

ÉTUDE DE LA TRADUCTION DE DIAGRAMMES DE TYPE  
EFFBD EN RÉSEAUX DE PETRI TEMPORELS

Charlotte SEIDNER

Septembre 2006



# Table des matières

<b>Principaux sigles et acronymes</b>	<b>7</b>
<b>Introduction</b>	<b>9</b>
<b>1 Description des modèles de haut-niveau</b>	<b>11</b>
1.1 Généralités . . . . .	11
1.2 Fonctions . . . . .	12
1.2.1 Fonctions simples . . . . .	12
1.2.2 Fonctions décomposées . . . . .	12
1.2.3 Fonctions à sorties multiples . . . . .	13
1.2.4 Fonctions périodiques . . . . .	13
1.3 Flux de données et ressources . . . . .	14
1.3.1 Triggers . . . . .	14
1.3.2 Ressources . . . . .	15
1.3.3 Flux, données et ressources . . . . .	16
1.4 Structures de contrôle . . . . .	16
1.4.1 Structures d'exécutions parallèles . . . . .	17
1.4.2 Branches de choix (OR) . . . . .	17
1.4.3 Itérations et boucles . . . . .	17
<b>2 Traduction vers les réseaux de Petri temporels</b>	<b>19</b>
2.1 Généralités sur les réseaux de Petri temporels . . . . .	19
2.2 Motifs de traduction des superFFBD vers les TPN . . . . .	21
2.2.1 Fonctions . . . . .	22
2.2.2 Flux de données et ressources . . . . .	23
2.2.3 Structures de contrôle . . . . .	25
<b>3 Méta-modélisation et transformation de modèles</b>	<b>29</b>
3.1 Généralités . . . . .	29
3.1.1 Principes de la méta-modélisation . . . . .	29
3.1.2 Transformation et génération de modèles . . . . .	31
3.2 Description des méta-modèles . . . . .	31
3.2.1 Méta-modèle d'entrée M2_Process . . . . .	31
3.2.2 Méta-modèle de sortie M2_Romeo . . . . .	34
3.3 Écriture des règles de transformation . . . . .	35

<b>4 Résultats</b>	<b>37</b>
4.1 Résultats théoriques . . . . .	37
4.2 Résultats pratiques . . . . .	37
<b>Conclusion</b>	<b>41</b>
<b>A Fichier romeo.xsd</b>	<b>43</b>
<b>B Fichier RomeoXMLWriter.java</b>	<b>47</b>
<b>C Algorithme de transformation des structures parallèles</b>	<b>49</b>
<b>Bibliographie</b>	<b>51</b>

# Table des figures

1.1	Exemple de diagramme EFFBD [Lon95]	12
1.2	Fonction à sorties multiples décomposée	13
1.3	Fonction périodique	14
1.4	Fonction recevant deux triggers de terminaison	15
2.1	Exemple de réseau de Petri temporel	20
2.2	Exemples de TPN avec arcs de vidange (g.), de lecture (c.) et d'inhibition logique (d.)	21
2.3	Fonction simple et traduction en TPN	22
2.4	Fonction décomposée et traduction en TPN	22
2.5	Traduction en TPN de la fonction périodique de la figure 1.3	23
2.6	Fonction émettant un flux	24
2.7	Branches parallèles et traduction en TPN	25
2.8	Branches parallèles avec une branche de terminaison	26
2.9	Branches de choix et traduction en TPN	26
2.10	Structure d'itérations et traduction en TPN	27
2.11	Structure de boucle avec nœud de sortie	27
3.1	Modélisation d'une application informatique en niveaux d'abstraction	30
3.2	Transformation de modèles par l'approche MDA	31
3.3	Diagramme Behavior Relations (M2_Process)	33
3.4	Diagramme Behavior Type Hierarchy (M2_Process)	34
3.5	Diagramme Functions and Flows (M2_Process)	35
3.6	Méta-modèle M2_Romeo	36
4.1	Exemple de diagramme superFFBD	38
4.2	Traduction en TPN du modèle superFFBD de la figure 4.1	39



