un *RdPCH* en format *CPNTools* composé de deux pages.

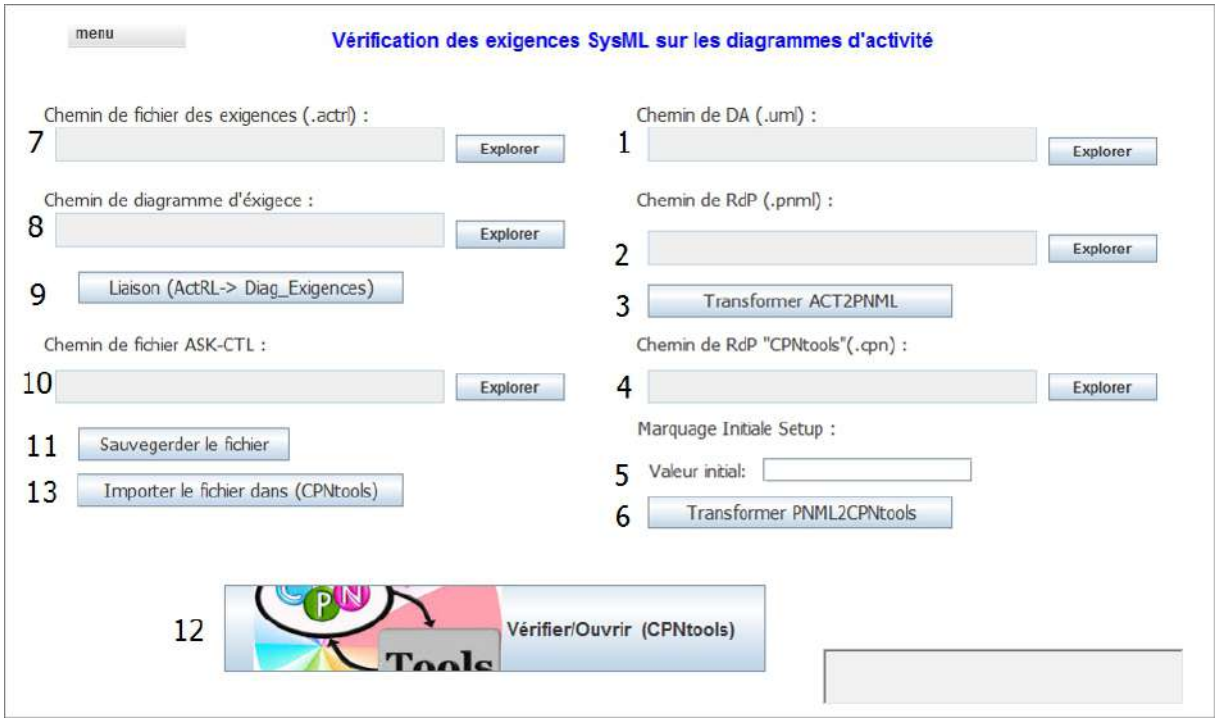


FIGURE VII.6 – Interface graphique de l'application de la vérification

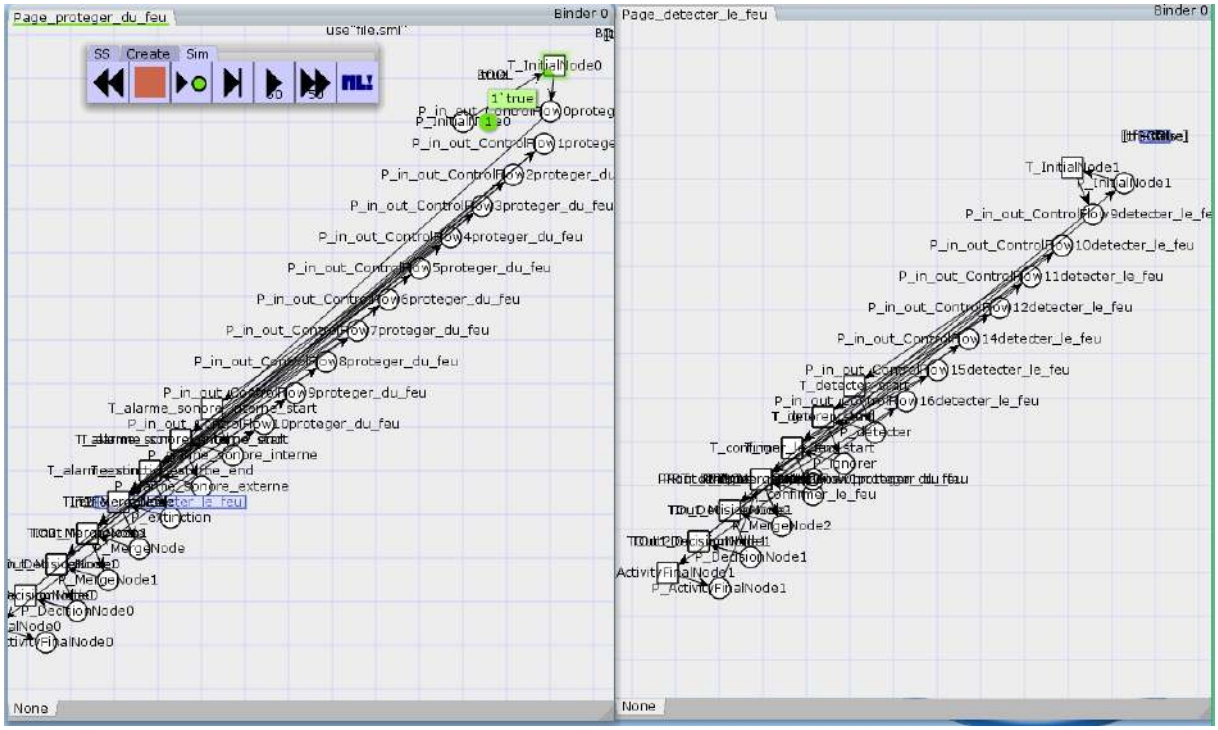


FIGURE VII.7 – Le réseau de Petri dérivée après la translation

VII.3.2 Translation des exigences ActRL en ASK-CTL

La translation des expressions *ActRL* génère des formules *ASK-CTL* sous formes des fonctions en code *ML* intégrés dans une fichier *SML*. Nous avons pris la translation des cas des expressions *ActRL* similaire aux cas mentionnés dans le tableau VI.3.3.

Pour sauvegarder ce résultat, on spécifié le chemin du fichier dans le champ numéro 10. Autrement, les propriétés *ASK-CTL* obtenu après translation sont exprimé en le langage *ML* va être intégrer dans le fichier (*.cpn*) qui contient le RdP adapté à l'outil *CPNTools* . La figure VII.8 présente une formule *ASK-CTL* en langage *ML* après translation.

