Transformation des diagrammes d'activités SysML1.2 vers les réseaux de Petri dans un cadre MDE

Damien Foures

Université Paul Sabatier - Laboratoire d'Analyse et d'Architecture des Systèmes

Groupe Ingénierie Système et Intégration

Juin 2011

Université Paul Sabatier - Laboratoire d'Analyse et d'Architecture des Systèmes

Groupe Ingénierie Système et Intégration

Transformation des diagrammes d'activités SysML1.2 vers les réseaux de Petri dans un cadre MDE

AUTEUR :

Damien FOURES

Résumé

Cette étude cherche à automatiser la transformation des diagrammes d'activités (DA) vers les réseaux de Petri. En nous basant sur les spécifications de l'Object Management Group (OMG), nous avons établi les règles de transformation en ATLAS Transformation Langage (ATL) pour obtenir un modèle conforme à notre méta-modèle réseaux de Petri. La sémantique du diagramme d'activité a été validée à l'aide de la transformation PetriNet2Tina qui va permettre de vérifier de façon formelle la correspondance sémantique entre les deux modèles après transformation. Cette vérification est effectuée avec le "model-checker" TIme petri Net Analyser (TINA) et le langage Linear Temporal Logic (LTL). L'utilisateur devra simplement établir le diagramme d'activité à partir des exigences des parties prenantes, la transformation et la vérification étant automatiques. Le formalisme des Réseaux de Petri nous permet de fournir de précieuses informations sur le diagramme d'activité pour le jouer et le simuler.

Mots clés : MDE, ATL, TINA, transformation, vérification, Réseaux de Petri, SysML, OMG, VHDL-AMS, Ecore

Table des matières

Table des figures

CHAPITRE 1

Introduction

De plus en plus d'industriels s'intéressent à l'ingénierie système, définie comme une approche scientifique interdisciplinaire, dont le but est de formaliser et d'appréhender la conception de systèmes complexes avec succès. Il y a quelques années la complexité des systèmes avait amené cette nouvelle discipline qui permettait une meilleure gestion des projets. Aujourd'hui la complexité est telle que l'on cherche à automatiser certaine phase du cycle de développement du produit pour améliorer de façon significative le temps avant mise sur le marché (time to marker) et donc de réduire les coûts de développpement. Les enjeux supportés par un système justifient de plus en plus la mise en place de processus de vérification et de alidation, ceci le plus rapidement possible dans le cycle de dévelopement du produit. Les processus de Validation et de Vérification (V&V) visent à s'assurer le plus tôt possible que les systèmes et les produits sont conformes aux exigences requises et respectent les caractéristiques de conception attendues. L'étude vise à améliorer le processus de V&V en automatisant une partie du cycle. Ces travaux se placent dans le cadre d'une approche d'Ingénierie Dirigée par les Modèles (IDM), une spécialité récente de l'ingénierie qui a su trouver sa place dans une industrie où la réutilisation de composant logiciel ou matériel est monnaie courante. Une nouvelle voiture est à 85% une ancienne voiture.

L'IDM permet de réutiliser des modèles de formalisme appelés méta-modèles, qui sont adaptables à toutes les plateformes comme les composants d'une voiture. Certains organismes s'intéressent de plus en plus à l'IDM et cherchent à la normaliser : c'est le cas de l'OMG (Object Management Group). Cette association qui est à l'origine de langage de modélisation comme UML, a défini le langage SysML (System Modeling Language) qui se veut orienté système. Ce langage n'est pas formel, il est donc impossible d'établir des preuves sur la concordance entre la solution envisagée et celle qui a été élaborée. C'est pourquoi nous nous intéressons ici à la transformation du langage SysML et plus particulierement des diagrammes d'activités vers un langage formel, les réseaux de Petri.

Nous tenterons d'établir les concepts nécessaires à une transformation automatique des diagrammes d'activités conforme à la spécification définie par l'OMG vers les réseaux de Petri, de vérifier formellement cette transformation avant d'utiliser les réseaux de Petri pour jouer le diagramme d'activité SysML et ainsi obtenir des résultats permettant de définir si la solution adoptée est en adéquation avec les exigences de départ.

Cette étude présentera dans le chapite 2 le contexte avec les outils, les langages utilisés ainsi que l'approche d'ingénierie dirigée par les modèle et la méta-modélisation où un example simple permettra d'illustrer cette démarche. La partie suivante traitera de l'élaboration des deux méta-modèles : Diagrammes d'activités et Réseaux de Petri avant d'établir les liens au niveau modèle entre les principaux élements des diagrammes d'activités et les réseaux de Petri. Le chapitre 4 abordera l'ensemble de la mise en oeuvre de la transformation de modèle suivant les concepts de l'IDM avant de terminer avec deux exemples d'application, le "butterfly", qui montrera les possibilités de gestion de la donnée fournie par la solution proposé dans ce mémoire et un exemple plus proche du monde industriel avec le résumé d'une étude complète d'un système de calcul d'injection de moteur thermique.

CHAPITRE 2

Contexte

2.1 L ?Environnement de travail

Les systèmes industriels actuels sont d'une complexité telle qu'il n'est plus envisageable de les développer sans un outil adéquat, souvent spécialisé dans un domaine. Ces outils et leurs langages sont souvent trop pointu pour une conception globale et directe. Il faut donc effectuer une conception amont, dite de haut niveau. Elle permet de garder une vue d'ensemble du projet au plus près des besoins du client sans entrer dans les détails techniques de réalisation. Aujourd'hui ces outils sont multidisciplinaires et permettent, par une gestion hiérarchique, de rendre transparentes à un non initié certaines parties du projet décrivant les solutions techniques. Ce chapitre présente ces outils mais aussi l'ensemble des outils et langages utilisés durant ces travaux.

2.1.1 SysML, un profil UML

SysML est un langage de modélisation graphique développé par l'OMG et l'INCOSE (The International Council on Systems Engineering) un organisme sans but lucratif qui a pour mission de partager, promouvoir et faire progresser l'ingénierie système. SysML est un profil d'UML 2.0 orienté système. La majorité des concepts présents dans SysML sont issus d'UML. Les concepteurs de ce langage insistent sur le fait que SysML tout comme UML n'est pas une méthode (description des étapes de mise en oeuvre) de modélisation mais bien un langage de modélisation graphique. Il est composé de 8 graphes (voir figure 2.1)

Les différents diagrammes de Sysml peuvent être décrits succinctement de la manière suivante :

Diagrammes Structurels :

– Diagramme de Définition de Bloc : *Block Definition Diagram (BDD)* est une représentation des entités du système, de leurs propriétés, de leurs arguments et de leurs opérations. Il est le

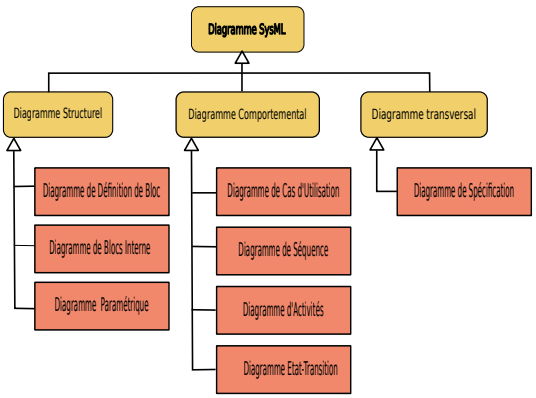

FIG. 2.1 – Les Diagrammes SysML.

pendant des Diagrammes de Classe (*Class Diagram*) UML.

- Diagramme de Bloc Internes : *Internal Block Diagram (IBD)* modélise les interconnections entre les blocs et les imbrications entre eux sous forme de boîte blanche. Il est l'équivalent du Diagramme de composants (*Composite Structure Diagram*) UML.
- Diagramme Paramétrique : *Parametric Diagram (Par)* modélise les paramètres physiques du système en vue d'un test de performance du système. Il n'a pas d'équivalent UML.

Diagrammes Comportementaux :

- Diagramme de Cas d'Utilisation : *Use Case Diagram (UC)* modélise le fonctionnement général du système et ses interactions avec le monde extérieur. On retrouve ce diagramme tel-quel dans UML.
- Diagramme de Séquence : *Sequence Diagram (SD)* modélise la chronologie des interactions avec les autres blocs (classes) du système modélisé pour un scénario donné. On retrouve ce diagramme tel-quel dans UML.
- Diagramme d'Activités : *Activity Diagram (Act)* modélise le flux de contrôle et de donnée du système de façon hiérarchique. On utilise ce diagramme de la même façon dans UML malgré quelques différences dans la syntaxe du langage.
- Diagramme Etat-Transition : *StateMachine Diagramme (STM)* modélise les différents états que peut prendre un élément du système. On retrouve ce diagramme tel-quel dans UML.

Diagramme Transversal :

- Diagramme d'Exigences :*Requirement Diagram (N/A)* permet d'organiser toutes les exigences textuelles définies par le client pour le système à modéliser. Il n'a pas d'équivalent UML.

2.1.2 Réseaux de Petri

Les Réseaux de Petri (RdPs) sont un formalisme mathématique. Il existe une représentation graphique et une représentation matricielle (mathématique) de ce formalisme aidant à la modélisation dans de nombreux domaines. Apparu pour la première fois dans la thèse de Carl Adam Petri en 1962,

les réseaux de Petri ont depuis été décliné en de nombreuses classes :

– Rdp Coloré, Rdp Hiérarchique, Rdp Stochastique, Rdp Prédicat-Transition, Rdp Temporel, Rdp Temporisé, Rdp Prédicat-Transition différentiel, ...

Le but est d'appliquer le formalisme des RdPs à des domaines variés et ne plus se limiter strictement aux systèmes à événements discrets. La possibilité de gestion des systèmes complexes avec les RdPs n'est plus à prouver [1, 2]. Il n'est pas essentiel de présenter ici plus en détail les RdPs et le formalisme associé, car celui-ci fait l'objet d'une étude particulière pour en extraire le méta-modèle.

2.1.3 Eclipse Modeling Framework

L'outil principal utilisé durant ces travaux est Eclipse Modeling Framework (EMF), c'est un kit de composants logiciels structurels (plus courament appelé framework) pour la construction d'outils basées sur une approche modèle. Les modèles y sont décrits en XML, mais peuvent être spécifiés dans des documents UML/SysML (solution utilisée durant ces travaux). L'interopérabilité avec d'autres outils basés sur Eclipse est un des points forts de ce framework. Un de ces outils appelé Graphical Modeling Framework (GMF) permet le développement d'éditeur graphique de modèle. L'utilisation de GMF apporterait à ces travaux un rayon d'utilisation plus important grâce à un éditeur graphique spécifique aux diagrammes d'activités. Le langage de transformation de modèle ATL (Atlas Transforming Language), a été intégré à EMF et fait l'objet d'une présentation plus détaillée dans le chapitre 4.

2.1.4 TOPCASED

Des boites à outils peuvent être ajoutées à EMF pour créer un environnement de travail orienté vers un domaine précis. C'est le cas de TOPCASED (Toolkit in Open Source for Critical Applications & Systems Development). TOPCASED est un logiciel d'ingénierie assisté par ordinateur. Il ajoute à EMF des fonctionnalités essentielles à ce projet liées à la mise en oeuvre des modèles SysML sous forme graphique.

La figure 2.2 montre l'environnement de travail TOPCASED, à gauche on peut voir la version graphique éditée grâce à TOPCASED, à droite le résultat sous forme arborescente stocké sous fichier XMI. Le tout est ensuite exploitable pour la transformation de modèle.

2.1.5 TINA toolbox

TIme petri Net Analyzer est une boîte à outils pour l'édition et l'analyse des RdPs. TINA a été développé au saint du Laboratoire d'Analyse et d'Architecture des Systèmes de Toulouse, il possède un "model-checker" SELT qui permet la saisie et la vérification des propriétés LTL (Linear Temporal Logic) sur les structures de Kripke définient à partir des RdPs. Cet outil servira à vérifier de façon formelle la transformation vers les réseaux de Petri.

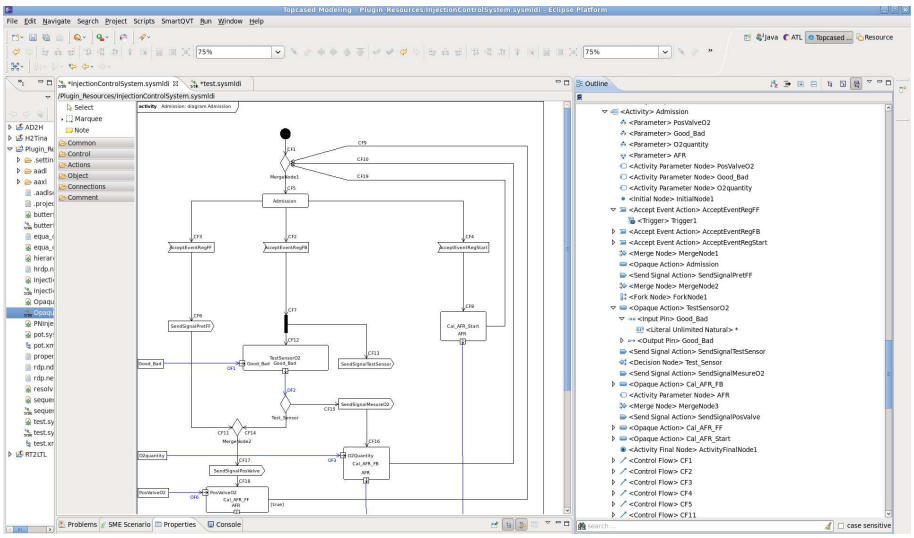

FIG. 2.2 – Création d'un diagramme d'Activités SysML sous environnement TOPCASED

2.1.6 Mentor Graphics - SystemVision

A ce jour il n'existe pas d'outil capable d'effectuer une vérification formelle en parallèle d'une validation par simulation sur les réseaux de Petri. C'est pourquoi nous avons également utilisé SystemVision, une plateforme de simulation spécialisée dans le designe et l'analyse de système électronique afin de simuler la partie continue des réseaux de Petri prédicats-transition-différentiels tout en respectant le comportement discret des RdPs.

2.2 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 de l'ingénierie système, elle fournit les outils et 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 à conduit les développeurs à manipuler des concepts de plus haut niveau. L'Object Management Group a fini par s'intéresser à l'Ingénierie Dirigée par les Modèles et a proposé en 1997 le standard MOF (Meta-Object Facility). On parle également parfois de MDA (Model Driven Architecture) qui est une marque déposée de OMG qui reprend et restreint les concepts de l'IDM.

2.2.1 La méta-modélisation

Un méta-modèle est une définition d'un point de vue donné d'un langage de modélisation, c'est le modèle du modèle. A première vue un modèle ne ressemble à aucun autre modèle. Suivant le système à modéliser, suivant la vision qu'a le concepteur du système, suivant sa maîtrise du langage de modélisation, le modèle résultant sera différent. Pourtant tous ces modèles sont regroupés sous une même syntaxe et une même sémantique. Il y a un langage, un formalisme qui peut lui même

être modélisé. C'est le rôle premier de la méta-modélisation. Comme il est dit dans la première partie de ce chapitre l'OMG à définie MDA, qui préconise une vision sur 4 niveaux de modélisation au travers du MOF [3]. La figure 2.3 montre une représentation graphique du MOF et ses quatre niveaux d'abstraction.