# Principaux sigles et acronymes

DTD	<i>Document Type Definition</i>
EFFBD	<i>Enhanced Functional Flow Block Diagram</i>
EMF	<i>Eclipse Modeling Framework</i>
IS	ingénierie système
MDA	<i>Model Driven Architecture</i>
MOF	<i>Meta Object Facility</i>
OMG	<i>Object Management Group</i>
PIM/PSM	<i>Platform Independant/Specific Model</i>
RdP	réseau de Petri
SysML	<i>Systems Modeling Language</i>
TPN	<i>Time Petri Net(s)</i>
UML	<i>Unified Modeling Language</i>
XMI	<i>XML Metadata Interchange</i>
XML	<i>eXtensible Markup Language</i>
XSD	<i>XML Schema Description</i>



# Introduction

Est-il encore besoin de rappeler combien les processus de vérification et de validation sont primordiaux au cours du développement d'un projet d'ingénierie système (IS) ? La certitude d'avoir « construit un *bon* système » (vérification) et celle d'avoir « construit *le bon* système » (validation) sont en effet au cœur du processus d'IS tel qu'il est décrit dans la norme IEEE-1220 [IEE94].

Cependant, comme nous l'avons montré lors du travail préliminaire de synthèse bibliographique, assez peu d'outils de modélisation de haut-niveau du domaine de l'IS ou du développement de logiciels se prêtent « facilement » à des vérifications formelles par *model-checking*, via une traduction vers un formalisme tel que les réseaux de Petri [Pet62] ou les automates temporisés [AD94].

Notre étude avait en particulier conclu à l'intérêt de décrire les systèmes à vérifier au moyen de diagrammes de type EFFBD<sup>1</sup> [Lon95], qui permettent de capturer le comportement du système au travers de sa structure fonctionnelle dynamique et des échanges de données entre les fonctions, plutôt que d'employer des langages de modélisation tels qu'UML<sup>2</sup> ou SysML<sup>3</sup>.

D'autre part, le formalisme retenu pour la description des systèmes en vue de leur vérification est celui des réseaux de Petri temporels (*Time Petri Nets*, TPN) introduits par Merlin [Mer74]. Ce choix nous permet de bénéficier de développements autour du logiciel Roméo<sup>4</sup>, créé au sein de l'équipe « Systèmes Temps Réel » du laboratoire de l'IRCCyN [GLMR05].

Les objectifs de ce stage pratique de Master de Recherche, qui s'est effectué en majeure partie au sein de la société Sodius, étaient au nombre de trois :

- étude des modèles de haut-niveau et affinement de leur sémantique ;
- construction des motifs en TPN des éléments de haut-niveau ;
- réalisation d'un premier outil de transformation.

Signalons par ailleurs que ce travail s'insère dans le cadre du développement d'une application, Kimono, réalisée pour la DGA (*Direction Générale de l'Armement*), par un groupement d'entreprises spécialisées dans l'ingénierie système, dont la société Sodius. Cette application permet notamment de modéliser des systèmes militaires complexes, aussi bien dans leur comportement avec leur environnement que dans les aspects concernant les implémentations physiques. Enfin, ce stage de Master constitue notre première étape d'un travail qui sera mené au cours d'une thèse industrielle, en partenariat avec l'IRCCyN et Sodius.

---

<sup>1</sup> *Enhanced Functional Flow Block Diagram*.

<sup>2</sup> *Unified Modeling Language*.

<sup>3</sup> *Systems Modeling Language*.

<sup>4</sup> <http://romeo.rts-software.org/>

Ce document présente dans un premier temps les diagrammes dits « superFFBD », un formalisme fortement inspiré des diagrammes EFFBD et qui a été développé pour les besoins de l'application Kimono (*chapitre 1*). Nous présentons ensuite la traduction en TPN des éléments décrits dans la première partie (*chapitre 2*). Le chapitre 3 décrit les principes de la *méta-modélisation* – domaine d'expertise de Sodus – qui a permis la réalisation pratique de la transformation des modèles superFFBD vers les TPN. Nous présentons enfin quelques résultats, aussi bien théoriques que pratiques, obtenus sur ces transformations (*chapitre 4*).

# Chapitre 1

## Description des modèles de haut-niveau

Nous présentons dans ce chapitre les diagrammes superFFBD, qui offrent un formalisme de haut-niveau et permettent de modéliser un système sur lequel, par le biais des transformations présentées dans les chapitres suivants, on cherchera à vérifier formellement un ensemble de preuves.

Les outils de modélisation sur lesquels se basent ces diagrammes étant issus du monde et des techniques de l'ingénierie des systèmes, assez peu d'auteurs se sont intéressés à leur formalisation et aucun, à notre connaissance, n'en a proposé de sémantique formelle [Her04]. Pour notre part, nous avons choisi de décrire cette sémantique sous forme textuelle, ce qui constituera une première étape pour la construction d'une sémantique formelle, étude que nous nous proposons d'effectuer dès le début du travail de thèse.