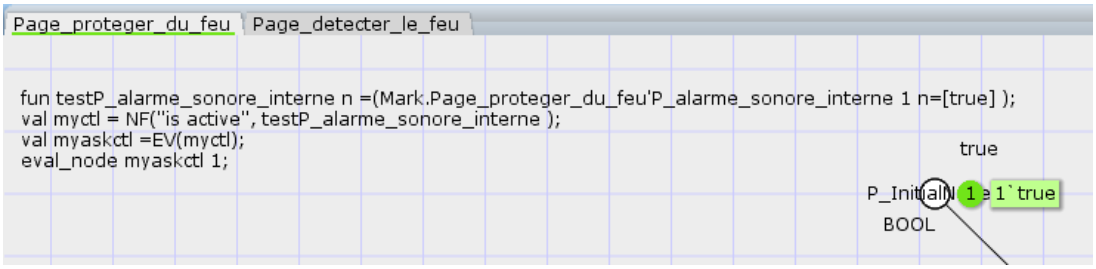


FIGURE VII.8 – Exemple d'intégration du fonction ML dans l'outil *CPNTools*

VII.4 Verification

La vérification se faite en utilisant l'outil *CPNTools* comme il a été dit précédemment. Dans cette partie, on va présenté les étapes de la vérification.

Premièrement, on lance la vérification utilisant l'interface présenté dans la figure VII.6 par bouton numéro 12, ensuite l'outil *CPNTools* va être lancer pour commencer la vérification, on doit générer l'espace d'état et invoquer la bibliothèque **ogpath** puis, importer le ficher *sml*. Pour plus de détail sur la procédure de vérification par *CPNTools* voir [22]. En utilisant l'option de l'évaluation offert avec la palette de simulation, on obtient le résultat présenté par la figure VII.9. On observe dans cette figure, que le résultat de l'évaluation de la formule *ASK-CTL* est vrai (*true*), donc le modèle répond au exigence équivalent par rapport au DA.