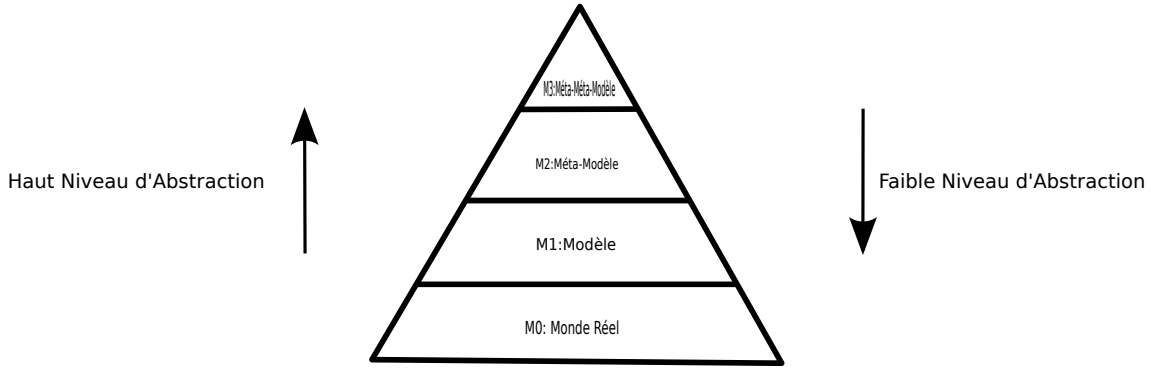

FIG. 2.3 – Modèle du MOF.

Example Afin de mieux comprendre la signification de ces 4 niveaux d'abstraction nous allons prendre l'exemple simple d'une carte IGN figure 2.4. La vue aérienne du LAAS est une vision réelle

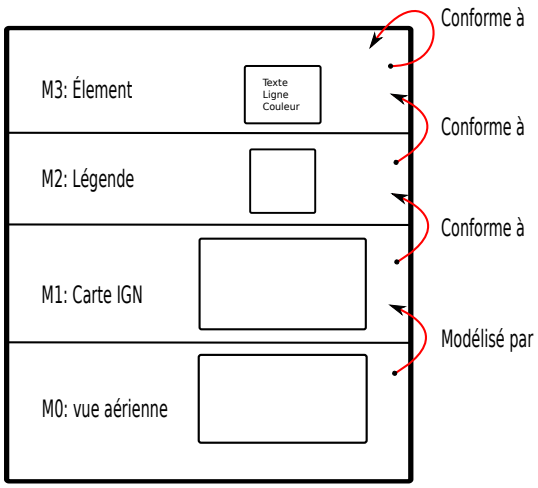

FIG. 2.4 – Example de modélisation

du laboratoire (M0) à un instant donné. C'est le système à modéliser. Il faut maintenant établir le modèle (M1). Ici le choix s'est porté sur un modèle type carte topographique (type IGN top 25), en Ingénierie Dirigé par les modèles on parle de choix du formalisme. Le modèle aurait put être une carte thermique ou géologique. C'est donc une des façons de modéliser le laboratoire. Cette carte, pour être comprise par tous se doit d'être conforme à sa légende (M2). C'est le méta-modèle des cartes IGN. L'ensemble des cartes IGN sont conformes à une seule et même légende. Tout les concepts d'une carte IGN sont présents dans la légende (le méta-modèle). Si la légende est modifiée, l'ensemble des instances du méta-modèle (l'ensemble des modèles, l'ensemble des cartes IGN) est modifié. Imaginons maintenant une carte thermique du laboratoire, il est évident que la carte ne sera pas la même, la légende non plus, pourtant les deux légendes auront des points communs. C'est le méta-méta-modèle

(M3), c'est lui qui décrit les concepts clef présents dans les méta-modèles. Pour construire une légende il faut : du texte,des couleurs,...Toutes les légendes sont conformes à ce méta-méta-modèle. Il est indispensable que ce niveau M3 soit conforme à lui-même, afin d'éviter un empilement des niveaux. Ici du texte permet de définir le méta-méta-modèle. Il est donc conforme à lui même.

C'est un méta-méta-modèle qui permet de définir tout les autres méta-modèles. Il existe en réalité deux méta-méta-modèles : le EMOF (Essentiel MOF) qui comme son nom l'indique ne reprend que les concepts clef, le CMOF (Complet MOF) qui quand à lui reprend l'ensemble des concepts. Ils sont tous deux définis dans le standard MOF [3]. Un troisième méta-méta-modèle a été proposé, le SMOF (Semantic MOF) mais n'a pas encore était validé par l'OMG. Il est également possible d'utiliser d'autres méta-méta-modèles comme Ecore [4] proposé dans Eclipse Modelling Framework (EMF), qui n'est pas une norme mais un langage de méta-modélisation alligné sur l'EMOF.

2.2.2 La transformation de modèle

Un des concepts les plus importants de l'IDM est la transformation de modèle : partir d'un modèle "source" et aller vers un modèle "cible". L'IDM distingue deux types de transformation (voir figure 2.5 :

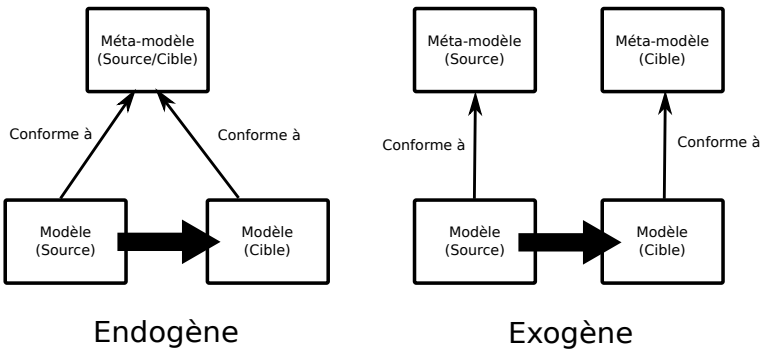

FIG. 2.5 – Type de Transformation

- 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-modèle différents (e.g. : d'un SysML vers un RdP).

Notons une convention implicite, les transformations sont toujours nommées par : nom-source2nom-cible (e.g. :ActivityDiagram2PetriNet)

Il faut établir des règles de transformation pour passer du modèle source au modèle cible. Classiquement, il suffit d'élaborer les règles à partir des éléments du modèle source pour les transformer en l'élément désiré du modèle cible. Cette approche pose un problème évident. La transformation n'est utile que pour un seul modèle. L'IDM et la méta-modélisation permettent d'établir des règles à un niveau d'abstraction plus élevé (M2 du MOF), d'établir les règles entre un méta-modèle source et un méta-modèle cible. Les règles sont donc définies pour toutes les instances du méta-modèle source (tous

les modèles source). Il y a alors possibilité d'automatiser la transformation. Pour l'écriture des règles de transformation au niveau méta-modèle, l'OMG à défini le standard QVT [5] (Query/View/Transformation). A 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 [6]. Notre choix s'est porté sur le langage ATL (ATLAS Transformation Language) [7] qui est intégré à la boîte à outils TOPCASED [8] d'Eclipse [9].

Principe de la transformation de modèle selon le méta-méta-modèle Ecore :

Cette démarche a été expliquée brièvement dans le paragraphe précédent. La figure 2.6 montre plus en détail le fonctionnement de la transformation dans l'environnement de travail choisi en vue d'une transformation ActivityDiagram2PetriNet. Pour effectuer une transformation de modèle, il faut :

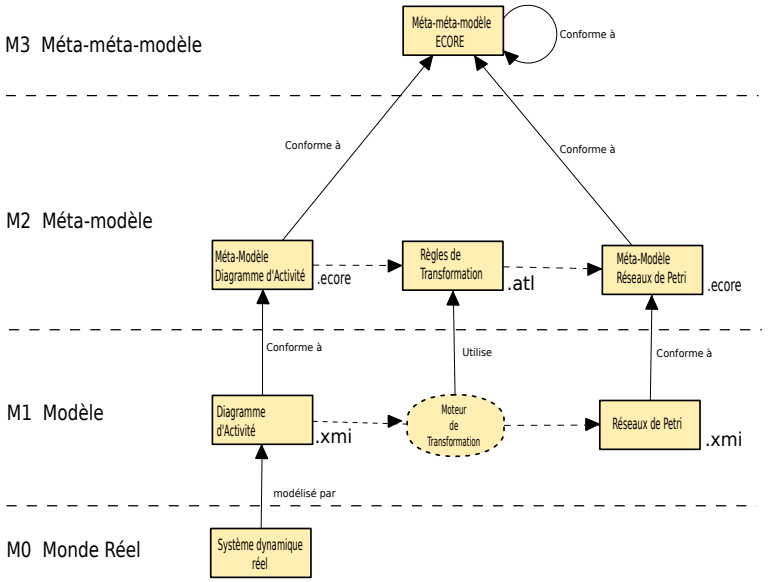

FIG. 2.6 – Concept de la transformation

- Le méta-méta-modèle : Ecore
- Le méta-modèle source conforme au MMmodèle Ecore qui décrit le langage de modélisation source : Les Diagrammes d'activité SysML
- Le méta-modèle cible conforme au MM Ecore qui décrit le langage de modélisation cible : Les réseaux de Petri
- Le modèle source qui décrit un système en respectant la sémantique et la syntaxe des Diagrammes d'activité
- Les règles de transformation en ATL.

Cet ensemble permet la création automatique du modèle cible : un réseau de Petri conforme au méta-modèle cible grâce au moteur de transformation. La conception des règles s'établit au niveau M2, mais il est plus aisé de conceptualiser les principes de la transformation au niveau modèle (M1), on parle de "Mapping des Concepts". L'ensemble des travaux présentés dans ce mémoire s'effectue au niveau Méta-modèle, mais la réflexion sur les choix de transformation s'est effectuée au niveau inférieur

(M1) par des exemples simples de transformation. De nombreux autres concepts sont présents dans l'IDM, qui ne sont pas utiles aux travaux présentés dans ce mémoire et ne seront donc pas développés.

CHAPITRE 3

Les concepts

Le but de ces travaux est de réaliser une transformation des diagrammes d'Activité vers les réseaux de Petri en respectant les concepts MDA et donc les spécifications définies par l'OMG. Ce chapitre définie le concept de méta-modélisation appliqué au problème de transformation DA2RdP. Une deuxième partie mettra en avant le mapping des concepts pour passer d'un modèle à l'autre.

3.1 Les Méta-modèles

Cette section présente les méta-modèles source et cible, dans un premier temps le formalisme sera mis en avant, avant de modéliser celui-ci en faisant apparaître la démarche de méta-modélisation.

3.1.1 Méta-modèle Diagramme d'Activité (MMDA)

Le diagramme d'activité est un des quatre diagrammes comportementaux de SysML. Il est utilisé pour décrire le comportement du flux de contrôle et de données des systèmes complexes. Il est donc parfaitement adapté pour décrire le déroulement d'un cas d'utilisation. Il est composé de noeuds et d'arcs. Ces noeuds sont divisés en deux parties, les noeuds de contrôle et les noeuds d'activités. Chacun de ces noeuds est ensuite spécialisé et peut grâce aux arguments associés avoir un comportement sensiblement différent, c'est pourquoi nous ne présenterons pas en détail chacune des possibilités offertes par SysML et préférons renvoyer à la spécification OMG [10, 11]. Il sera tout de même présenté dans ce paragraphe les principaux noeuds du diagramme.

Nous avons extrait le méta-modèle du diagramme d'activité de la spécification SysML [10] et UML [11] de l'OMG. En effet, l'OMG spécifie le diagramme d'activité SysML en ce basant sur celui d'UML. Il faut donc extraire des informations des deux documents. Il semble plus judicieux pour créer le méta-modèle d'établir une liste des noeuds et concepts que l'on veut voir apparaître dans son méta-modèle et compléter le méta-modèle à partir de ces blocs de base. Il semble difficile d'extraire le MMDA

autrement. La figure 3.1 présente une traduction en diagramme de classe UML du MMDA présent dans la spécification OMG. Ce méta-modèle ne se veut pas exhaustif mais représente la majeure partie des concepts présents dans les diagrammes d'activités.

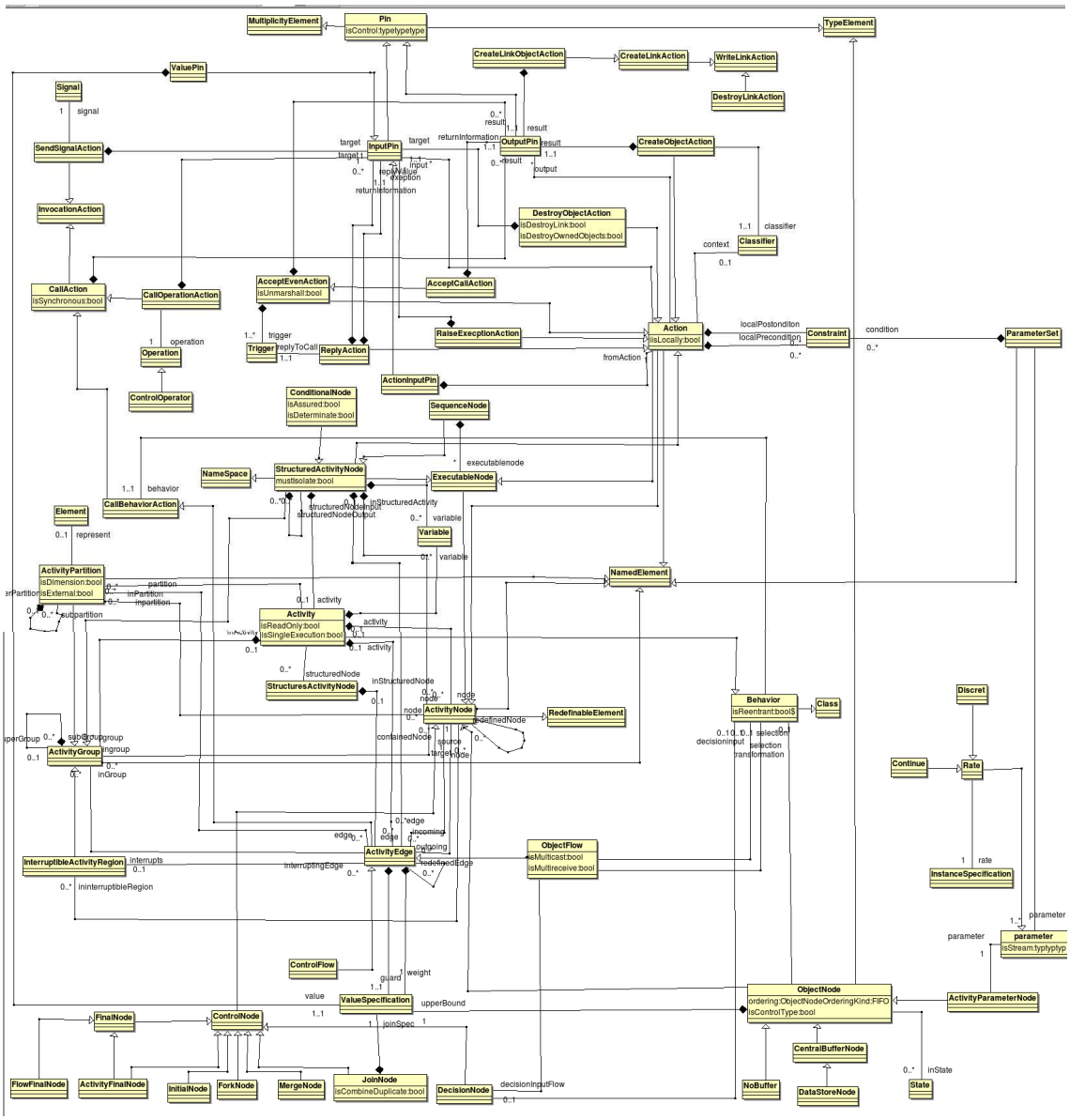

FIG. 3.1 – Méta-modèle Diagramme d'Activités extrait de la spécification OMG

Présenter l'ensemble de ce méta-modèle reviendrait à écrire la spécification OMG, ce chapitre ne présentera qu'une petite partie de ce méta-modèle, figure 3.2.