### 1.1 Généralités

Les diagrammes superFFBD ont été développés pour les besoins de l'application Kimono ; comme précisé ci-dessus, ils reprennent un grand nombre d'outils de modélisation couramment mis en œuvre en ingénierie système. En particulier, ces diagrammes s'inspirent fortement des EFFBD et des outils proposés par le logiciel CORE<sup>1</sup>. On rappelle qu'un diagramme EFFBD se présente comme un enchaînement de constructions placées sur des branches, se lisant de la gauche vers la droite. Un exemple de diagramme est donné figure 1.1. Nous renvoyons le lecteur au mémoire de séminaire bibliographique pour une description plus poussée de ces diagrammes et du logiciel.

Les principaux éléments modélisés dans un diagramme superFFBD sont :

- les fonctions, qui représentent des activités élémentaires du système ;
- les structures de contrôle usuelles (*choix, exécutions parallèles, itérations, etc.*) ;
- les flux de données ;
- les ressources ;
- les critères de performances attendues ;
- les constituants physiques auxquels sont allouées les fonctions.

Seuls les éléments des diagrammes décrivant les flux de contrôle et, plus généralement, le comportement dynamique du système sont traduits en réseau de Petri (RdP). Ainsi,

---

<sup>1</sup> Logiciel développé par Vitech Corp. <http://www.vitechcorp.com>

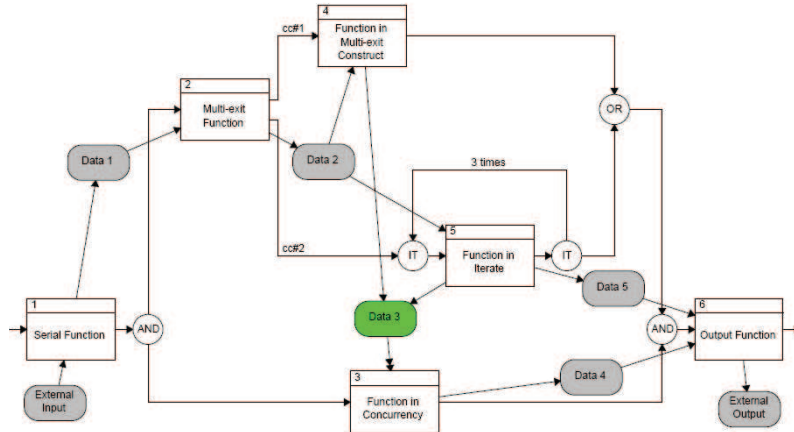


FIG. 1.1 – Exemple de diagramme EFFBD [Lon95]

nous ne présentons dans la suite de ce chapitre que la sémantique des fonctions, des flux de données et des ressources ainsi que des structures de contrôle. Par ailleurs, nous désignerons dans la suite de ce document sous le terme commun de « construction » (*Constructs*) les fonctions et les structures de contrôle.

## 1.2 Fonctions

Les fonctions représentent l'élément de plus bas niveau dans le diagramme. Toutes les descriptions de fonctions incluent :

- un identifiant  $f\_ID$  unique dans le modèle ;
- les identifiants des flux et ressources d'entrée et de sortie (cf. *infra*) ;
- un intervalle temporel  $[a; b]$  caractérisant la durée d'exécution de la fonction.

Les paramètres  $a$  (fin d'exécution au plus tôt) et  $b$  (fin d'exécution au plus tard) sont des entiers naturels ;  $b$  est éventuellement égal à  $a^1$  ou à l'infini.

Tout diagramme a exactement une fonction initiale et une fonction finale. Toutes les fonctions ont exactement une construction en entrée (aucun dans le cas de la fonction initiale) et une construction en sortie (aucun dans le cas de la fonction finale), sauf pour les fonctions à sorties multiples, présentées ci-dessous.

### 1.2.1 Fonctions simples

Une fonction est dite *validée* par le flux de contrôle ce qui permet ensuite de l'exécuter. La durée d'exécution est un réel  $t \in [a; b]$ .

### 1.2.2 Fonctions décomposées

Les formalismes EFFBD et superFFBD autorisent la modélisation du système par niveaux hiérarchiques, ce qui permet d'avoir une vision macroscopique du système, notamment lors de simulations. Ainsi, il est possible d'attacher à une fonction un sous-scénario, qui représente la décomposition dynamique de cette fonction<sup>2</sup>.