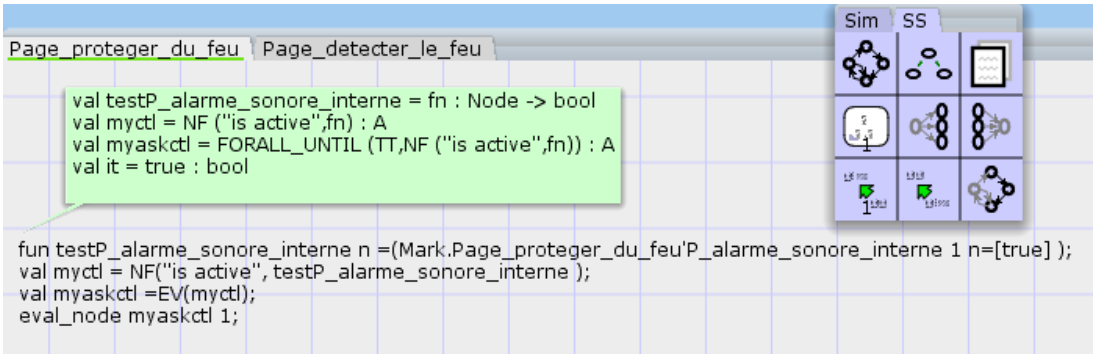


FIGURE VII.9 – Un résultat d'évaluation par *CPNTools*

VII.5 Conclusion

Ce chapitre, a été consacré pour l'application de notre méthodologie par un étude de cas. Pour cela, nous avons proposé comme des spécifications *SysML*, un diagramme d'activité et un diagramme d'exigence en exprimant les exigences en langage *ActRL* avec l'outil proposé. Ensuite, nous avons translaté le diagramme d'activité en réseau de Petri compatible avec *CPNTools*, puis, on a translaté les exigences exprimé avec *ActRL* en *ASK-CTL*. Enfin, nous avons arriver a vérifier les exigences *SysML* sur le diagramme d'activité. Pour faciliter ces procédures, on a utilisé l'application fournie avec une interface graphique qu'on a proposé.

Conclusion générale

a vérification des systèmes critiques et complexes est nécessaire, surtout des systèmes dont le mauvais fonctionnement aurait des conséquences dramatiques. À fin de valider les modèles de ces systèmes au moment de leurs conception, des travaux plus conséquents sur les méthodes et les techniques de vérification et de validation ont été consacrés. De même, les méthodes et les approches formelles offrent un large éventail des techniques d'analyse et de vérification.

Dans ce travail, nous avons proposé une démarche qui permette la vérification automatique des exigences *SysML* sur les diagrammes d'activité. Notre démarche a été basé sur la transformation des spécifications *SysML* en spécifications formelles. Comme Première étape, nous avons transformé des *DA* en *RdP*, dans un cadres *IDM* en respectant les spécifications définies par l'OMG. Le langage utilisé pour implémenter les règles de transformation est le langage de transformation atlas *ATL*. Comme nous avons ciblé un outils de simulation et de vérification des RdP, l'outil *CPNTools*, nous avons adapté les RdP à cet outil. La deuxième étape sera la translation des exigences $SysML$ en formules de logique temporelle.

Comme une étape préliminaire à notre approche, nous avons proposé de formaliser les exigences $SysML$ utilisant le langage $ActRL$ et nous avons implémenté pour cet objectif un outil pour permettre l'édition des expressions $ActRL$. Cette formalisation permette d'éviter toute ambiguïté due à l'expression en langage naturel des exigences $SysML$ et par conséquence pouvoir translater d'une façon automatique ces exigences en formules logique *CTL*. Pour pouvoir réaliser cette deuxième étape de translation des exigences $ActRL$ en logique *ASK-CTL* qui est une variante de la logique *CTL* implémenté dans l'outil *CPNTools*, nous avons implémenté la translation de quelques expressions $ActRL$ en langage extensible de java, *XTend*. À fin de vérifier les formules de logique *ASK-CTL* sur les RdP, nous exploitons l'outil *CPNTools* qui comporte un *model-checker* consiste à la vérification des RdP.

Nous avons conçu et réalisé un prototype d'un outil qui implémente les translations citées. Cependant, ce prototype est en phase de développement et en test.