Comme dit précédemment, les diagrammes d'activité sont composés de noeuds, certains de ces noeuds sont de type controlNode. Nous allons nous intéresser ici à ce type de noeuds. Pour ce faire, il faut dans un premier temps regarder quels sont les noeuds de contrôle connus par une recherche bibliographique succincte, l'excellent cours en ligne de Mr Laurent Audibert [12] nous servira de base (dans cet example) pour établir cette recherche. Nous retrouvons ces noeuds dans la spécification SysML et UML, il est alors possible de créer la classe équivalente du noeud étudié et d'établir les liens avec les autres classes. Cette démarche a été appliquée pour l'ensemble du méta-modèle présenté en

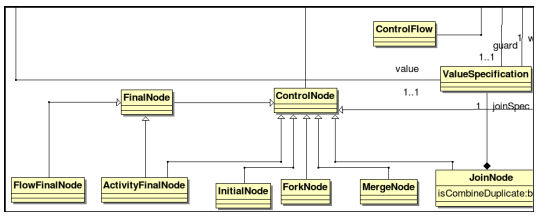

FIG. 3.2 – Partie du Méta-modèle Diagramme d'Activités extrait de la spécification OMG : Autour du ControlNode

figure 3.1.

Noeuds de contrôle :

Dans le cours d'UML/SysML [12] il est dit : Il existe plusieurs types de noeuds de contrôle :

- noeud initial *(initial node)* ;
- noeud de fin d'activité *(final node)*
- noeud de fin de flot *(flow final)* ;
- noeud de décision *(decision node)* ;
- noeud de fusion *(merge node)* ;
- noeud de bifurcation *(fork node)* ;
- noeud d'union *(join node)*.

Les spécifications de l'OMG pour le noeud de fin d'activité :

Intéressons nous maintenant plus particulièrement au noeud de fin d'activité, et regardons la spécification OMG associée.

Extrait de la spécification OMG [11] : 12.3.6 ActivityFinalNode (from BasicActivities, IntermediateActivities)
An activity final node is a final node that stops all flows in an activity

- Generalizations :
  ControlNode (from BasicActivities) on page 366
  FinalNode (from IntermediateActivities) on page 384
- Description :
  An activity may have more than one activity final node. The first one reached stops all flows in the activity.
- Attributes :
  No additional attributes
- Associations :
  No additional associations
- Semantics :
  ...
- Constraints :
  No additional constraints

On peut voir ici que le noeud de fin d'activité hérite du noeud de contrôle et du noeud final, il ne possède ni attribut, ni lien d'association, ni contraintes supplémentaires. Il faudra cependant se pencher plus en détail sur la sémantique par la suite pour être sûr qu'il n'y ait pas de contrainte sur le méta-modèle. Si tel est le cas, il faudra établir des contraintes OCL (Object Constraint Language) [13], pour un méta-modèle plus détaillé. A noter qu'il est parfois plus complexe d'extraire le méta-modèle,

en effet les informations sont parfois disséminées dans la spécification.

Le diagramme équivalent :

A partir de ces informations recueillies dans la spécification OMG, il est possible d'établir une représentation graphique figure 3.3 de la méta-classe (classe du niveau méta-modèle) "ActivityFinalNode" : une méta-classe possédant deux liens d'héritage avec les méta-classes "FinalNode" et "ControlNode".

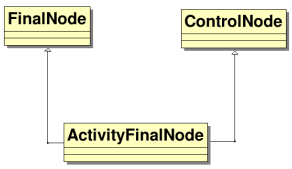

FIG. 3.3 – Construction de la méta-classe ActivityFinalNode

Afin d'établir le méta-modèle complet du Diagramme d'Activité, il faut effectuer cette opération pour chaque concept du DA. Pour rester conforme à l'OMG il était indispensable d'effectuer ce travail le plus précisément possible. Une fois le méta-modèle établi, il s'est avéré plus intéressant d'utiliser le méta-modèle SysML présent dans EMF et la bôite à outils TOPCASED, et qui après vérification, est conforme à l'OMG (pour les parties utilisées). L'intérêt majeur de ce choix, réside dans l'existence d'un editeur graphique de diagramme d'activité dans TOPCASED.

3.1.2 Réseaux de Petri

Maintenant que le méta-modèle source a été créé il faut définir le méta-modèle cible. Dans l'ingénierie dirigée par les modèles on pourrait dire : il faut modéliser le formalisme des RdP, ou bien encore méta-modéliser les RdPs. Avant tout, il est important de redéfinir clairement les réseaux de Petri voulu pour cette transformation afin d'établir un méta-modèle conforme à nos attentes, et compatible avec les outils d'analyse définis au chapitre 2 (TINA et SystemVision). Si le méta-modèle est compatible avec ces outils, alors on peut assurer que toutes les instances de ce méta-modèle (tous les RdPs engendrés par la transformation) seront compatibles. Cette technique de méta-modélisation n'est pas totalement dans l'esprit de l'IDM. Il serait plus intéressant d'établir un méta-modèle totalement indépendant des plateformes (TINA, SystemVision), une transformation pourra par la suite adapter le méta-modèle à la plateforme. Le méta-modèle a ici été orienté vers la plateforme TINA pour obtenir une transformation "model2text" plus aisée.

Un réseau de Petri est un graphe biparti, il est composé de places et de transitions. Les places sont représentées par des cercles et les transitions par des barres. Des arcs orientés relient deux noeuds de type différent entre eux. Un réseau de Petri binaire marqué contient un nombre fini de marques, appelé jetons. Une place peut contenir ou non des jetons. Cette répartition à un instant donné décrit l'état discret du système.

Le formalisme des Réseaux de Petri prédicats-transitions-différentiels :

Pour notre transformation de modèle cette vision des RdPs n'est pas suffisante. Les diagrammes d'activités peuvent décrire un comportement hybride, discret et continu, ainsi que l'aspect hiérarchique des systèmes. Il faut donc utiliser des réseaux de Petri plus riches. Nous utiliserons les RdPs prédicat-transitions-différentiels (RdP-PTD) définis dans [14]. Ces réseaux permettent la gestion d'équations différentielles et donc de l'aspect continu des diagrammes d'activités. Des travaux précédents [15] ont permis d'élaborer un méta-modèle RdP de façon progressive, seule la solution finale après modification sera détaillée.

Le formalisme présenté dans [14] propose de combiner les réseaux de Petri classique servant à décrire l'aspect discret du système à un système d'équations différentielles algébriques (réseaux prédicats-transitions-différentiels) servant à modéliser le comportement continu du système.

Réseaux de Petri Ordinaire :

La description formelle de ce modèle est définie par un 6-uplet :

$$PN =< P, T, M_0, A, Pre, Post > \ avec :$$

- P, un ensemble fini de places, T, un ensemble fini de transitions,
- M_0, le marquage initial du réseau, A, un ensemble fini d'arcs tel que

$$P \cap T = P \cap A = T \cap A = 0 \,,$$

  , Pre, indique combien de jetons sont consommés depuis une place vers une transition, Post, indique combien de jetons sont produits par une transition dans la place avale.

Réseaux de Petri prédicats-transitions :

Un réseau de Petri prédicats-transitions initialement marqué est un triplet :

$$PN_{PT} =< PN, A, M_0 > \ avec :$$

- PN un réseau de Petri ordinaire <P, T, M_0, A, Pre, Post>.
- A est l'annotation de PN_{PT}, A = < C_{onst} , X, A_{tc}, A_{ta}, A_c > avec
  - X l'ensemble des variables de A, tel que :

$$X = \cup_{p_k \in p} X_{p_k}$$

    , avec X_{p_k} l'ensemble des variables attachées aux places p_k.
  - $A_{tc} : T \rightarrow L_c(X)$, une application associant à chaque transition une condition L_c sous la forme d'un prédicat utilisant les variables.
  - $A_{ta} : T \rightarrow L_a(X)$, une application associant à chaque transition une action L_a sous la forme d'une suite d'affectations de valeurs aux variables.
  - Ac une application associant à chaque arc une somme formelle de N-uplets d'éléments de X telle que son module soit égal au poids de l'arc correspondant :

$$||A_c(p,t)|| = C(p,t)$$

(Avec C = Post - Pre)

– M_0 est le marquage initial : à chaque place p_k est associée une somme formelle de N-uplets de constantes C_{onst}

Réseaux de Petri prédicats-transitions-différentiels :

Le formalisme [14] décrit les RDP-PTDs comme la combinaison d'un RDP-PT vu précédemment et d'équations algébriques différentielles. Soit la paire :

$$PN_{PTD} =< PN_{PT}, F > \ avec :$$

– PN_{PT} un réseau prédicats-transitions défini par < R, A, M_0 >.

– F l'ensemble des équations différentielles associées aux places ; cet ensemble de fonctions définit l'évolution en fonction du temps des variables X associées aux jetons .

$$F = \begin{pmatrix} F_1(\dot{X}_1, X_1, \theta) \\ \vdots \\ F_n(\dot{X}_n, X_n, \theta) \end{pmatrix} \ avec\ n\ le\ nombre\ de\ place$$

– T, l'ensemble des transitions,

$$card(T) = m\ , m\ étant\ le\ nombre\ de\ transition$$

– X_{in_i} est l'union des ensembles des variables associées aux places d'entrée de la transition t_i :

$$X_{in_i} = \cup_{p_k \in (.,t_i)} X_{p_k}$$

– X_{out_i} est l'union des ensembles des variables associés aux places de sortie de la transition t_i :

$$X_{out_i} = \cup_{p_k \in (t_i,.)} X_{p_k}$$

Si l'on associe à F l'annotation A on obtient :

– S, l'expression particulière de A_{tc}, qui associe à chaque transition t_i, une fonction de sensibilisation (*définition voir* [16]).

$$S = \begin{pmatrix} s_1(\dot{X}_{in_1}, X_{in_1}, \theta) \\ \vdots \\ s_m(\dot{X}_{in_m}, X_{in_m}, \theta) \end{pmatrix}$$

Soit s_i une fonction de sensibilisation associée à la transition t_i. La première solution de la fonction de sensibilisation

$$s_i(\dot{X}_{ini}, X_{ini}, \theta)$$

est définie comme le seuil que doit atteindre la variable X_i afin de sensibiliser la transition t_i. Une fois le seuil atteint, un évènement est détecté et la transition t_i est sensibilisée .

– J, l'expression particulière de A_{ta} associe une fonction de jonction à chaque transition t_i. (*définition voir* [16]).

$$J = \begin{pmatrix} j_1(\dot{X}_{out_1}, X_{out_1}, \dot{X}_{in_1}, X_{in_1}, \theta) \\ \vdots \\ j_m(\dot{X}_{out_m}, X_{out_m}, \dot{X}_{in_1}, X_{in_1}, \theta) \end{pmatrix}$$

Soit j_i la fonction de jonction associée à la transition t_i :

$$\dot{X}_{outi}(\theta^+) = j_{i\dot{X}}(\dot{X}_{in_i}, X_{in_i}, \theta^-)$$

$$X_{outi}(\theta^+) = j_{iX}(\dot{X}_{in_i}, X_{in_i}, \theta^-)$$

j_i calcule la valeur des variables X_i, associées à la transition i, à la date de franchissement θ. Une nouvelle valeur initiale sera alors appliquée à chaque X_i en cas de discontinuité.

Cette définition formelle se traduit par le fonctionnement suivant dans un réseau de Petri :

- Un jeton mis dans une place déclenche l'activation de la résolution des équations différentielles attachées à cette place.
- Parallèlement, un certain nombre de seuils sont surveillés. Chaque seuil étant associé à une transition aval d'une place. La première équation de la fonction de sensibilisation qui atteint le seuil crée un évènement détecté par la transition associée.
- La transition est sensibilisée, il y a tir de la transition, un nouveau marquage est calculé grâce aux fonctions de jonction associées à cette même transition.
- De nouvelles places sont marquées, il y a déclenchement de nouvelles résolutions d'équation attachées aux places marquées.

Méta-modèle Réseaux de Petri prédicats-transitions-différentiels

Avant d'établir le méta-modèle des RdP-PTD, il est intéressant de montrer le principe de création du méta-modèle RdP. La figure 3.4 présente un méta-modèle RdP simplifié.

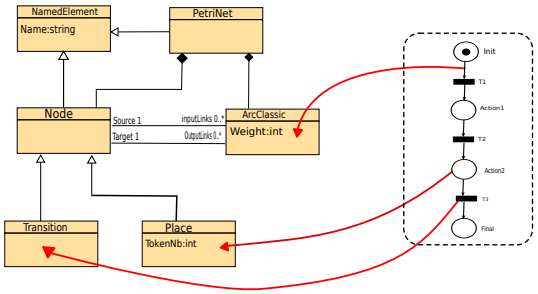

FIG. 3.4 – Principe de méta-modélisation des RdPs

Un *PetriNet* (réseau de Petri) est composé de *Node* (noeud) et d'*ArcClassic* (arc). Un *Node* peut être une *Transition* ou une *Place* reliés par des *ArcClassic*. Un *ArcClassic* ne peut avoir qu'une seule *Source* et qu'un seul *Target* (cible). On voit également qu'un *PetriNet*, un *Node*, une *Transition*, une *Place* reçoit par héritage un attribut de type *Name* permettant de nommer explicitement les instances de ces méta-classes.

Le méta-modèle figure 3.5 représente le formalisme RdP-PTD où :

- La classe *Hiles* implémente un réseau de Petri.
- La classe *LinkageElement* implémente un arc d'un réseau de Petri. Cet arc peut être de type classique (*ArcClassic*) ou inhibiteur (*ArcInhib*).

- La classe *NodeElement* implémente un noeud d'un réseau de Petri. Ce noeud peut être une place (*Place*) ou une transition (*Transition*). Un *NodeElement* possède un nom et un identificateur unique pour le référencer.
- La classe *FonctionalBlock* implémente un bloc "Action" RdP. Il peut implémenter une action de type à remise à zéro explicite (*ExplicitAction*) ou à remise à zéro implicite (*ImplicitAction*) ou un procédé.
- La classe *StructuralBlock* implémente un ensemble des noeuds du RdP d'un niveau d'abstraction donné. Elle permet la structuration et le découpage par composition/décomposition hiérarchique du modèle.
- La classe *AnalogAction* implémente une variable d'un système discontinu.
- La classe *DAE* implémente une équation différentielle.
- La classe *SensitiveFunction* implémente une fonction de sensibilisation.
- La classe *JunctionFunction* implémente une fonction de jonction.
- La classe *WeightedElement* implémente une fonction poids/nombre de jeton associée aux places et aux arcs.

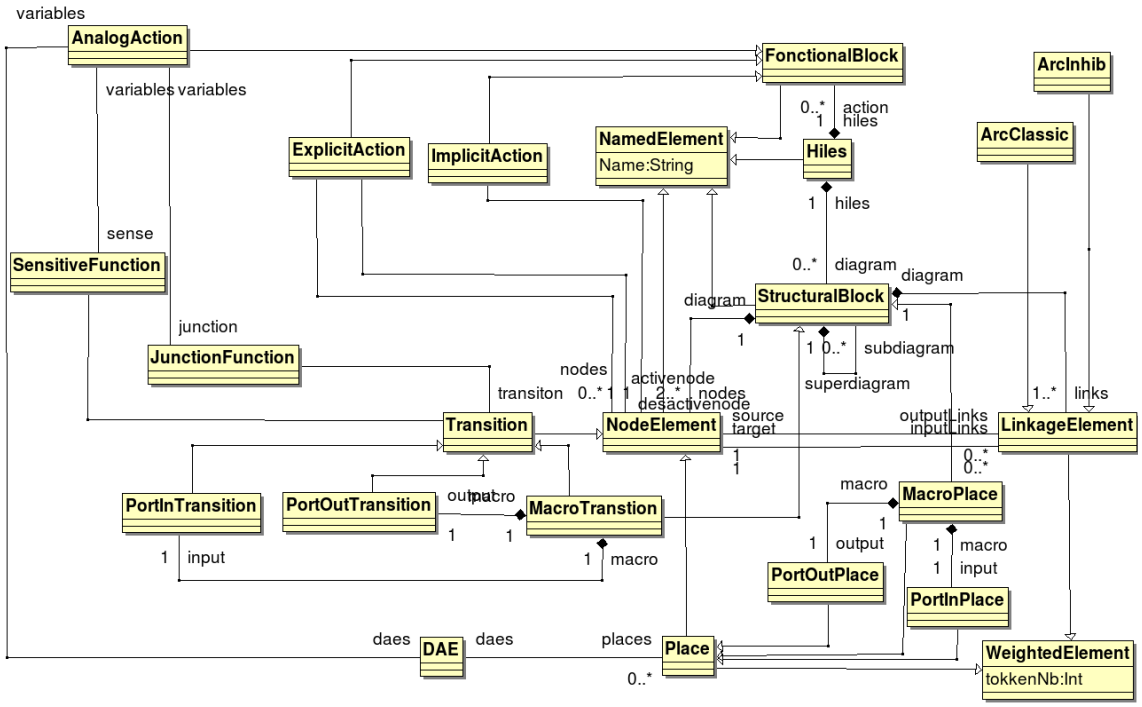

FIG. 3.5 – Méta-modèle réseaux de Petri prédicats-transitions-différentiels

L'implémentation des méta-modèles sous TOPCASED doit être conforme au méta-méta-modèle ECORE. TOPCASED utilise une description arborescente pour ces réseaux de Petri. On peut observer l'implémentation choisie pour ce méta-modèle figure 3.6.