<sup>1</sup> Modélisation d'une fonction à durée déterministe.

<sup>2</sup> Cette décomposition ne doit pas être confondue avec la *décomposition fonctionnelle*, qui donne une vue statique du système et ne traduit donc pas son comportement dynamique.

On considère qu'une fonction décomposée est en exécution lorsqu'une des fonctions de son scénario au moins est en exécution ; elle est *active* lorsque le flux de contrôle se situe dans la décomposition.

Enfin, l'intervalle temporel porté par la fonction-mère n'a pas la valeur sémantique usuelle : en effet, cet intervalle n'est précisé qu'à des fins de vérification. On souhaitera en particulier vérifier que l'exécution de la décomposition s'effectue toujours dans l'intervalle de temps de la fonction-mère.

### 1.2.3 Fonctions à sorties multiples

Les fonctions à sorties multiples comportent plusieurs branches de sortie se rejoignant par la suite sur un nœud OR. Dans le cas où la fonction n'a pas de décomposition, on se ramène à une fonction simple suivie d'une structure de choix (cf. partie 1.4.2).

Dans le cas contraire, on exécute le scénario de décomposition puis l'une des branches de sortie de la fonction-mère. Le choix de la branche se fait au moyen des nœuds de sortie EXIT présents dans le sous-scénario ; celui-ci doit contenir autant de nœuds EXIT que la fonction-mère comporte de branches de sortie. Un exemple de fonction décomposée à sorties multiples est donné figure 1.2.

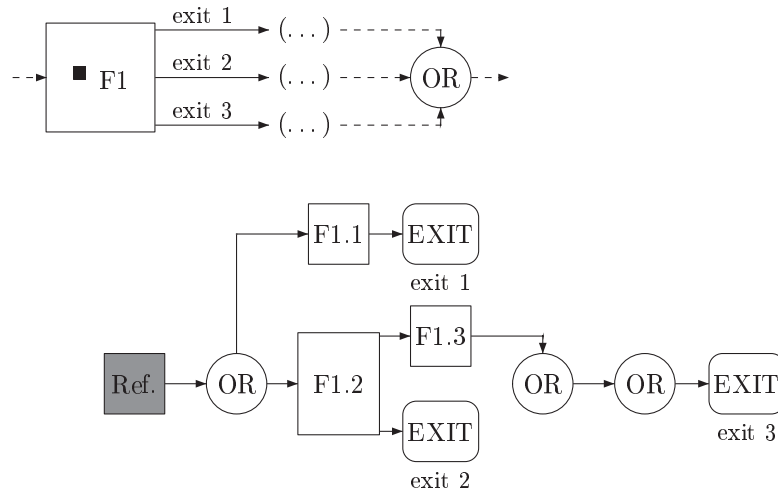


FIG. 1.2 – Fonction à sorties multiples décomposée

Après analyse de ces structures, il est apparu que seules certaines constructions peuvent être employées dans la décomposition pour contenir les nœuds EXIT :

- branches parallèles (AND) (cf. partie 1.4.1) : dans ce cas, il ne peut y avoir d'élément après le nœud AND fermant ;
- branches parallèles avec branche de terminaison (kill) : le ou les modificateurs (kill) ne peuvent être sur les branches comportant un nœud de sortie (redondance) ;
- branches de choix : pas de condition particulière.

### 1.2.4 Fonctions périodiques

Les fonctions périodiques forment une innovation par rapport aux EFFBD « classiques » ; elles permettent typiquement de modéliser des capteurs effectuant des mesures périodiques, avec une période  $p$  (figure 1.3). Elles peuvent être insérées à tout endroit du

diagramme, se situent entre deux blocs quelconques et émettent (en général) au moins un trigger. Le comportement est le suivant :

- validation de la fonction par le flux issu de la construction amont ;
- exécution de la fonction et initialisation d’une horloge ;
- émission des triggers une fois l’exécution terminée ;
- validation de la construction aval ;
- lorsque l’horloge atteint la valeur  $p$  : remise à zéro de l’horloge et nouvelle exécution de la fonction (qui rentre dans un cycle exécution  $\leftrightarrow$  attente de l’échéance de l’horloge, jusqu’à la terminaison globale de la fonction).