Dans des futurs travaux, nous envisageons d'aller plus loin dans les étapes de transformation, Premièrement, de DA en RdP en donnant une large possibilités d'utilisation par l'implémentation de maximum des règles de transformation des éléments de *DA* (*Swimlanes*, *Pins*, *ObjectNodes*, *ActivityParameterNode*, *signales*). Deuxièmement, la généralisation de translation des expressions $ActRL$ en *ASK-CTL*. Nous avons comme

perspective aussi, l'intégration de l'outil de l'édition des exigences *ActRL* avec l'éditeur des diagrammes d'exigence *PAPYRUS* à l'environnement *Eclipse*, nous envisageons aussi de valider notre approche par l'application des études de cas plus complexes.

Bibliographie

[1] George Avrunin Spec Patterns, Hamid Alavi. http ://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml.

[2] Philippe Meyne. Introduction à l'ingénierie système. 2012.

[3] Lionel GENDRE et Jean-Marie. Virely ens cachan sysml-gendre-virely. *Computational Materials Science*, (1), 2013.

[4] Guillaume FINANCE. Modélisation sysml. *UMLchannel*, 2010.

[5] Object Management Group. Systems modeling language (sysml) specification. *SysML Partners (www.sysml.org)*, 2012.

[6] Pascal Roques. Modélisation de systèmes complexe savec sysml. *Groupe Eyrolles*, ISBN : 978-2-212-13641-8, 2013.

[7] Jeannette M. Wing. "a specifier's introduction to formal methods". *Computer, vol.23(9) :8-23*, 1990.

[8] 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.

[9] T. MURATA. Petri nets : properties, analysis and applications. *Proceedings of IEEE*, Vol. 77(N4) :pp. 571–580, april 1989.

[10] M.P.J. STEVENS L.P.M. BENDERS. Petri net modelling in embedded system design. *CompEuro 92, Computer systems and software engineering*, (1) :612–17., 1992.

[11] G. W. BRAMS. Réseaux de petri : théorie et pratique. *EATCS Monographs on Theoretical Computer Science, W. BRAUER, G. ROZENBERG, A.SALOMAA, Springer-Verlag*, Tl et T2 Masson, 1983.

[12] H. ALLA R. DAVID. Lecture notes in computer science, g. goos, 1.hartmanis, springer-verlag,. *Proceedings of an advanced course Bad Honnef*, part. l :254, 1986.

[13] A. N. MICHEL K. PASSINO. Stability and boundedness analysis of des. *ACC-92IFM7 Chicago*, 1992.

[14] CARDOSO. Sur les réseaux de petri avec marquages flous. *Thèse de Doctorat de l'université Paul SABATIER*, 23 oct. 1990.

[15] K. Jensen. "coloured petri nets : Basic concepts, analysis methods and practice use". *Monographs in Theoretical Computer Science, Springer-Verlag, Berlin*, Volumes 1-3, 1997.

[16] Hartmann J. Genrich and Kurt Lautenbach. "system modelling with high-level petri nets". *In Proceedings of the Theoretical Computer Science, vol. 13*, 1981.

[17] Jorge C. A. de Figueired Yi Deng, S. K. Chang 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.

[18] K. Jensen. Coloured petri nets and the invariant-method.

[19] K. Jensen. "high-level petri nets. *Third European workshop on applications and theory of Petri Nets, Varenna, Italy*, pages pp. 261–276, 1982.

[20] H. ALLA R. DAVID. Du grafcet aux réseaux de petri. *HERMES*, 1989.

[21] K. Jensen et R.M. Shapiro P. Hubert. Hierarchies in coloured petri nets. *Advances in Petri Nets, Lecture Notes in Computer Science. Springer Verlag, Berlin Heidelberg, New York*, vol 483, 1990.

[22] Aarhus University. http ://cpntools.org/.

[23] Jeffery-D Ullman. Elements of ml programming. *Pearson Education, 2nd edition*, ISBN : 9780137903870, 1998.

[24] Henry Michael Lassen Mads Laursen Jacob Frank Qvortrup Martin Stig Stissing Michael Westergaard Søren Christensen Anne Vinter Ratzer, Lisa Wells and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets.

[25] Aarhus University Kurt Jensen. http ://cpn-tools.org/documentation/examples/diningphilosopherssp.

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

[27] Henri Habrias and Marc Frappier. Software specification methods. *ISTE Ltd*, 2006.