Les deux méta-modèles étant maintenant définis conformément au méta-méta-modèle Ecore de TOPCASED, il est possible d'établir les règles de transformation, pour ce faire il faut avant tout établir le concept de ces règles, c'est l'objet de la prochaine partie.

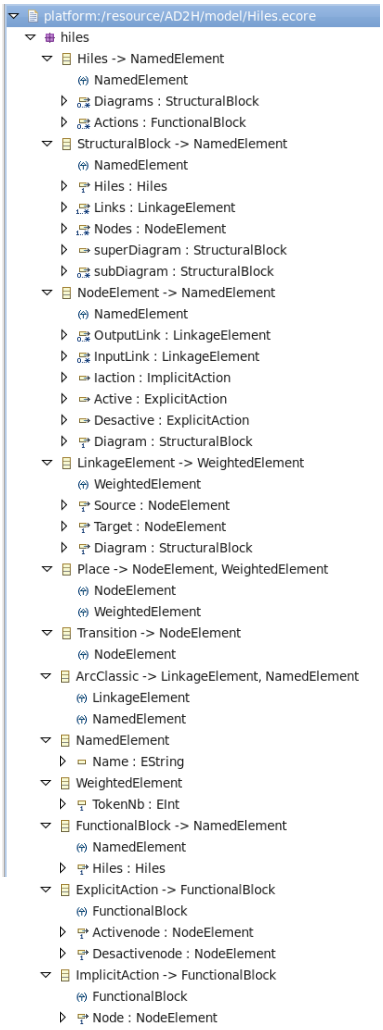

FIG. 3.6 – Méta-modèle réseaux de Petri dans l'environnement TOPCASED

3.2 Mapping des concepts

Cette contribution cherche à automatiser la transformation des diagrammes d'activités vers les réseaux de Petri. Il est essentiel d'avoir une vision au niveau modèle des résultats attendus. Ceci afin d'assurer une conservation des propriétés, et du comportement des diagrammes d'activités, dans le RdP équivalent. C'est le passage d'un langage semi-formel standardisé au langage formel.

3.2.1 Création de réseaux complexes

L'approche adoptée pendant ces travaux a été d'élaborer des blocs de base avant de les interconnecter. Cette solution à l'avantage d'être générique. Tous les blocs RdP créés le sont selon le même schéma (Transition-Place-Transition), voir figure 3.7.

Seuls les artefacts *ControlFlow et ObjectFlow* utilisent un schéma Arc-Place-Arc afin de connecter les blocs de base entre eux. Cette solution n'est pas idéale car elle engendre des réseaux de Petri parfois complexes qui peuvent être simplifiés mais ne respectent plus la sémantique des DAs.

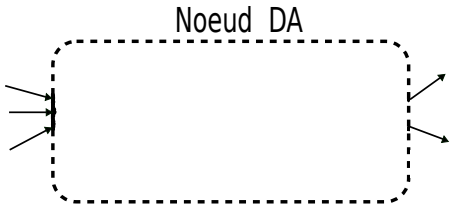

FIG. 3.7 – Définition du bloc de base.

3.2.2 Des artefacts du diagramme d'activité vers les blocs réseaux de Petri

Ce paragraphe va établir les concepts de la transformation entre les entités atomiques du DA (noeud/arc) et un bloc équivalent RdP. Ces blocs devront respecter le comportement (sémantique) du DA préconisé par l'OMG [11, 10].
A première vue, un réseau de Petri peut sembler proche d'un diagramme d'activité. Leurs représentations graphiques et leurs comportements (déplacement de jetons dans le "réseau", composition des graphes : noeud,arc) peut donner à un non initié l'impression d'avoir deux techniques de modélisation très proches. Ce paragraphe montre la complexité non apparente de la sémantique des DAs.

3.2.3 Noeud initial (figure 3.8)

Description issue de la spécification OMG : Un noeud initial est un noeud de contrôle qui est activé quand l'activité est invoquée.

Propriété issue de la spécification OMG :

- Des jetons dans un noeud initial sont offerts à tous les arcs sortants.
- Si une activité a plusieurs noeud initiaux, ils activent chacun leurs flots de controle respectifs (fork implicite).

Solution :

- Création d'une place marquée si c'est l'activité de haut niveau, sinon elle sera marquée par l'activité de plus haut niveau.
- Une transition permet de fournir à chaque arc sortant un jeton.

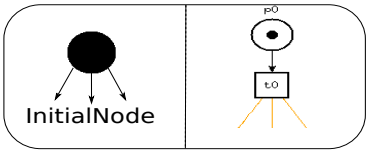

FIG. 3.8 – Mapping de Initialnode

Noeud final de l'activité (figure 3.9) :

Description issue de la spécification OMG : Un noeud d'activité final est un noeud final qui arrête tous les flux dans une activité.

Propriété issue de la spécification OMG :

– Il arrête toutes les procédures d'exécution dans l'activité, et détruit tous les jetons dans les noeuds en cours d'exécution, sauf dans les noeuds de sortie type "paramètre" de l'activité.

Solution :

– Création d'une transition pour chaque arc entrant, afin d'activer la place finale et donc démarquer l'ensemble de l'activité.

– Mettre en place une solution qui élimine l'ensemble des jetons présents dans l'activité en cours ne pose pas de problème technique particulier, il suffit de mettre en place des transitions puits à chaque place que l'on tire avant de sortir de l'activité en cours. Cependant adopter cette solution augmente la complexité du RdP équivalent sans pour autant apporter un réel intérêt au développement de ce projet.

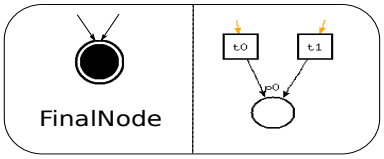

FIG. 3.9 – Mapping de ActivityFinalnode

Noeud de flot final (figure 3.10) :

Description issue de la spécification OMG : Un noeud de flot final est un noeud final qui met fin à un flux.

Propriété issue de la spécification OMG : Le Flot final détruit les jetons qui atteignent le noeud.

Solution :

– Une transition pour chaque arc entrant dans le *FlowFinalNode*.

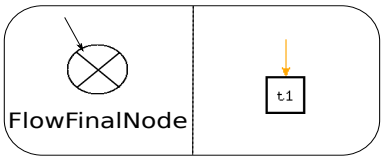

FIG. 3.10 – Mapping de FlowFinalNode

Noeud de jonction (figure 3.11) :

Description issue de la spécification OMG : Un noeud de jonction est un noeud de contrôle qui synchronise des flux multiples.

Propriété issue de la spécification OMG :

- Si tous les jetons disponibles en entrée sont des jetons de contrôle, alors un jeton de contrôle est fourni en sortie.
- Si certains des jetons présents en entrée sont des jetons de contrôle et d'autres sont des jetons de données, seuls les jetons de données sont fournis en sortie. Les jetons sont offerts dans le même ordre qu'ils sont arrivés.

Solution :

- Création d'une transition qui a les mêmes propriétés de synchronisation dans les RdPs.
- La gestion de l'ordre d'arrivé (buffer) ne sera pas modélisée par réseau de Petri.

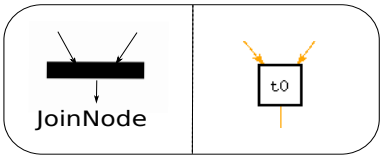

FIG. 3.11 – Mapping de JoinNode

Noeud de bifurcation (figure 3.12) :

Description issue de la spécification OMG : Un noeud de bifurcation est un noeud de contrôle qui sépare un flot en plusieurs flots simultanés.

Propriété issue de la spécification OMG :

- Les jetons entrant sont offerts à tous les arcs sortants du noeud.

Solution :

- Création d'une transition qui permet de redistribuer autant de jetons que d'arcs sortants de façon synchrone.

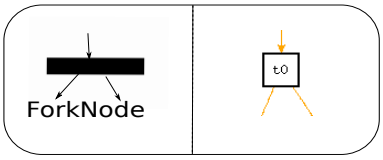

FIG. 3.12 – Mapping de ForkNode

Noeud de décision (figure 3.13) :

Description issue de la spécification OMG : Un noeud de décision est un noeud de contrôle qui choisit entre les flux sortants.

Propriété issue de la spécification OMG :

- Chaque jeton arrivé à un noeud de décision ne peut être redistribué qu'à une seule sortie.

Solution :

- Une transition (t0) attend le flot de contrôle et/ou le flot de donnée
- une place d'attente (p0) durant la phase de décision.
- Création des sorties (ici t1,t2) en fonction du nombre de sorties de décision.

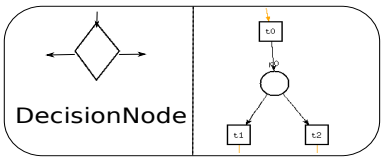

FIG. 3.13 – Mapping de DecisionNode

Noeud de fusion (figure 3.14) :

Description issue de la spécification OMG : Un noeud de fusion est un noeud de contrôle qui réunit des flots. Il n'est pas utilisé pour synchroniser des flots concurrents, mais pour en accepter un parmi plusieurs.

Propriété issue de la spécification OMG :

- Tous les jetons entrants sont distribués en sortie. Il n'y a pas de synchronisation des flux ou d'aggrégations de jetons.

Solution :

- Création d'une transition (ici t0,t1) pour chaque entrée du *Merge*
- Une place qui redistribue directement chaque jeton en sortie (t2)

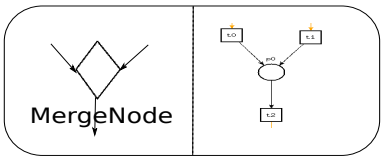

FIG. 3.14 – Mapping de MergeNode

Noeud d'action (figure 3.15) :

Description issue de la spécification OMG : Une action représente une étape unique dans une activité,une action est un élément nommé qui est l'unité fondamentale de la fonction exécutable. L'exécution d'une action représente une transformation ou un traitement dans le système modélisé, que ce soit un système informatique ou autre.

Propriété issue de la spécification OMG :

- Une exécution de l'action est créée lorsque toutes les conditions sur les flots d'objets et de contrôle ont été satisfaits (jonction implicite).
- A la fin d'exécution de l'action tous les arcs sortants reçoivent un jeton (fork implicite).

– Si plusieurs jetons de contrôle sont disponibles sur un *ControlFlow* entrant unique, ils sont tous consommés.

Solution :

– Création d'une transition d'entrée et d'une transition de sortie pour la gestion du *JoinNode* et du *ForkNode* implicite.
– La dernière propriété est déplacée sur les *ControlFlow* pour qu'il ne stocke pas les jetons mais qu'il fasse office de buffer en vehiculant comme seule information : présence ou non d'un jeton.

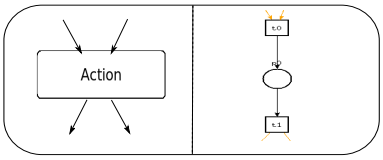

FIG. 3.15 – Mapping de ActionNode

Noeud d'envoi de signal (figure 3.16) :

Description issue de la spécification OMG : *SendSignalAction* est une action qui crée un signal une fois toutes ses entrées actives, et le transmet à l'objet cible

Propriété issue de la spécification OMG :

– Identique à une action, elle hérite directement d'*Action*

Solution :

– Ajout d'une place de communication asynchrone.

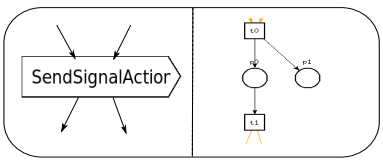

FIG. 3.16 – Mapping de SendSignalAction

Noeud de réception de signal (figure 3.17) :

Description issue de la spécification OMG : AcceptEventAction est une action qui attend l'arrivée d'un signal.

Propriété issue de la spécification OMG :

– Identique à une action, elle hérite directement d'*Action*
– Si une information est reçue mais non lue, l'arrivée d'une autre information viendra écraser la première, (limitation d'une des spécifications de l'OMG.)

– Important : Un noeud *AcceptEvent* communique au travers d'un *Port* et d'un *Trigger* avec le noeud *SendSignalAction*. Il faut s'assurer de leur présence afin d'établir le lien entre ces deux noeuds.

Solution :

– idem Action
– création d'un buffer en entrée de l'action reliée à la place de communication *SendSignal*.

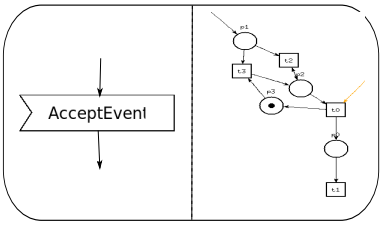

FIG. 3.17 – Mapping de AcceptEvent

Noeud broche d'entrée (figure 3.18) :

Description issue de la spécification OMG : Une broche d'entrée est une broche qui détient les valeurs d'entrée à consommer par une action.

Propriété issue de la spécification OMG :

– Une action ne pourra pas commencer l'exécution si une broche d'entrée a moins de valeurs que la plus faible multiplicité.
– La multiplicité supérieure détermine le nombre maximum de valeurs qui peut être consommé par une seule exécution de l'action.

Solution :

– Création d'un buffer qui permet la gestion du rafraichissement de la donnée. La présence d'une seule et unique donnée a été pris en compte à ce jour.
– A noter : Ce noeud peut être stéréotypé, selon 6 types. La transformation de deux d'entre eux apparait comme essentiel pour un fonctionnement correct des DAs.

Stéréotypes :

– *Optionnal* (figure 3.19), l'action ne commence pas sans présence de valeur en entrée.
– *Continuous* (figure 3.20), l'action ne peut commencer que si une valeur est présente en entrée, mais peut, durant son exécution, récupérer de nouvelles valeurs, contrairement à l'*InputPin* classique qui doit attendre une nouvelle exécution de l'action.

Noeud broche de sortie (figure 3.21) :

Description issue de la spécification OMG : Une broche de sortie est une broche qui détient les valeurs de sortie produites par une action.

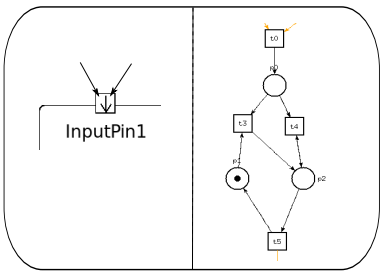

FIG. 3.18 – Mapping de InputPin

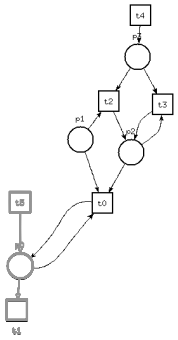

FIG. 3.19 – Mapping de InputPin stéréotypé «optionnal»

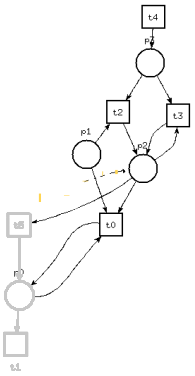

FIG. 3.20 – Mapping de InputPin stéréotypé «continuous»

Propriété issue de la spécification OMG :

- Pour chaque exécution, une action ne peut se terminer que lorsqu'il y a au moins, autant de valeurs sur ses broches de sortie que requis par la multiplicité inférieure de ces broches.
- Les valeurs sont effectivement délivrées une fois l'action terminée.
- Une action ne peut pas mettre plus de valeurs dans une broche de sortie en une seule exécution que la multiplicité supérieure de la broche.