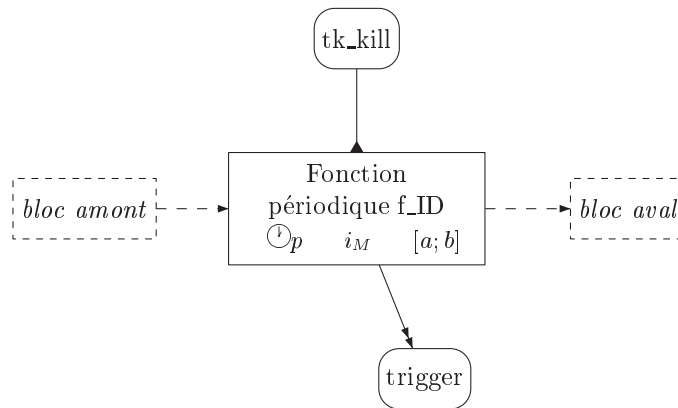


FIG. 1.3 – Fonction périodique

Précisons que la nouvelle exécution de la fonction ne conduit pas à une nouvelle validation de la construction aval mais se comporte comme une boucle parallèle, « indépendante » du reste du modèle. La terminaison globale de la fonction peut s’effectuer de deux façons : au bout d’un certain nombre d’itérations  $i_M$  et/ou par réception d’un (ou plusieurs) trigger(s) de terminaison `tr_kill` (cf. partie 1.3.1). La désactivation de la fonction se traduit par *l’absorption* du flux de contrôle : il n’y a pas de sortie spécifique.

La fonction périodique peut comprendre une décomposition hiérarchique mais, pour des raisons de simplicité, ne peut être à sorties multiples. Dans tous les cas, on suppose que la fonction périodique a toujours le temps de s’exécuter entièrement avant l’échéance d’horloge suivante. Pour une fonction « simple », il suffit d’avoir  $b \leq p$  pour respecter cette condition.

Enfin, dans la définition actuelle, la fonction doit d’abord être désactivée (au bout du nombre maximal d’itérations ou par un trigger de terminaison) avant d’être réactivée.

## 1.3 Flux de données et ressources

### 1.3.1 Triggers

Les diagrammes EFFBD permettent de distinguer deux types de données, les « triggers » et les « data-store ». Ces derniers n’interviennent pas dans la détermination du comportement du système : ils représentent en effet des données non causale, dont la présence ne déclenche pas l’exécution des fonctions les recevant.

### Émission de triggers

Une fonction produit ses triggers une fois que son exécution est achevée et avant de transmettre le flux de contrôle à la construction suivante.

### Réception de triggers

Les triggers définissent une condition supplémentaire pour l'exécution d'une fonction qui a été activée par le flux de contrôle. En effet, une fonction activée doit attendre que l'ensemble de ses triggers soit présents avant de s'exécuter.

Nous avons également défini des *triggers de terminaison*, qui forment une innovation par rapport aux EFFBD classiques. Ces triggers permettent de forcer la terminaison d'une fonction donnée, qui est alors à sorties multiples (une sortie « normale » et autant de sorties que de triggers de terminaison), sauf s'il s'agit d'une fonction périodique, dont le cas a été évoqué plus haut. La figure 1.4 illustre le cas d'une fonction  $f\_ID$  recevant deux triggers de terminaison  $tk\_1$  et  $tk\_2$ .

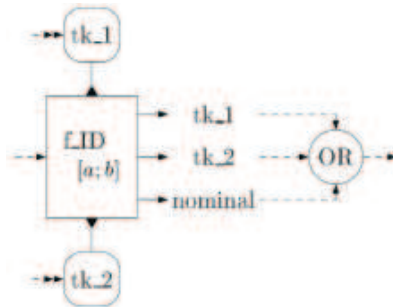


FIG. 1.4 – Fonction recevant deux triggers de terminaison

Par ailleurs, le logiciel CORE introduit la notion de « *time out* », que nous avons choisi de garder. On définit un délai maximal  $\Delta_M$  d'attente entre la validation d'une fonction et son exécution, ce qui correspond à la durée maximale d'attente de l'ensemble de triggers (hors les triggers de terminaison). Si ce délai est atteint, la fonction (implicitement à sorties multiples) sort par une branche labélisée **TimeOut** ; sinon, la sortie se fait – après exécution de la fonction – par la branche nominale.

Enfin, on considère qu'un trigger est une entité non physique, *i.e. distribuée* à l'ensemble des fonctions qui l'attendent.