[28] J. M. Wing et al E.M Clarke. Formal methods : State of the art and futur directions, in strategic directions in computing research workshop. *Cambridge, MA , ETATS-UNIS*, part. II :vol. 28, no 4 (290 p.) (4 p.3/4), pp. 626–643, (06/1996) 1996.

[29] O. Grumberg E. Clarke and D. Peled. Model checking. *MIT Press*, 1999.

[30] M. Huth and M.D. Ryan. Logic in computer science - modelling and reasoning about systems. *Cambridge University Press.*

[31] G. BERTHELOT. « temporal and modal logic ». *VAN LEEUWEN J., Ed., Handbook of Theoretical Computer Science*, vol. B Formal Models and Semantics, chapitre 16, p. 994–1072, Elsevier Science B.V., 1990.

[32] RYAN M. D HUTH M. R. A. Logic in computer science : Modelling and reasoning about systems. *University Press, Cambridge, England*, 2000.

[33] ENJALBERT P AUDUREAU E. Farias del cerro l, logique temporelle – sémantique et validation de programmes parallèles. *E.R.I, MASSON*, 1990.

[34] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen. Scc model checking. *University of Aarhus*, August 20, 1996.

[35] X a v i e r B l a n c. *MDA en action Ingénierie logicielle guidée par les modèles.* Groupe Eyrolles, 2005.

[36] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. rsti-l'objet. 2004.

[37] Marvin Minsky. Matter mind and models. *Semantic Information Processing*, 1968.

[38] Object Management Group. Model driven architecture (mda). *site Internet http ://www.omg.org/mda*, 2004.

[39] Jean Bézivin and Xavier Blanc. Promesses et interrogations de l'approche mda. *Journal Développeur Référence*, Septembre 2002.

[40] http ://www.obeo.fr/. obeo. *model driven company*, 2014.

[41] ATLAS group LINA and INRIA Nantes. Atlas transformation language user manual. *working draft*, January 2005.

[42] Messaoud Rahim Malika Boukala and Ahmed Hammad. Petri nets based approach for modular verification of sysml requirements on activity diagrams. *Proceedings of the International Workshop on Petri Nets and Software Engineering*, 2014.

[43] Hosted by LIP6 (CNRS UMR 7606 and UPMC). © pnml.org. 2009-2015.

[44] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen. Model checking coloured petri nets exploiting strongly connected components. *University of Aarhus, Computer Science Department Ny Munk egade, DK8000 Aarhus C, Denmark fac heng,schristensen,khmg,.*

[45] Matthew B. Dwyer George S. Avrunin and James C. Corbett. Patterns in property specifications for finite-state verification. *In Proceedings of the 21st Interna-tional Conference on Software Programming*, 1999.

[46] Gilles Muller. Framework for scheduler development. *LIP6 INRIA*, 2009.

[47] Standard document URL : http ://www.omg.org/spec/SysML/1.0/PDF. Omg systems modeling language (omg sysml^{TM})v1.0. *OMG Document Number : formal/2007-09-01*, 2007.

[48] Version 1.5. Object management group : Omg unified modeling language specification. *OMG Document Number : formal/2007-09-01*, mars 2003.

[49] http ://www.eclipse.org/papyrus/.

[50] Lom M. Hillah1 et Laure Petrucci2. Standardisation des réseaux de petri : état de l'art et enjeux futurs.

[51] lip6. (pnml framework)http ://pnml.lip6.fr.

[52] Jean-Claude Pascal Damien Foures, Vincent Albert. Activitydiagram2petrinet : Transformation-based model in accordance with the omg sysml specifications. *CNRS ;*

LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France, University of Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS ; F-31077 Toulouse Cedex 4, France.

[53] Ivan Trikovski. formalizing activity diagram of uml by petri nets.

[54] Antony Spietr Stainesi. Intuitive transformation of uml2 activities into fundamental modeling concept petri nets and colored petri nets. *Univesity of Malta.*

[55] Bhawana Agarwal. Transformation of uml activity diagrams into petri nets for verification purposes. *Advances in Petri Nets, Lecture Notes in Computer Science. Springer Verlag, Berlin Heidelberg, New York*, Volume 2 Issue 3 March 2013, International Journal Of Engineering And Computer Science ISSN :2319-7242.

[56] Søren Christensen Kurt Jensen and Lars M.Kristensen. Cpn tools state space manual. *University of Aarhus Department of Computer Science*, Last updated : January 2006.

[57] xtend. http ://www.eclipse.org/xtend/.