Solution :

- Les solutions sont équivalentes à celle mise en place pour les broches d'entrées.

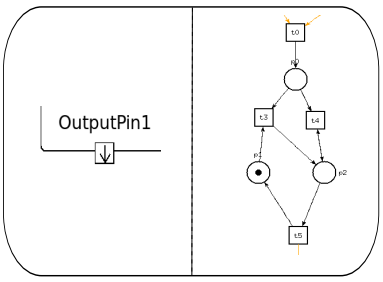

FIG. 3.21 – Mapping de OutputPin

Noeud de paramètre d'activités (figure 3.22) :

Description issue de la spécification OMG : Les noeuds *ActivityParameterNode* sont des noeuds objet de début et de fin de flot qui fournissent les entrées et les sorties à une activité, à travers les paramètres d'activité.

Propriété issue de la spécification OMG :

- Au cours de l'exécution de l'activité, les jetons peuvent s'écouler dans les noeuds paramètres d'activité au sein de l'activité.
- Lorsque l'exécution de l'activité est terminée, les valeurs de sortie détenues par ces *ActivityParemeterNode* de retour sont délivrées.

Solution :

- Création d'une communication par place entre le *InputPin* et l'*Activity* de niveau inférieur, toutes les propriétés étant redondantes avec les noeuds *Pin*, elles sont reportées sur le *Pin* correspondant.

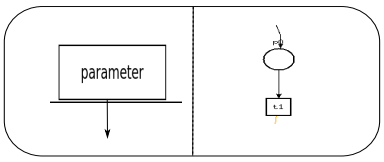

FIG. 3.22 – Mapping de ActivityParameterNode

Arc de control de flot(figure 3.23) :

Description issue de la spécification OMG : Un arc de contrôle de flux est un arc qui démarre un noeud d'activité après que la précédente se soit terminée.

Propriété issue de la spécification OMG :

- Un contrôle de flux est une arc qui ne laisse passer que les jetons de contrôle.
- Les jetons offerts par le noeud source sont tous offerts au noeud cible. N'importe quel nombre de jetons peut passer l'arc, par groupes, ou individuellement.

Solution :

– Un simple arc de RdP pourrait semblait suffisant, mais certaines propriétés ont été déportées sur l'arc lors de notre transformation (dans un DA une action consomme tous les jetons présents sur l'arc), nous modélisons donc le *ControlFlow* par un buffer pour éviter le stockage de jetons et ne garder que l'information de présence de jetons.

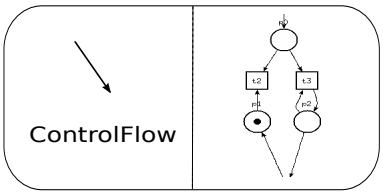

FIG. 3.23 – Mapping de ControlFlow

Arc de Control de donnée (figure 3.24) :

Description issue de la spécification OMG : Un flux d'objets est un arc qui fait circuler les données.

Propriété issue de la spécification OMG :

– Idem ControlFlow

Solution :

– Un *ObjectFlow* est toujours précédé et suivi d'un *Pin* d'entré ou de sortie, la solution adoptée n'est donc pas la même que pour le *ControlFlow*. Un arc seul n'aurait pas permis une construction générique. La solution ARC-PLACE-ARC permet de relier les *Pin* entre eux.

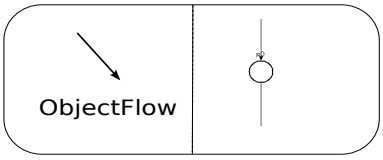

FIG. 3.24 – Mapping de ObjectFlow

La liste des mappings défini ici permet d'élaborer des diagrammes d'activités relativement complexes. Cependant le formalisme des diagrammes d'activités est riche, une transformation totale des DAs vers les RdPs demanderait des travaux plus conséquents sans toutefois poser de problème technique majeur. Il a également été traité les zones interruptibles qui ne sont plus des noeuds au sens premier du terme mais une propriété distribuable à un ensemble de noeud. Ce mapping a été réalisé et validé par simulation mais n'a pu faire l'objet d'une implémentation en règle ATL par manque de temps. Un récapitulatif des mappings est disponible en annexe 3.

CHAPITRE 4

La mise en oeuvre

Ce chapitre parle de l'ensemble du travail effectué après la méta-modélisation, avec les règles de transformations ATL. Il illustre également les possibilités offertes par ces travaux au travers de deux exemples : le premier montrant la possibilité de gestion de situation complexe nommé Butterfly, un second plus proche du monde de l'industrie, l'injecteur.

4.1 La Transformation : ActivityDiagram2PetriNet

Cette partie parle de l'ensemble du travail de conception réalisé pendant cette étude : les transformations en langage ATL. La vérification formelle des règles de transformation grâce au "model-checker" de TINA toolbox pour finir par la mise en place de la simulation.

4.1.1 Transformation via ATL

Suite à la création du standard MDA de l'OMG, de nombreux outils basés sur cette approche ont vu le jour, une équipe du laboratoire de l'INRIA de Nantes a développé le langage ATL (Atlas Transformation Language) [7] servant à la transformation de modèle. Ce langage est proche du standard QVT(Querry,View,Transformation) [5], proposé par l'OMG. Cette ressemblance est historique puisque ATL est la première tentative d'implémentation de QVT. ATL est aujourd'hui un des langages de transformation de modèle les plus matures, c'est donc naturellement que notre choix s'est porté sur ce langage. De plus la communauté qui développe autour de ce langage est importante et met à disposition un "zoo" (bibliothèque) de transformation [17].

Etablir une règle de base en ATL :

Un programme ATL est composé de règles qui spécifient comment les éléments du modèle cible doivent être créés en fonction des éléments présents dans le modèle source. Ces règles sont toujours

établies suivant le schéma suivant :

```
rule Nom_de_la_regle
    from
       i: Nom_MM_Source!Nom_Méta-Classe_Source

    to
        o: Nom_MM_Cible!Méta-Classe_Cible
              Attribut1 <- i.AttributA,
              Attribut2 <- i.AttributB+i.AttributA,
```

- *rule*, *from* et *to* sont les instructions du langage.
- i (resp. o) est le nom de la variable qui dans le corps de la règle représente une instance de la méta-classe source (resp. cible) dans la modèle source (resp. cible).
- Attribut1,2 (resp. A,B) sont les attributs de la méta-classe cible (resp. source) du méta-modèle cible (resp. source).

Le point d'exclamation permet de spécifier à quel méta-modèle appartient une méta-classe.

Un traduction en langage courant donnerait :
La règle *Nom de la règle* crée pour chaque instance *i* identifiée dans le modèle source une instance *o* de la méta-classe cible dans le modèle cible, en donnant à *Attribut1* la valeur de *AttributA* et à *Attribut2* la valeur de la somme *AttributA* plus *AttributB*.

ATL permet de factoriser les rêgles avec l'utilisation de *helper*, que l'on peut assimiler à des fonctions.

Les transformations :

Ce projet basé sur la transformation des diagrammes d'activités vers les réseaux de Petri n'a pas engendré qu'une seule transformation mais plusieurs, comme illustré dans la figure 4.1.

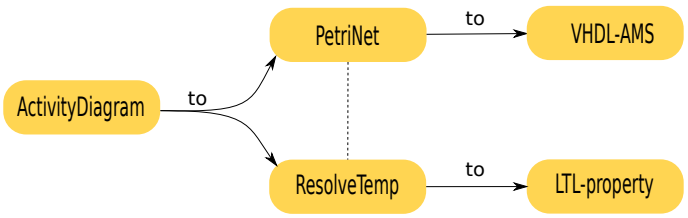

FIG. 4.1 – Les Transformations

En effet la transformation ActivityDiagram2PetriNet implémente la mapping des concepts. Le réseau de Petri généré sous la forme de modèle XML n'est pas directement exploitable par TINA qui à une description textuelle des RdPs qui lui est propre. Enfin SystemVision nécessite une transformation PetriNet2VHDL-AMS. Il faut adapter le modèle aux outils. C'est pourquoi apparaissent les transformations :

PetriNet2TINA Cette transformation de type "model2text" permet d'adapter le modèle réseaux de Petri obtenue à l'outil TINA.

PetriNet2VHDL-AMS Cette transformation à déjà fait l'objet d'un travail [15] est n'est ici qu'un outil. Elle prend en entrée la description textuelle des RdPs de TINA.

ActivityDiagram2ResolveTemp Cette transformation s'effectue en même temps que la transformation ActivityDiagram2PetriNet, elle permet de récupérer la structure de chaque bloc de base 3.7 en donnant le nom :

- des arcs entrants
- des arcs sortants
- de la place symbolisant l'artefact en cours d'exécution
- la transition d'entrée
- la transition de sortie
- exceptionnellement des informations supplémentaires (place de non-exécution,...)

A noter que cette transformation est liée au modèle réseaux de Petri est n'est pas exploitable sans celui-ci. Le méta-modèle ResolveTemp est visible en Annexe B.

ResolveTemp2LTL Cette transformation de type "model2text" permet la création automatique d'un fichier texte avec l'ensemble des propriétés LTL permettant de vérifier de façon formelle la transformation.

Implémentation d'une règle ATL :

L'ensemble de ces transformations établissent des règles comme cela a été vu dans le paragraphe "Etablir une règle de base en ATL". Regardons maintenant l'implémentation de la règle de transformation du noeud *Action* figure 3.15. Le principe est de créer un transition, un arc, une place , un arc et une transition à chaque fois qu'un noeud de type action est reconnu.

```
1 rule action
2      from a:MMAD!Action
3
4      to b : MMH!Transition(
5             Name<-'t1_Action_'+a.name+'_'+a.activity.name,
6             OutputLink<-c,
7            InputLink<-a.incoming->collect(inp|thisModule.resolveTemp(inp,'a32')),
8            Diagram<-a.activity),
9
10        c: MMH!ArcClassic
11           (Name<-'a1_Action_'+a.name+'_'+a.activity.name,
12            Target<-d,
```

```
13            Diagram<-a.activity),
14
15          d: MMH!Place(
16           Name<-'P_Action_'+a.name+'_'+a.activity.name,
17           OutputLink<-e,
18           Diagram<-a.activity),
19
20         e: MMH!ArcClassic
21          (Name<-'a2_Action_'+a.name+'_'+a.activity.name,
22          Target<-f,
23          Diagram<-a.activity),
24
25          f: MMH!Transition (
26           Name<-'t2_Action_'+a.name+'_'+a.activity.name,
27           OutputLink<-a.outgoing,
28           Diagram<-a.activity,
29           OutputLink<-a.output)
```

- Ligne 1 :Création de la règle "action".
- Ligne 2 : "a" sera définit comme méta-classe "Action" du méta-modèle MMDA dans cette règle et viendra du modèle source (from).
- Lignes 4,10,15,20 : Définition des méta-classes cibles. Pour chaque Méta-classe Action on crée :
  - Ligne 5 : Une instance de la méta-classe Transition du méta-modèle MMH dont l'attribut *Name* prend : $t1\_Action + nom\ de\ l'\ action + nom\ de\ l'activité$..
  - Ligne 11 : Une instance de la méta-classe ArcClassic du méta-modèle MMH dont l'attribut *Name* prend : $A1\_Action + nom\ de\ l'\ action + nom\ de\ l'activité$.
  - Ligne 16 : Une instance de la méta-classe Place du méta-modèle MMH dont l'attribut *Name* prend : $P\_Action + nom\ de\ l'\ action + nom\ de\ l'activité$.
  - Ligne 26 : Une instance de la méta-classe ArcClassic du méta-modèle dont l'attribut *Name* prend : $A2\_Action + nom\ de\ l'\ action + nom\ de\ l'activité$.
- Lignes 6, 12, 17, 22 : Mise en place des liens d'associations entre les méta-classes. Pour une instance d'action sera créé un lien entre b et c (entre la transition et l'arc), c et d (entre l'arc et la place), d et e (entre la place et l'arc), e et f (entre l'arc et la transition). Ce sont les connections internes du bloc RdP.
- Ligne 27 : OutputLink reçoit l'ensemble des sorties d'une des instances de la méta-classe Action (a.outgoing). C'est là que s'établit la connexion entre les blocs de base du RdP.

L'application de cette règle de transformation est une traduction au niveau méta-modèle des idées développées durant le mapping. En développant quelques règles de base il est possible d'effectuer une transformation simple comme l'illustre la figure 4.2.

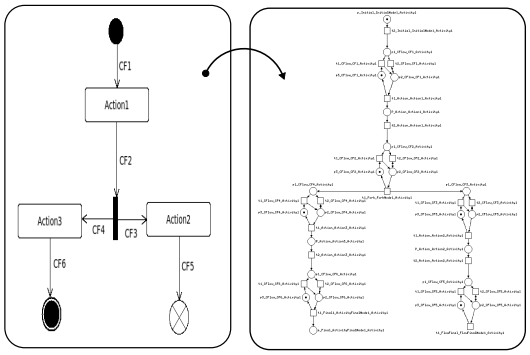

FIG. 4.2 – Transformation Simple DA2RdP

On retrouve les blocs élémentaires décrits dans le chapitre 3. Le parallélisme engendré par le noeud *Fork* apparait clairement sur la figure de droite avec la transition distribuant un jeton sur les deux blocs modélisant CF3 et CF4.

La validation d'une transformation composée d'une dizaine de noeuds RdP est triviale, cependant la complexité des systèmes à modéliser aujourd'hui amène des RdP de grande taille (des centaines de places/arcs/transitions). On peut voir le résultat de la transformation d'un diagramme d'activité complexe (plusieurs dizaine de noeuds de DA) figure 4.3. C'est pourquoi la vérification formelle et automatique du réseau engendré est indispensable.

4.1.2 Vérification formelle de la transformation

Il existe plusieurs méthodes d'analyse :

- La simulation.
- Les tests.
- La vérification formelle.

Cette section va s'intéresser à la vérification formelle basée sur des modèles formels. Elle consiste à explorer tout les états du système pour en tirer des propriétés d'atteignabilité, vivacité et de sécurité. Nous avons choisis d'utiliser le model-checking qui est adapté à la vérification des RdPs [18].

Pour effectuer une vérification formelle par model-checking, il faut :

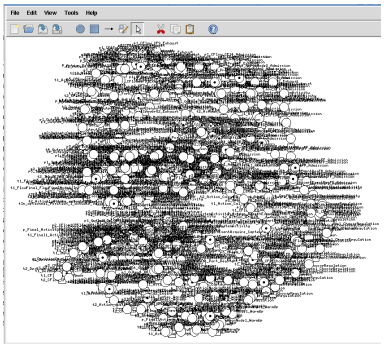

FIG. 4.3 – Résultat de transformation d'un diagramme d'activité complexe (exemple de l'injecteur)

– Elaborer un modèle qui décrit le comportement du système, ici le réseau de Petri résultant de la transformation.
– Exprimer les propriétés dans un formalisme comme la logique temporelle, ici LTL.

Le Model Checking comporte une limitation forte, les modèles comportementaux doivent être finis. En pratique il est souvent possible d'obtenir facilement un RdP fini.
Notre modèle RdP est prêt pour la vérification formelle, il a été vu précédemment que d'autres transformations de modèle avait été nécessaires. Une d'elle sert à automatiser la génération des propriétés LTL (ResolveTemp2LTL).

La logique temporelle et le langage LTL :

Les logiques temporelles sont utilisées pour décrire des propriétés, c'est donc un langage de description. Les propriétés décrites en logique temporelle sont vérifiées sur un modèle. Le temps n'est pas exprimé de manière explicite, mais on cherche à voir si une propriété sera toujours vérifiée ou ne le sera jamais. Les propriétés de base sont : jamais, éventuellement, toujours, un jour, demain, jusqu'à. A partir de ces propriétés de base on cherche a établir des propriétés générales. Ceci afin de vérifier si la sémantique des diagrammes d'activité est conservée dans le réseau de Petri équivalent. De plus, il est possible d'établir des propriétés propres au diagramme établi par l'utilisateur qu'il rentre dans le model-checker.