### 1.3.2 Ressources

La gestion des ressources est un ajout notable aux EFFBD, mais il convient de noter que cette notion est déjà présente dans CORE. À l'heure actuelle, les ressources, qui peuvent être consommées ou produites par un nombre quelconque de fonctions, ne peuvent être manipulées qu'en des quantités entières ou rationnelles. La définition d'un taux de production ou de consommation d'une ressource (*et la traduction vers des RdP dont la sémantique reste à préciser*) pourrait faire l'objet d'une étude au cours du travail de thèse.

### Production de ressources

Tout comme précédemment, une fonction produit ses ressources une fois son exécution achevée, et avant de transmettre le flux de contrôle à la construction suivante.

### Attente de ressources

En ce qui concerne la consommation de ressources, celles-ci se distinguent des triggers présentés ci-dessus par le fait qu'elles peuvent ne pas se trouver en quantité suffisante pour satisfaire toutes les demandes, ce qui conduit à des mises en attente. Deux comportements d'attente sont définis : la fonction peut être en mode « normal » ou en mode « *Acquire Available* » (notons que cette distinction est directement inspirée du fonctionnement proposé par CORE).

Dans le mode normal, la fonction attend que la totalité de la part de ressource demandée soit disponible. De plus, l'attente est non bloquante pour les autres fonctions faisant une requête sur la ressource. Ainsi, si la demande d'une fonction F2, nécessitant moins de ressource qu'une fonction F1, peut être satisfaite, c'est la fonction F2 qui obtiendra la quantité nécessaire de la ressource, même si F1 attend depuis plus longtemps.

Dans le second cas, la fonction en mode « *Acquire Available* » devient prioritaire devant les autres et consomme la ressource au fur et à mesure de sa production, jusqu'à recevoir la quantité nécessaire à son exécution. De plus, une priorité est donnée à la première fonction en attente. Une fois que cette fonction a obtenu la quantité nécessaire de ressource, la priorité n'est pas transférée à une fonction de type « *Acquire Available* » déjà en attente mais sera prise par la prochaine fonction de ce type à faire une requête sur la ressource.

### 1.3.3 Flux, données et ressources

Il a été décidé lors de la construction du méta-modèle `M2_Process` décrivant les diagrammes superFFBD (cf. chapitre 3) de fusionner dans une même notion des données (triggers ou data-store) et les ressources. Un paramètre booléen `isDestructive`, attaché à la relation entre une fonction et le flux qu'elle reçoit permet de discriminer entre triggers (`isDestructive` faux) et ressources (`isDestructive` vrai). De plus, le niveau de flux demandé (ou produit) par une fonction est quantifiée *via* un entier  $n$ .

Dans le cas où `isDestructive` est faux, la fonction recevant le flux est exécutée après avoir été validée par le flux de contrôle et après avoir vérifié que le flux attendu est présent dans une quantité supérieure ou égale à  $n$ .

Dans le cas contraire, la fonction est exécutée après avoir été validée par le flux de contrôle et après *consommation* du flux dans la quantité  $n$ .

Enfin, la notion de `timeOut` définie pour les triggers s'applique également au cas des flux : on le définit comme étant le temps maximal pouvant s'écouler entre l'activation de la fonction et la réception de tous ses flux.

## 1.4 Structures de contrôle

Les structures de contrôle sont en général formées de deux nœuds qui encadrent un certain nombre de branches, elles-mêmes comportant un nombre quelconque de constructions. Seules les imbrications de structures « bien formées » sont autorisées : on sort des structures imbriquées dans l'ordre inverse d'entrée. Les structures possèdent en outre un identifiant unique, et chaque construction possède un attribut identifiant la structure dont il dépend.

### 1.4.1 Structures d'exécutions parallèles

Les structures de type branches parallèles sont les seules constructions (avec les fonctions périodiques évoquées plus haut) produisant plusieurs flux de contrôle en parallèle.

#### Branches parallèles (AND)

La structure se compose d'un nœud d'entrée AND d'où partent  $n$  branches et d'un nœud de sortie AND où convergent ces  $n$  branches. Le comportement suit les règles suivantes :

- entrée dans la structure  $\rightarrow$  validation simultanée de la première construction de chaque branche ;
- terminaison de la dernière construction de **chaque** branche  $\rightarrow$  sortie de la structure.