Annexe

Le code source ATL des règles de transformation DA en RdP proposé

getBeginActivity()

```
1 helper context MMuml!CallBehaviorAction def: getBeginActivity(): MMuml!
    ActivityNode =
2   if not self.behavior.ownedElement -> select(e | e.oclIsTypeOf(MMuml!
    InitialNode)).
3       isEmpty() then
4     self.behavior.ownedElement -> select(e | e.oclIsTypeOf(MMuml!
    InitialNode))
5   else
6     self.getBeginWithAction()
7   endif;
```

Listing 1 – Le code *ATL* de *helper getBeginActivity*()

getEndActivity()

```
1 helper context MMuml!CallBehaviorAction def: getEndActivity(): MMuml!
    ActivityNode =
2   if not self.behavior.ownedElement -> select(e | e.
3       oclIsTypeOf(MMuml!ActivityFinalNode)).isEmpty() then
4     self.behavior.ownedElement -> select(e | e.oclIsTypeOf(MMuml!
    ActivityFinalNode))
5   else
6     self.getEndWithAction()
7   endif;
```

Listing 2 – Le code *ATL* de *helper getEndActivity*()

Activity

```
1 rule Activity {
2   from
3     activity: MMuml!Activity
4   to
5     page: MMpn!Page (
6       id <- 'Page_' + activity.name
7     )
8 }
```

Listing 3 – Le code *ATL* de la règle "*PetriNet*"

Opaque Action

```
1       id <- 'A_' + actn.name + '_arc_start',
2       containerPage <- thisModule.resolveTemp(actn.activity, 'page'),
3       source <- ActStart,
4       target <- plc_on
5     ),
6     plc_on: MMpn!Place (
```

```
7       id <- 'P_' + actn.name,
8       containerPage <- thisModule.resolveTemp(actn.activity, 'page')
9     ),
10    arc_end: MMpn!Arc (
11      id <- 'A_' + actn.name + '_arc_end',
12      containerPage <- thisModule.resolveTemp(actn.activity, 'page'),
13      source <- plc_on,
14      target <- EndTrst
15    ),
16    EndTrst: MMpn!Transition (
17      id <- 'T_' + actn.name + '_end',
18      containerPage <- thisModule.resolveTemp(actn.activity, 'page'),
19      --InArcs <-  thisModule.resolveTemp(actvt.incoming, 'arc'),
20      OutArcs <- thisModule.resolveTemp(actn.outgoing, 'inarc')
21    )
22 }
```

Listing 4 – La règle "*OpaqueAction*"

Controlflow

```
1 rule ctlflw {
2   from
3     ctl: MMuml!ControlFlow
4   to
5     inarc: MMpn!Arc (
6       id <- 'A_' + ctl.name+ctl.activity.name + '_in',
7       containerPage <- thisModule.resolveTemp(ctl.activity, 'page'),
8       source <- Sequence{thisModule.resolveTemp(ctl.source, 'trst'),
9                  thisModule.resolveTemp(ctl.source, 'trst1'),
10                 thisModule.resolveTemp(ctl.source, 'EndTrst'),
11                   thisModule.resolveTemp(ctl.source, 'SubsTrns')},
12      target <- ArcPlc
13    ),
14    ArcPlc: MMpn!Place (
15      id <- 'P_in_out_' + ctl.name+ctl.activity.name,
16      containerPage <- thisModule.resolveTemp(ctl.activity, 'page'),
17      referencingPlaces <- Sequence{thisModule.resolveTemp(ctl.target,
18          'refPlacein'),
19            thisModule.resolveTemp(ctl.source, 'refPlaceout')}
20    ),
21    outarc: MMpn!Arc (
22      id <- 'A_' + ctl.name+ctl.activity.name + '_out',
23      containerPage <- thisModule.resolveTemp(ctl.activity, 'page'),
24      source <- ArcPlc,
25      target <- Sequence{thisModule.resolveTemp(ctl.target, 'ActStart'),
26            thisModule.resolveTemp(ctl.target, 'start'),
27            thisModule.resolveTemp(ctl.target, 'SubsTrns')}
28    )
```

```
}
```

Listing 5 – La règle "*Controlflow*"