Les propriétés LTL :

Pour vérifier de façon automatique, les propriétés sont décrites pour chaque entité de base du méta-modèle ResolveTemp. Pour rappel, le méta-modèle ResolveTemp décrit les blocs de base RdP sous la forme Transition-Place-Transition, le modèle résultant définit le nom de la transition de départ, le nom de la place symbolisant l'exécution de l'activité, et la transition de sortie comme indiqué sur la figure 4.4

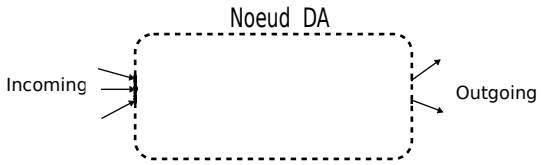

FIG. 4.4 – Description des blocs de bases

La démarche d'élaboration est basée sur la logique A ⇒ B et B ⇒ C alors A ⇒ C. En d'autres termes si la propriété définie pour l'entrée du bloc de base (liens entre les blocs) est vérifiée, si la propriété dans le bloc et celle à la sortie du bloc sont vérifiées, alors l'ensemble des blocs est vérifié et donc l'ensemble du réseau.

Elaboration des propriétés du bloc Action :

$$1\ [](startAction- >< > stopAction)$$

$$2\ [](All\_incoming\_run- >< > startAction)$$

$$3\ [](stopAction- >< > All\_Outgoing\_run)$$

- La propriété 1 peut se traduire : Si le franchissement de la transition StartAction implique ($- >$) un jour ($<>$) le franchissement de la transition StopAction est toujours vrai ($[]$) alors vrai sinon faux . Soit : une action qui commence, ce termine toujours.
- La propriété 2 peut se traduire : Si toutes les entrées des actions actives en même temps impliquent ($- >$) un jour ($<>$) le franchissement de la transition StartAction est toujours vrai ($[]$) alors vrai sinon faux . Soit, une action ne commence que si toutes ces entrées sont actives ("join" implicite)
- La propriété 3 peut se traduire : Si le franchissement de la transition StopAction implique ($- >$) un jour ($<>$) l'exécution de toutes les sorties est toujours vrai ($[]$) alors vrai sinon faux . Soit, une action qui se termine active toute ces sortie ("fork" implicite)

Ces propriétés sont ensuite traduites dans le langage LTL acceptées par TINA grâce à la transformation de type *model2text* ResolvTemp2LTL.

4.1.3 Simulation

Le model-checking ne s'applique pas aux RdP-PTDs. Nous avons donc recours à la simulation dès lors que du continu et de la gestion de données sont mis en jeu dans les modèles. L'inconvénient majeur de la simulation est qu'elle ne garantie pas que tous les cas possibles aient été essayés. Pour effectuer cette simulation nous utilisons la transformation PetriNet2VHDL-AMS. Cette transformation doit être modifiée à la main car les équations différentielles ne sont pas prises en compte dans la première transformation ActivityDiagram2PetriNet. La prochaine partie traite deux exemples qui utilisent la simulation, par faute de place la démarche de mise en place de la simulation ne sera pas traité dans ce rapport.

4.2 Applications

Cette seconde partie du chapitre permet de mêttre en application l'ensemble de la chaine de transformation de l'élaboration du diagramme d'activité à la simulation.

4.2.1 Butterfly

Cet exemple minimaliste montre comment résoudre une équation différentielle du second ordre avec deux entités résolvant chacune une équation différentielle du premier ordre échangeant leur résultat à chaque cycle de résolution.

Soient les équations différentielles :

$$\dot{q_1} + 200.10^3 q_1 + 30.10^6 q_2 = 5.10^3 \quad (1)$$

$$\dot{q_2} - q_1 = 0 \quad (2)$$

L'équation (1) (resp.(2)) est associée à l'action I1 (resp. I2) du diagramme d'activité de la figure 4.5. Au lancement de l'activité les deux actions sont prêtes à être exécutées, I1 est initialisé et commence la

résolution de l'équation qui lui est associée. A chaque cycle de la résolution est fournie une donnée par l'intermédiaire de OF1 à I2. I2 peut alors commencer sa résolution et effectuer un cycle de résolution de l'équation avant de donner une nouvelle valeur q2 à I1 par l'intermédiaire de OF2. Ce cycle permet, grâce à deux équations différentielles du premier ordre, d'obtenir une équation différentielle du second ordre. Entre chaque cycle, la résolution est stoppée et garde la valeur en mémoire, c'est un choix de conception, il est tout à fait envisageable de résoudre parallèlement les deux équations et de mettre à jour les variables q1 et q2 dans les résolutions lorsque celles-ci sont disponibles. La résolution ce fait ici de façon alternative.

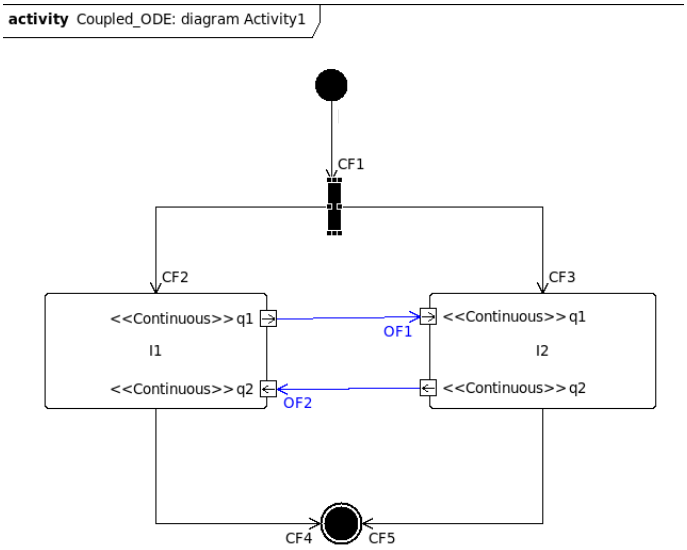

FIG. 4.5 – Diagramme d'activité de l'exemple butterfly

Après transformation vers les réseaux de Petri, et vers l'outil TINA nous obtenons automatiquement le réseau de Petri figure 4.6.

- La tête représente le noeud initial, CF1 et le *ForkNode*.
- Les bouts des ailes représentent les pins d'entrées et de sorties stéréotypées «continuous»,
- Les ailes représentent CF2, CF3 et les actions I1 et I2.
- Le coeur du papillons représente l'échange de donnée via OF1 et OF2.
- La queue représente CF5,CF5 et le noeud final.

Le réseau obtenu est montré ici de façon graphique, une fois adapté à une échelle industrielle cet aspect des réseaux de Petri n'a plus de sens, c'est ce qui nous amène à effectuer une vérification formelle automatisée grâce à la transformation ResolveTemp2LTL. Cette étape franchie, il ne reste plus qu'a implémenter dans SystemVision le réseaux de Petri obtenu. Ce codage est en partie automatisé grâce à la transformation PetriNet2VHDL-AMS. En l'état actuel du projet il faut compléter manuellement les équations différentielles même si une grande partie de la transformation RdP-prédicat-transition-différentielle vers VHDL-AMS est opérationnelle, l'apparition de quelques problèmes de fonctionnement amène à coder les équations manuellement.

Nous obtenons finalement les résultats de simulation de la figure 4.7 . L'effet d'escalier vient de la

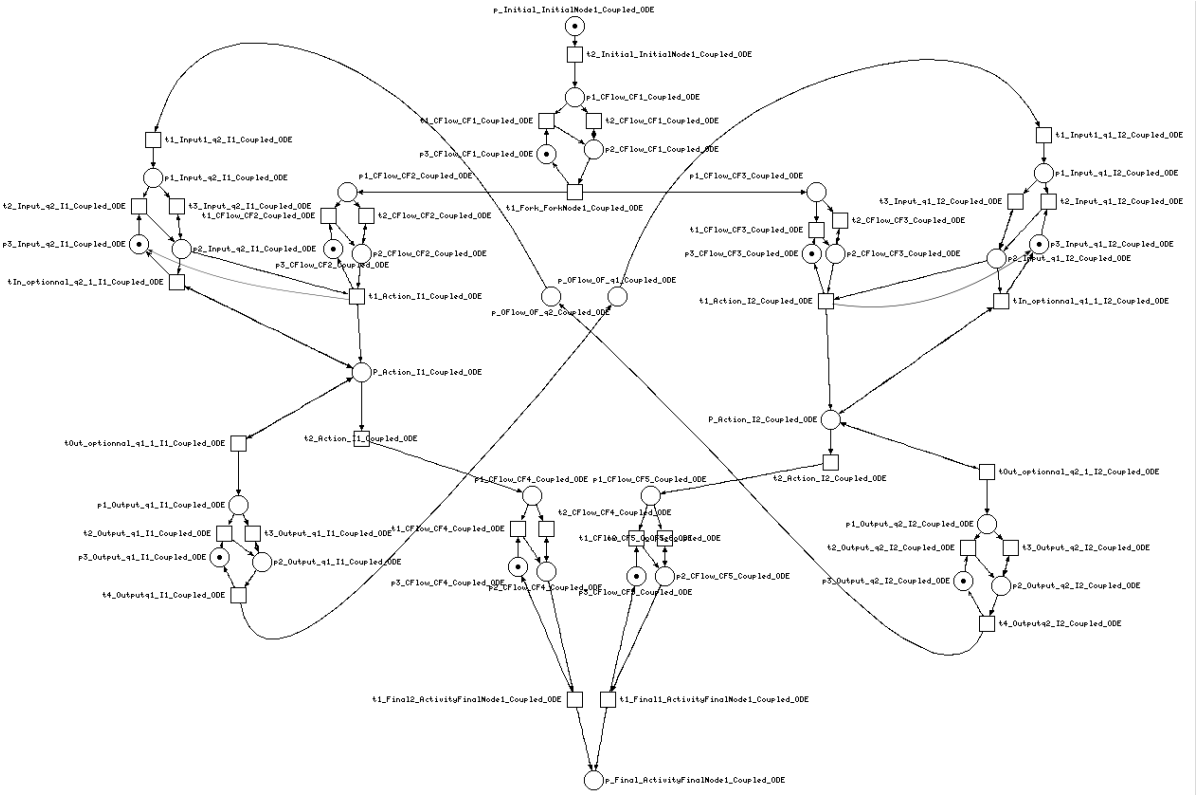

FIG. 4.6 – Réseaux de Petri de l'exemple butterfly

structure de l'exemple, les nouvelles valeurs arrivent de façon synchrone aux activités. On observe les oscillations typiques d'une équation du second ordre avec son dépassement avant stabilisation.

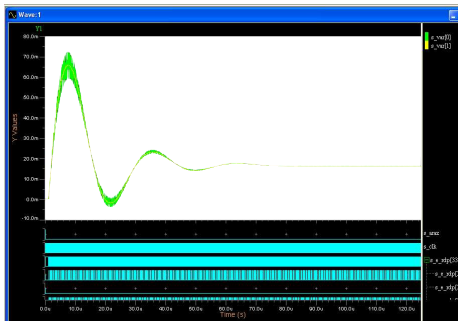

FIG. 4.7 – Simulation d'une équation du second ordre sous SystemVision

Cet exemple montre qu'il est possible à partir d'un diagramme d'activité, d'obtenir automatiquement une simulation exploitable grâce au formalisme des RdPs et à VHDL-AMS même sur des systèmes complexes avec un échange de donnée continue.

4.2.2 Injecteur

L'exemple précédent a permi de montrer de façon simple la gestion de l'interaction entre deux entités. L'application présentée dans cette seconde partie se veut plus proche des réalités industrielles et a été développée dans avec une approche identique à celle que l'on peut trouver dans les bureaux

d'études. Sa structure plus complexe fait apparaitre la notion de hiérarchie et d'échange de données entre niveaux hiérarchiques.

Le cahier des charges

Le système est chargé de commander le dosage du mélange gazeux (air + essence + gaz d'échappement) à fournir pour un moteur à explosion à 4 temps (voir figure 4.8). Le système temps réel engendre deux types d'informations :

- Une première sortie discrète calibrée en temps (instant d'activation et largeur) qui commande le temps d'injection (variable suivant le mode de fonctionnement du moteur).
- Une deuxième sortie de type continu commande la position de la valve de remise en circulation d'une partie des gaz d'échappement (pour limiter la pollution).

Ces deux sorties sont engendrées en permanence à partir de deux sources d'informations :

- Un ensemble de mesures effectuées en temps réel à l'aide de différents capteurs embarqués disséminés dans les différents organes du moteur.
- Un ensemble de paramètres propres à chaque type de moteur et de voiture, déterminés en laboratoire et en essais. Ils se présentent sous la forme de tables et sont fournis ainsi pour satisfaire un but économique (réduire le nombre et la complexité des capteurs) et pour diminuer la complexité technique des calculs en temps réel.

Le système de commande d'injection commande trois facteurs essentiels :

- La quantité du mélange gazeux air-essence envoyée par le carburateur via le collecteur d'admission.
- La richesse du mélange gazeux déterminée par le rapport quantité d'essence/quantité d'air.
- La quantité des gaz d'échappement remis en circulation dans le collecteur d'admission.

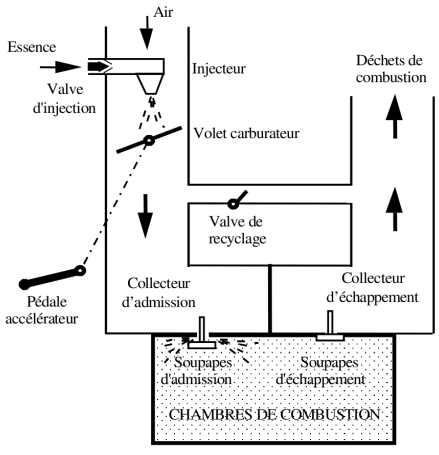

FIG. 4.8 – Schéma de principe d'un moteur à injection

Le comportement du moteur sera séparé en trois phases :

- Le démarrage du moteur
- La montée en température du moteur
- Le fonctionnement à température normale qui ce divise lui même en trois phases

- le moteur est conduit à régime constant.
- le moteur est sollicité en accélération.
- le moteur est sollicité en décélération.

De la conception à la simulation

Selon chacune de ces phases définies par les capteurs, le moteur recevra une quantité d'essence pendant un temps défini. Une étude approfondie du système permet d'obtenir le diagramme de contexte, le diagramme des cas d'utilisation ainsi que les diagrammes de séquences. On peut alors modéliser l'échange de donnée en figure 4.9 avec le diagramme d'activité complet du système de commande d'injection.

La figure 4.10 permet d'avoir une vue d'ensemble du projet et du cheminement chronologique à suivre. En partant des parties prenantes (stakeholders), une phase de conception amène à créer le diagramme d'activité. Un fois le modèle créé, la transformation diagramme d'activité vers réseaux de Petri est automatique. L'utilisateur possède alors deux fichier xmi, un modèle arboressent du réseau de Petri et un modèle "resolvtemp" définissant les instances des blocs de base. Ces deux modèles sont alors transformés pour obtenir un premier fichier contenant les propriétés LTL et un second contenant la netlist (la description textuelle) du réseau de Petri orienté TINA. L'utilisateur doit alors entrer ces deux fichiers dans l'outil TINA afin de procéder à la vérification formelle de la transformation (outil : selt). La netlist TINA peut alors être transformée en un ensemble de fichier VHDL. L'utilisateur n'a alors plus qu'a créer le testbench (scénario de simulation) afin d'observer et de valider le comportement analogique du système étudié.

L'analyse en détail

Afin de montrer plus en détail les possibilités offertes par cette transformation, observons une partie du diagramme d'activité et les résultats de simulation associés.

Pour ce faire nous allons étudier la partie Admission du diagramme d'activité (figure 4.11) et les phases de calcul associées.