Call behavior action

```
rule CallBehaviorAction {
  from
    CllBhvrAct: MMuml!CallBehaviorAction
  to
    refPlacein: MMpn!RefPlace (
      id <- 'PRin_' + refPlacein.ref.id,
      containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page'),
      OutArcs <- PInArc --thisModule.resolveTemp(CllBhvrAct.behavior, '
    PInArc')

    ),
    PInArc: MMpn!Arc (
      id <- 'APgIn_' + CllBhvrAct.name,
      --source <- thisModule.resolveTemp(OclUndefined,'refPlacein' ),
      target <- PInTrans,
      containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
    ),
    PInTrans: MMpn!Transition (
      id <- 'TPgIn_' + CllBhvrAct.name,
      OutArcs <- PInArc2,
      --thisModule.resolveTemp(CllBhvrAct.behavior,'PInArc2'),
      containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
    ),
    PInArc2: MMpn!Arc (
      id <- 'APgIn2_' + CllBhvrAct.name,
      target <- CllBhvrAct.getBeginActivity(),
      containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
    ),

    SubsTrns: MMpn!Transition (
      id <- 'T_' + CllBhvrAct.name,
      Type <- 'SubsTransition',
      RefPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page'),
      containerPage <- thisModule.resolveTemp(CllBhvrAct.activity, 'page'),
      InArcs <- thisModule.resolveTemp(CllBhvrAct.incoming, 'outarc'),
      OutArcs <- thisModule.resolveTemp(CllBhvrAct.outgoing, 'inarc')
    ),

    POutArc1: MMpn!Arc (
      id <- 'APgOut1_' + CllBhvrAct.name,
      source <- CllBhvrAct.getEndActivity(),
      target <- POutTrans,
      containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
    ),
```

```
44      POutTrans: MMpn!Transition (
45        id <- 'TPgOut_' + CllBhvrAct.name,
46        --InArcs <- thisModule.resolveTemp(CllBhvrAct.behavior,'POutArc1'),
47        OutArcs <- POutArc,
48        containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
49      ),
50      POutArc: MMpn!Arc (
51        id <- 'APgOut_' + CllBhvrAct.name,
52        --target <- thisModule.resolveTemp(OclUndefined,'refPlaceout'),
53        containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
54      ),
55      refPlaceout: MMpn!RefPlace (
56        id <- 'PRout_' + refPlaceout.ref.id,
57        containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page'),
58        InArcs <- POutArc --thisModule.resolveTemp(CllBhvrAct.behavior, '
      POutArc')
59
60      )
61
62  }
```

Listing 6 – La règle ”Call behavior action”

```
1         id <- 'TPgOut_' + CllBhvrAct.name,
2         --InArcs <- thisModule.resolveTemp(CllBhvrAct.behavior,'POutArc1'),
3         OutArcs <- POutArc,
4         containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
5       ),
6       POutArc: MMpn!Arc (
7         id <- 'APgOut_' + CllBhvrAct.name,
8         --target <- thisModule.resolveTemp(OclUndefined,'refPlaceout'),
9         containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page')
10      ),
11      refPlaceout: MMpn!RefPlace (
12        id <- 'PRout_' + refPlaceout.ref.id,
13        containerPage <- thisModule.resolveTemp(CllBhvrAct.behavior, 'page'),
14        InArcs <- POutArc --thisModule.resolveTemp(CllBhvrAct.behavior, '
      POutArc')
15
16
17  }
18
19  --Activity
20
21  rule Activity {
22    from
23      activity: MMuml!Activity
24    to
25      page: MMpn!Page (
```

```
26        id <- 'Page_' + activity.name
27      )
28 }
29
30 --ActivityParametre_Node
31
32 lazy rule ActivityParametre_Node {
33   from
34     ActPrm: MMuml!ActivityParameterNode
35   to
36     ActRefPlc: MMpn!RefPlace (
37       id <- 'C_' + ActPrm.name,
38       referencingPlaces <- thisModule.resolveTemp(ActPrm.redefinedNode, '
     fin_plc'),
39       containerPage <- thisModule.resolveTemp(ActPrm.activity, 'ForkNode')
40     )
41 }
42
43 --In_putPin
44
45 lazy rule InputPin {
46   from
47     InPin: MMuml!InputPin
48   to
49     PlcINPin: MMpn!Place (
50       id <- 'P_pin_' + InPin.name,
51       containerPage <- thisModule.resolveTemp(InPin.owner.owner, 'page'),
52       OutArcs <- ArcPin
53     ),
54     ArcPin: MMpn!Arc (
55       id <- 'A_pin_' + InPin.name,
56       containerPage <- thisModule.resolveTemp(InPin.owner.owner, 'page'),
57       --source <- PlcINPin--,
58       target <- thisModule.resolveTemp(InPin.owner, 'PInTrans')
59     )
60 }
61
62 --Out_putPin
63
64 lazy rule OutputPin {
65   from
66     OutPin: MMuml!OutputPin
67   to
68     ArcOutPin: MMpn!Arc (
69       id <- 'A_pin_' + OutPin.name,
70       containerPage <- thisModule.resolveTemp(OutPin.owner.owner, 'page'),
71       source <- thisModule.resolveTemp(OutPin.owner, 'POutTrans'),
72       target <- PlcOutPin
```

```
73      ),
74      PlcOutPin: MMpn!Place (
75        id <- 'P_pin_' + OutPin.name,
76        containerPage <- thisModule.resolveTemp(OutPin.owner.owner, 'page')
77      )
78  }
79
80  --Fork node
81
82  rule ForkNode {
83    from
84      fknd: MMuml!ForkNode
85    to
86      start: MMpn!Transition (
87        id <- 'T_' + fknd.name,
88        containerPage <- thisModule.resolveTemp(fknd.activity, 'page'),
89        InArcs <- thisModule.resolveTemp(fknd.incoming, 'outarc'),
90        --OutArcs <- thisModule.resolveTemp(fknd.outgoing.first(), 'in_arc'),
91        OutArcs <- fknd.outgoing -> collect(e | thisModule.resolveTemp(e, '
      inarc'))
92      )
93  }
94
95  --Join node
96
97  rule JoinNode {
98    from
99      joind: MMuml!JoinNode
100   to
101     trst: MMpn!Transition (
102       id <- 'T_' + joind.name,
103       containerPage <- thisModule.resolveTemp(joind.activity, 'page'),
104       InArcs <- joind.incoming -> collect(e | thisModule.resolveTemp(e, '
      outarc')),
105       -- InArcs <- thisModule.resolveTemp(joind.incoming.last(), 'out_arc')
      ,
106       OutArcs <- thisModule.resolveTemp(joind.outgoing, 'inarc')
107     )
108 }
109
110 --Merge node
111
112 rule MergeNd {
113   from
114     mergend: MMuml!MergeNode
115   to
116     trst1: MMpn!Transition (
117       id <- 'TIn1_' + mergend.name,
```

```
118       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
119       InArcs <- thisModule.resolveTemp(mergend.incoming.first(), 'outarc'),
120       OutArcs <- arc1
121     ),
122     trst2: MMpn!Transition (
123       id <- 'TIn2_' + mergend.name,
124       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
125       InArcs <- thisModule.resolveTemp(mergend.incoming.last(), 'outarc'),
126       OutArcs <- arc2
127     ),
128     arc1: MMpn!Arc (
129       id <- 'A1_' + mergend.name,
130       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
131       --source <- trst1,
132       target <- plc
133     ),
134     arc2: MMpn!Arc (
135       id <- 'A2_' + mergend.name,
136       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
137       --source <- trst2,
138       target <- plc
139     ),
140     plc: MMpn!Place (
141       id <- 'P_' + mergend.name,
142       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
143       OutArcs <- arc
144     ),
145     arc: MMpn!Arc (
146       id <- 'A_' + mergend.name,
147       containerPage <- thisModule.resolveTemp(mergend.activity, 'page')
148     ),
149     trst: MMpn!Transition (
150       id <- 'TOut_' + mergend.name,
151       containerPage <- thisModule.resolveTemp(mergend.activity, 'page'),
152       InArcs <- arc,
153       OutArcs <- thisModule.resolveTemp(mergend.outgoing, 'inarc')
154     )
155 }
156
157 --Decision node
158
159 rule DecisionNode {
160   from
161     dicsnd: MMuml!DecisionNode
162   to
163     start: MMpn!Transition (
164       id <- 'TIn_' + dicsnd.name,
165       containerPage <- thisModule.resolveTemp(dicsnd.activity, 'page'),
```

```
166       InArcs <- thisModule.resolveTemp(dicsnd.incoming, 'outarc'),
167       OutArcs <- arc
168     ),
169     arc: MMpn!Arc (
170       id <- 'A_' + dicsnd.name,
171       containerPage <- thisModule.resolveTemp(dicsnd.activity, 'page')
172     ),
```

Listing 7 – Les autres règle