Les capteurs envoient des données à la partie moteur du diagramme d'activité qui choisit en fonction du nombre de tours par minute du moteur et de la température du moteur un des trois modes de régulation :

- Régulation de démarrage $(cal\_AFR\_Start)(AFR : Air/Fuelrate)$
- Régulation boucle ouverte $(cal\_AFR\_ff)(ff : feedforward)$
- Régulation boucle fermée $(cal\_AFR\_fb)(fb : feedback)$

Notons le noeud final qui n'est pas relié au reste du graphe, ça présence n'est pas obligatoire. Le scénario choisi est tel que dans un premier temps le moteur est en phase de démarrage (phase 1), puis le nombre de tour par minute est inférieur à 1000 donc régulation boucle ouverte (phase 2) avant d'atteindre un régime moteur suffisant pour arriver en boucle fermée (phase 3).
Sur le diagramme d'activité cela ce traduit par :

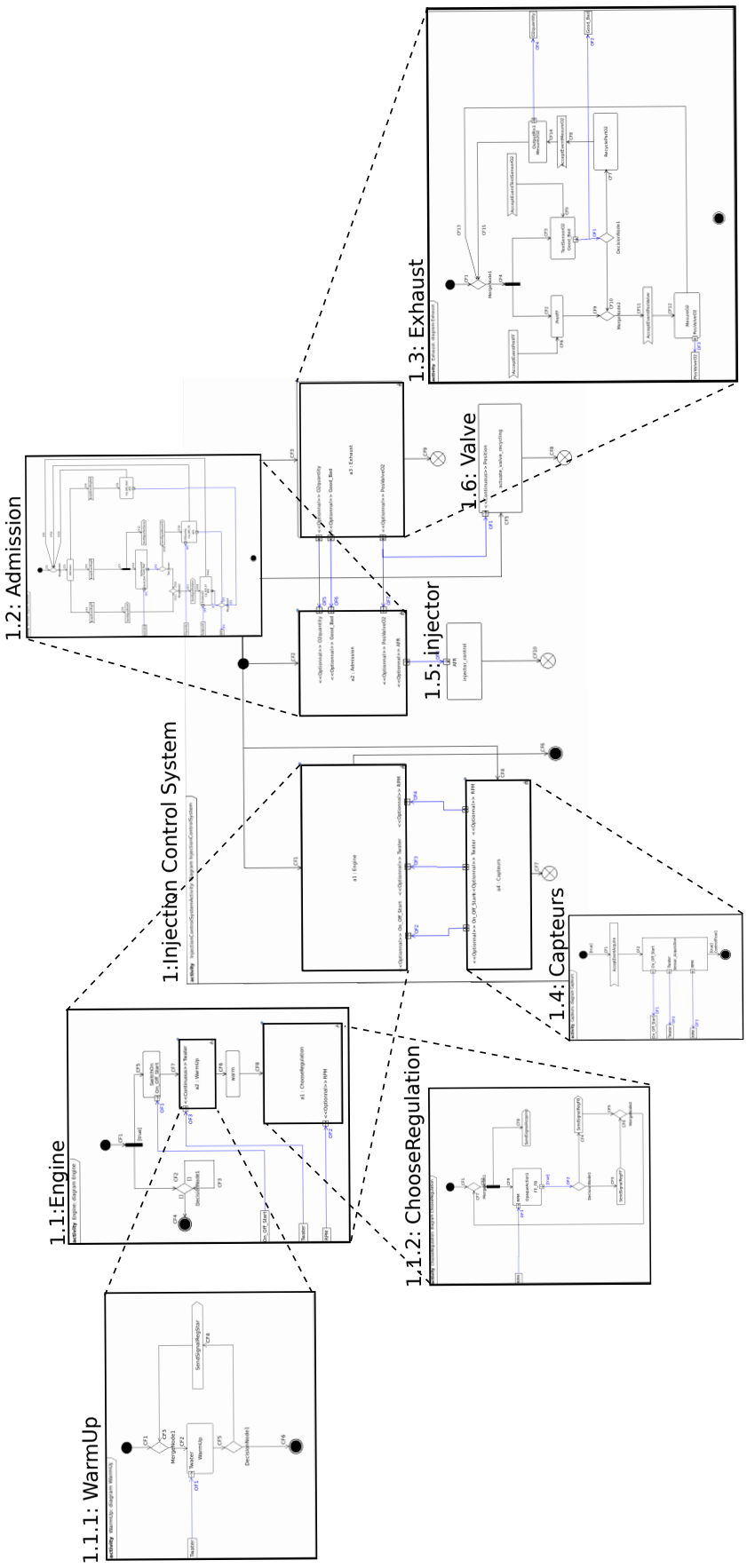

Fig. 4.9 – Diagramme d'activité du système de commande d'injection

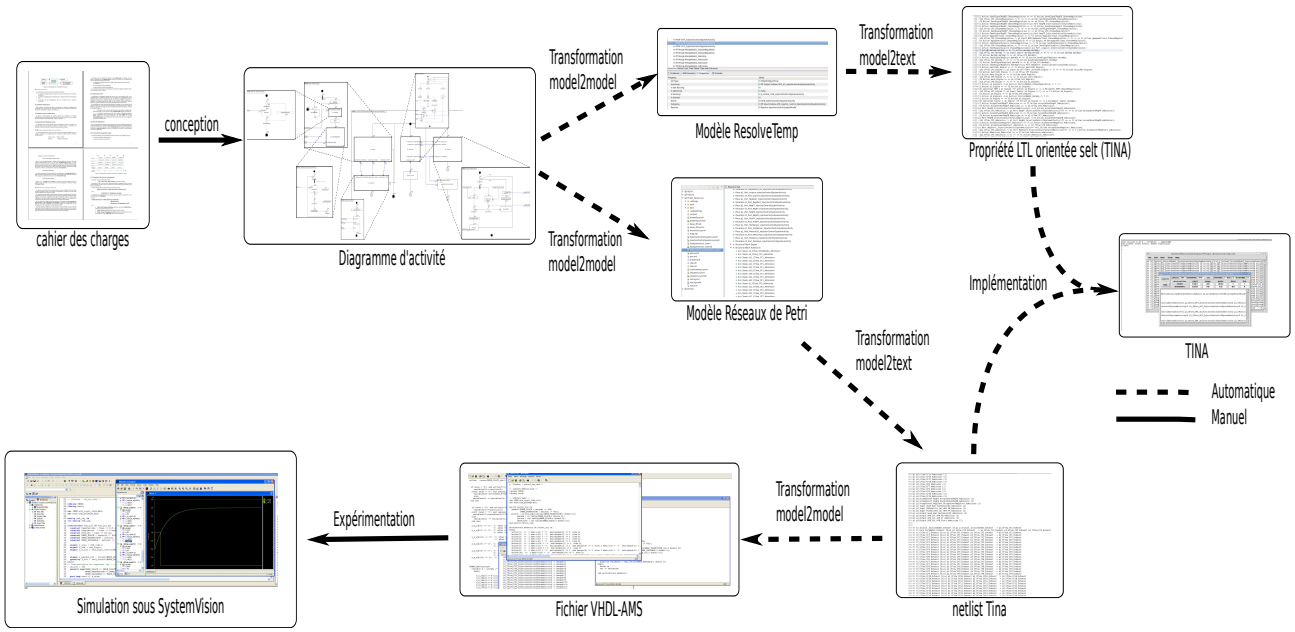

FIG. 4.10 – Du diagramme d'activité conforme OMG à la simulation

- Phase 1 : un envoi de signal RegStar de l'activité moteur, reçu par AcceptEventRegStart qui lance le calcul Régulation de démarrage (cal_AFR_Start)
- Phase 2 : un envoi de signal RegFF de l'activité moteur, reçu par AcceptEventRegFF qui lance le calcul Régulation boucle fermée (cal_AFR_ff)
- Phase 3 : un envoi de signal RegFB de l'activité moteur, reçu par AcceptEventRegFB qui lance le calcul Régulation boucle ouverte (cal_AFR_fb)

On retrouve ces trois phases dans la simulation (figure 4.12) avec les trois types de régulation et donc des ratios air/essence différents en fonction de la vitesse (en tourr/min) du moteur. Ces taux sont calculés grâce à des équations différentielles dont la résolution débute au lancement des activités cal_AFR. Dans les réseaux de Petri cela ce traduit par l'initialisation des variables mises en jeu au franchissement de la transition grâce à la fonction de jonction, à la résolution de l'équation durant la période où le jeton est dans la place, puis à la fin de la résolution lorsque la fonction de sensibilisation est activée et donc la transition de sortie du bloc Activité est franchie.

Cet exemple a montré qu'il était possible de transformer les diagrammes d'activités en un langage formel comme les réseaux de Petri et de les simuler. La partie combinatoire des réseaux de Petri est transparente pour l'utilisateur ,il n'a donc pas à superviser la gestion des transitions du RdP. Le testbench est donc une façon de contrôler le déroulement de la simulation, mais peut rester transparent si aucun débugage n'est souhaité. Cet exemple a amené à la création de 459 places et 442 transitions, pourtant cette complexité est restée totalement transparente à l'utilisateur, autant dans la phase de vérification que de simulation.

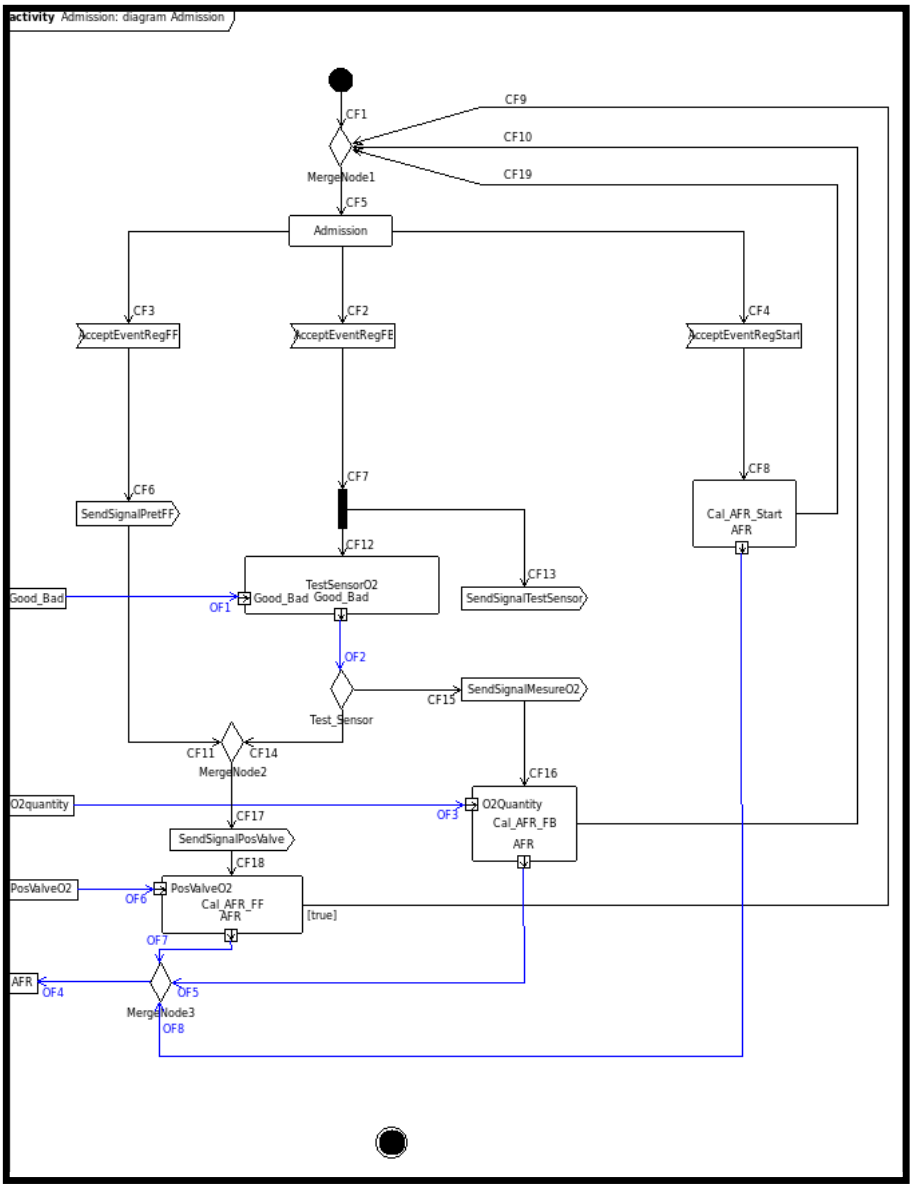

Fig. 4.11 – Activité Admission du diagramme d'activité calcul d'injection

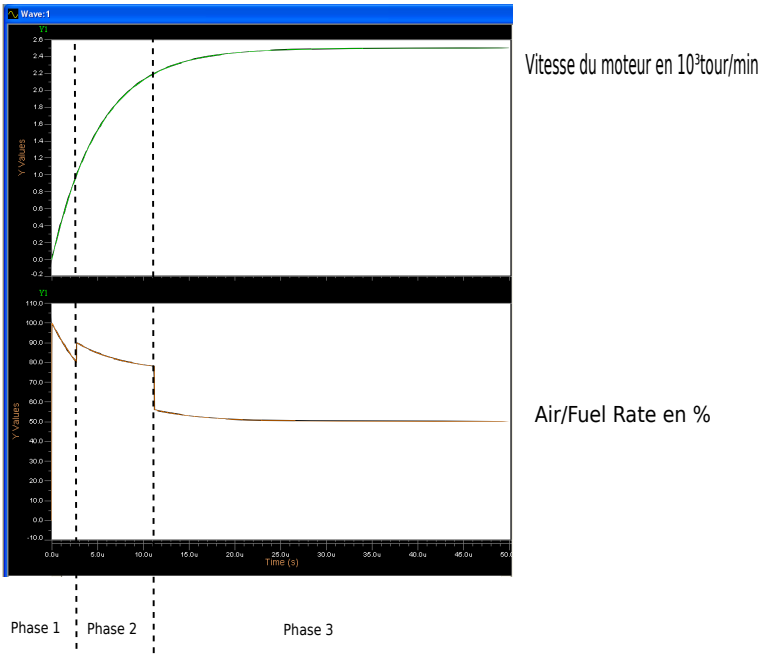

FIG. 4.12 – Simulation du calcul du taux d'injection Air/essence

CHAPITRE 5

Conclusion

L'objectif de ces travaux a été de proposer une transformation des diagrammes d'activités vers les réseaux de Petri dans un cadres IDM en respectant les spécifications définise par l'OMG. Il est apparu difficile de respecter l'ensemble des spécifications, le manque d'outils conformes aux spécifications, ou leur manque de maturité est un obstacle important à l'élaboration d'une solution en adéquation avec l'OMG. Notre approche s'est voulu la plus généraliste possible afin de transformer en réseaux de Petri la majorité des diagrammes d'activités, ce qui amène parfois à des solutions lourdes mais transparentes à l'utilisateur. De nombreux raccourcis (simplification) dans l'élaboration des concepts de la transformation on était ignoré pour coller au plus proche des spécifications de l'OMG. L'utilisation du principe du mapping des concepts permet un gain de temps important lors de l'établissement des règles. Un concept de méta-modèle des règles de transformation remplacant le mapping a été développé dans d'autres travaux mais n'a pas su trouver sa place et son utilité dans ces travaux

Le langage utilisé pour implémenter les règles de transformation (ATL) s'est avéré être un langage rapidement lourd et complexe. L'élaboration de règles de base est triviale si les deux méta-modèles sont proche (une méta-classe source correspond à une ou deux méta-classe cible) mais devient lourd et complexe pour élaborer des transformations complexes. ATL est apparu avec nos connaissances actuelles du langage peu efficace vis à vis des liens d'héritages entre méta-classe et donc de l'héritage de règles. De la même façon le choix d'application d'une règle en fonction de la valeur d'un argument du modèle source peut vite devenir un casse tête pour le développeur des règles, alors que l'ajout de quelques instructions pourrait permettre d'éviter de trop nombreuses lignes de code. La dernière difficulté observée dans l'utilisation de ce langage repose sur la quantité impressionnante de possibilité pour obtenir le même résultat, ce qui peut s'avérer déroutant pour un débutant en ATL. Un "zoo" de transformation permet d'illustrer l'utilisation d'ATL et de réutiliser ces transformations. La majorité des exemples traités étant relativement simples il est difficile de se lancer dans des transformations

complexes.

Ce projet ne devait à ses débuts que transformer les diagrammes d'activités en réseaux de Petri, au fil de l'étude il a semblé interessant de se pencher sur la vérification formelle de la transformation, de nouvelles transformations sont venu compléter ces travaux afin d'adapter nos modèles à l'outil TINA et son "model-checker" mais aussi à l'outils SystemVision utilisant VHDL-AMS. Cette vérification formelle nous permet d'affirmer que le comportement de notre réseau de Petri et celui des diagrammes d'activités sont identiques. Il est tout a fait possible d'envisager que l'utilisateur de cette transformation élabore ces propre propriétés LTL qu'il soumet au model-checker afin de voir si les propriétés qu'il attend sont présentes dans le diagramme d'activité qu'il a élaboré. La simulation elle, permet d'avoir une vision des signaux véhiculés dans le diagramme d'activité. Le réseau de Petri est animé de façon autonome (si le modèle utilisé se veut être autonome, e.g. l'appui sur un bouton devra toujours être géré par l'utilisateur) la complexité du réseau de Petri décrit en VHDL est alors transparente pour l'utilisateur.

5.0.3 Travaux Futurs

Cette étude est fonctionnelle et permet d'élaborer la transformation d'une grande partie des diagrammes d'activités mais ne fournie pas encore la totalité des concepts présent dans les diagrammes d'activités. La modification de la transformation réseaux de Petri vers VHDL-AMS semble obligatoire pour prendre en compte certain aspect comme l'élaboration d'équations différentielles dépendantes de plusieurs variables, ou l'exécution de méthodes. Les diagrammes d'activités ne permettent pas d'exprimer clairement les actions
Une projet de grande envergure pourrait établir les règles respectant l'ensemble des spécifications OMG et des contraintes associées aux diagrammes d'activités. Une automatisation totale de la transformation, rendant transparents les réseaux de Petri à l'utilisateur, et permettant de passer de la modélisation DA vers la simulation en une seule étape rendrait l'outil accéssible à l'utilisateur n'ayant aucune connaissance des RdPs.
L'élaboration de la chaine inverse, réseaux de Petri vers le diagramme d'activité semble pouvoir offrir des informations aux diagrammes d'activité, grâce aux invariants de places ou à la présence ou l'absence de certaines propriétés LTL dans le RdP équivalent : Création de swimline dans le DA en fonction des invariants, mise en garde pour l'utilisateur sur telle ou telle activité qui semble ne jamais s'exécuter,... .

A terme il est souhaité de joindre au diagramme d'activité, les diagrammes de séquence et de vérifier leur interopérabilité, est-ce que le diagramme d'activité étudié permet d'avoir la séquence d'exécution désirée.

References

[1] A. Semenov, A. M. Koelmans, L. Lloyd, and A. Yakovlev, "Designing an asynchronous processor using petri nets," *IEEE Micro*, vol. 17, pp. 54–64, March 1997.

[2] Q. Qinru, W. Qing, and P. Massoud, "Dynamic power management of complex systems using generalized stochastic petri nets," 2000.

[3] "Meta object facility (MOF) 2.0 core specification," 2006. Version 2.

[4] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[5] "Meta object facility (mof) 2.0 query/view/ transformation specification," 2011. Version 1.1.

[6] G. de travail WP5 TOPCASED, "Un panorama des techniques de transformation de modèles,"

[7] A. group LINA and I. Nantes, "Atl :atlas transformation language atl user manual," *OMG specification*, February 2006.

[8] "Topcased," 2011.

[9] "Eclipse," 2011.

[10] OMG, "Omg systems modeling language (omg sysml)," *OMG specification*, pp. 1–246, JUNE 2010.

[11] OMG, "Omg unified modeling language(omg uml), superstructure," *OMG specification*, pp. 1–742, MAY 2010.

[12] "Cours-uml," 2011.

[13] OMG, "Omg object constraint language (ocl), superstructure," *OMG specification*, pp. 1–256, DECEMBER 2010.

[14] R. CHAMPAGNAT, P. ESTEBAN, H. PINGAUD, and R. VALETTE, "Modeling hybrid systems by means of high-level petri nets : benefits and limitations," in *Conférence sur la Commande des Systèmes Industriels (CIS'97)*, (Belfort (France)), pp. 469–474, 1997.

[15] V. Albert, "Traduction d'un modèle de système hybride basé réseau de petri en vhdl-ams," Master's thesis, UPS, 2005.

[16] R. CHAMPAGNAT, *Supervision des systèmes discontinus : définition d'un modèle hybride et pilotage en temps-réel*. PhD thesis, Université Paul Sabatier, Toulouse, 150p., 1998. Doctorat.

[17] "Eclipse," 2011.

[18] *http ://homepages.laas.fr/bernard/tina/*.

ANNEXE A

Règles LTL

Cette annexe présente par bloc élémentaire l'ensemble des règles LTL qui permettent la vérification formelle de la transformation diagramme d'activité vers réseau de Petri. Ces règles sont établies dans le langage d'entrée de selt , le model-cheker de TINA dont voici une présentation rapide des commandes : Contantes :

- constantes : T (vrai), F (faux)
- préfixes : [] (*toujours*) , <> (*un jour*), () (*next*), − (*négation logique*)
- suffixes :

  U (*jusqu'*à), V (*tant que*), *ordre de priorité* 0

  $\Rightarrow$ (*implique*), $\Longleftrightarrow$ (*équivalent*), *ordre de priorité* 1,

  $\bigwedge$ (*et logique*), $\bigvee$ (*ou logique*), *ordre de priorité* 2

  $<=$, $>=$, $=$, *le*, *lt*, *ge*, *gt*, *ordre de priorité* 3

  $+$, *ordre de priorité* 4 $*$, ordre de priorité 5

ActionNode

- Propriété 1 :[](*ActionStart* $\Rightarrow<>$ *ActionStop*), toujours : si l'action démarre elle s'arrête un jour.
- Propriété 2 :[](*AllActionNOIncomingRun* $\Rightarrow<>$ *ActionStart*), toujours : si toutes les entrées non optionnelles sont actives alors l'action démarre un jour.
- Propriété 3 :[](*ActionStop* $\Rightarrow<>$ *AllActionNOOutgoingRun*), toujours : si l'action s'arrête, alors toutes les sorties non optionnelles démarrent un jour.

InitialNode

- Propriété 1 :[](*InitRun* $\Rightarrow<>$ *InitStop*), toujours : si le noeud initial s'exécute alors il s'arrête un jour.

– Propriété 2 :$[](InitStop \Rightarrow <> AllInitOutgoingRun)$, toujours : si le noeud initial s'arrête alors toutes ces sorties démarrent un jour.

ForkNode

– Propriété 1 :$[](ForkIncomingRun \Rightarrow <> AllForkOutgoingStart)$, toujours : si l'entrée du noeud de bifurcation est actif alors toutes ces sorties sont activées un jour.

JoinNode

– Propriété 1 :$[](AllJoinIncomingRun \Rightarrow <> Joinoutgoingstart)$, toujours : si l'ensemble des entrées du noeud de jonction sont actifs alors la sortie est activée un jour.

ControlFlow

– Propriété 1 :$[](ControlisRunning + ControlisNotRunning = 1)$, toujours : le nombre de jetons présent dans un ControlFlow doit être égal à 1, cette propriété n'est pas spécifiée par l'OMG, c'est un choix de modélisation qui restreint les possibilités de la transformation.

– Propriété 2 :$[](ControlStart \Rightarrow ()ControlRun)$, toujours : si l'arc de control du flot reçoit un jeton alors il est actif à l'instant suivant.

– Propriété 3 :$[](ControlisRunning \Rightarrow <> ControlStop)$, toujours : si l'arc de control du flot est actif alors il s'arrête un jour.

ObjectFlow

– Propriété 1 :$[](ObjectStart \Rightarrow ()ObjectRun)$, toujours : si l'arc de control de donnée reçoit un jeton alors il est actif à l'instant suivant.

– Propriété 2 :$[](ObjectisRunning \Rightarrow <> ObjectStop)$, toujours : si l'arc de control de donnée est actif alors il s'arrête un jour.

MergeNode

– Propriété 1 :$[](MergeStart \Rightarrow <> MergeStop)$, toujours : si un noeud de fusion démarre alors il s'arrête un jour.

– Propriété 2 :$[](OneofAllMergeIncomingRun \Rightarrow ()MergeStart)$, toujours : si une des entrées du noeud de fusion est actif alors le noeud de fusion démarre à l'instant suivant.

DecisionNode

– Propriété 1 :$[](DecisionStart \Rightarrow <> DecisionStop)$, toujours : si un noeud de décision démarre alors il s'arrête un jour.

– Propriété 2 :$[](DecisionIncomingRun \Rightarrow ()DecisonStart)$, toujours : si l'entrée du noeud de décison est actif alors le noeud de décision démarre à l'instant suivant.

– Propriété 3 :$[](DecisonStop \Rightarrow <> OneofAllDecisionOutgoingStart)$, toujours : si un noeud de décision s'arrête un seul noeud de sortie est activé. .

FinalNode

- Propriété 1 :$[](FinalStart \Rightarrow ()FinalRun)$, toujours : si un noeud final démarre alors il est actif à l'instant suivant.

FlowFinalNode

- Propriété 1 :$[](FlowFinalStart \Rightarrow ()FinalRun)$, toujours : si un noeud de flow final démarre alors il est actif à l'instant suivant.
- Propriété 2 :$[](FlowFinalRun \Rightarrow () - FinalRun)$, toujours : si un noeud de flow final est actif, il ne l'est plus à l'instant suivant.

Input/Output

- Propriété 1 :$[](InputStart \Rightarrow ()InputRun)$, toujours : si un noeud d'entrée démarre à l'instant suivant il est actif.
- Propriété 1 :$[](InputRun \Rightarrow <> InputStop)$, toujours : si un noeud d'entrée est actif alors il s'arrête un jour.

Send et AcceptEvent

- Propriété 1 :$[](SendStart \Rightarrow ()SendRun)$, toujours : si un noeud d'envoi de signal démarre à l'instant suivant il est actif.
- Propriété 2 :$[](SendRun \Rightarrow <> SendStop)$, toujours : si un noeud d'envoi de signal est actif alors il s'arrête un jour.

Notons que de nombreux noeuds hérite d'autre noeud, ce qui permet de ne pas avoir à établir l'ensemble des règles pour chaque noeud.

ANNEXE B

Méta-modèle ResolveTemp

Le principe du méta-modèle a été montré dans ce mémoire en voici une vue arborescente :

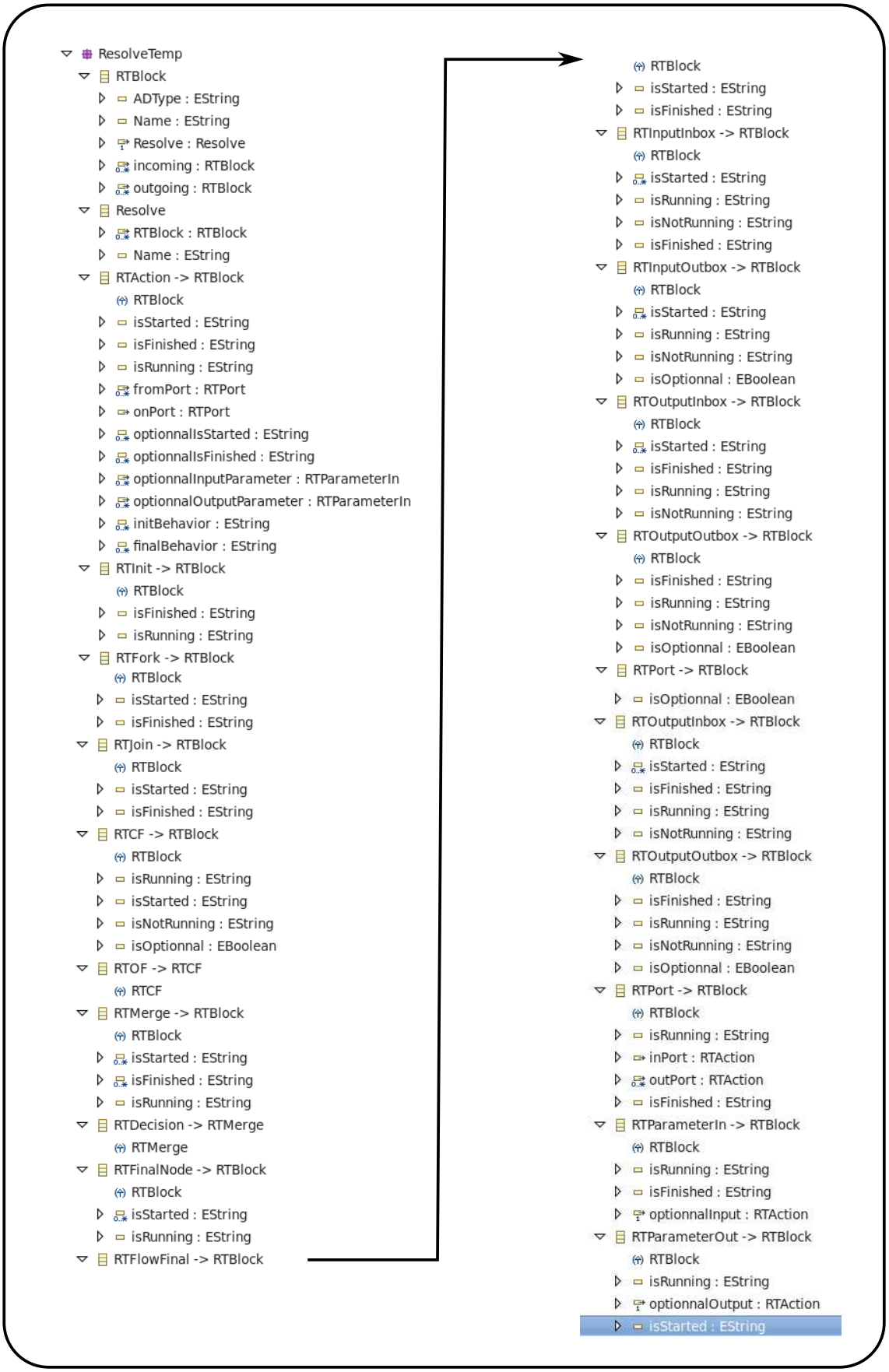

FIG. B.1 – Méta-modèle ResolveTemp

ANNEXE C

Mapping des concepts

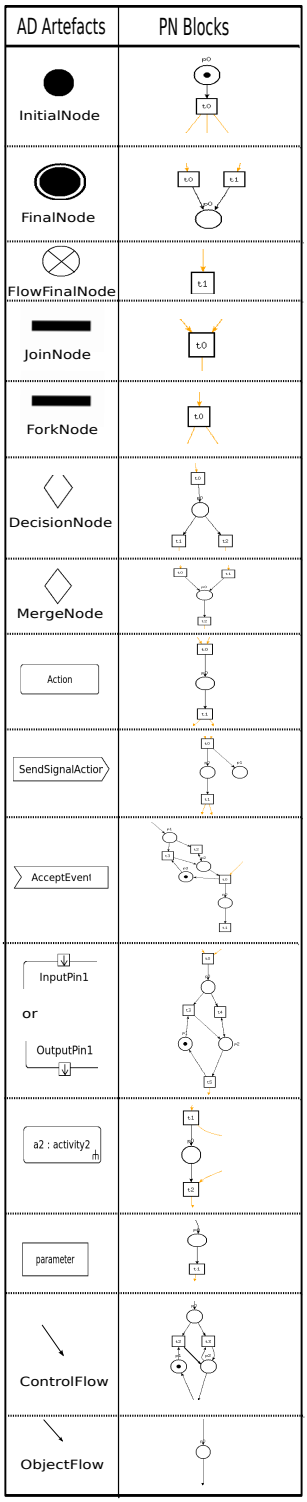

FIG. C.1 – Mapping des concepts