#### Branche de terminaison forcée (kill)

Cette structure permet la terminaison forcée des fonctions de la structure lorsqu'une branche marquée du modificateur **kill** se termine ; un nombre *quelconque* de branches peuvent être marquées. Le comportement est donc le suivant :

- entrée dans la structure  $\rightarrow$  validation simultanée de la première construction de chaque branche ;
- terminaison de la dernière construction d'une branche **kill**  $\rightarrow$  terminaison (*éventuellement forcée*) de chaque autre branche  $\rightarrow$  sortie de la structure.

#### Répliques (RP)

La réplique permet de créer  $n$  instances parallèles (et indépendantes) d'une même branche ; ces instances sont coordonnées entre elles par une branche de contrôle.

Il s'agit d'une certaine manière d'un raccourci de notation, la structure étant équivalente à un ensemble de  $n + 1$  branches parallèles, dont  $n$  branches identiques.

Le comportement suivi par la structure est alors :

- entrée dans la structure  $\rightarrow$  validation de la première construction de chaque instance de la branche répliquée et de la branche de contrôle ;
- terminaison de la dernière construction de chaque branche  $\rightarrow$  sortie de la structure.

### 1.4.2 Branches de choix (OR)

Une structure par branches de choix comprend deux nœuds OR encadrant  $n$  branches ; les règles de comportement sont les suivantes :

- entrée dans la structure  $\rightarrow$  validation de la première construction *de l'une des  $n$  branches* ;
- terminaison de la dernière construction de la branche choisie à l'entrée dans la structure  $\rightarrow$  sortie de la structure.

Le choix de l'une ou l'autre des branches est déterminé par une probabilité attachée à chaque branche (si ces probabilités ne sont pas précisées, le choix est équiprobable).

### 1.4.3 Itérations et boucles

#### Itérations (IT)

La structure d'itération comprend une branche à itérer et un « domaine d'itération ». À l'heure actuelle, il s'agit d'un nombre d'itérations  $i$ . Le comportement de la structure

est le suivant :

- entrée dans la structure → validation de la première construction de la branche et initialisation du compteur d'itérations ;
- terminaison de la dernière construction de la branche :
  - si le nombre d'itérations  $i$  est atteint → sortie de la structure ;
  - sinon → incrémentation du compteur d'itérations et validation de la première construction de la structure.

### **Boucles infinies (LP)**

Une structure de boucle comprend deux nœuds LP encadrant une branche comprenant un nombre quelconque de constructions, ainsi qu'une branche de retour. Par défaut, les boucles sont exécutées indéfiniment :

- entrée dans la structure → validation de la première construction de la branche ;
- terminaison de la construction précédant le second nœud LP → validation de la première construction de la structure.

### **Sorties de boucle (LE)**

L'emploi d'un nœud de sortie de boucle (**Loop Exit**, LE) permet de définir des conditions de sortie des boucles ; il est limité aux mêmes structures que pour les nœuds EXIT. En outre, une structure de boucle peut contenir un nombre quelconque de nœuds LE. Les règles de comportement sont alors :

- entrée dans la structure → validation de la première construction de la branche ;
- terminaison de la construction précédant un nœud LE → sortie de la structure (et terminaison éventuellement forcée des fonctions de la boucle) ;
- terminaison de la construction précédant le second nœud LP → validation de la première construction de la structure.

## Chapitre 2

# Traduction vers les réseaux de Petri temporels

Le chapitre précédent s'est attaché à présenter les modèles de haut-niveau que nous cherchons à transformer, en leur donnant une sémantique aussi précise que possible. Après une brève description des réseaux de Petri temporels (où nous supposons le lecteur déjà familiarisé avec les RdP classiques), nous décrivons dans la suite de ce chapitre les motifs de traductions des éléments des modèles superFFBD.

### 2.1 Généralités sur les réseaux de Petri temporels

Les TPN forment avec les réseaux de Petri *temporisés* les deux principales extensions temporelles aux réseaux de Petri classiques. Ils modélisent l'écoulement du temps sous forme d'un intervalle de temps ; selon l'élément du réseau qui porte cette information, on parle de réseaux A-temporels (*intervalle sur les arcs*), P-temporels (*places*) ou T-temporels (*transitions*) [Mer74]. Nous nous intéresserons ici uniquement à cette dernière variante.

De façon informelle, on associe à chaque transition  $t$  une horloge  $x_t$  et un intervalle  $[a, b]$  modélisant les instants où l'on peut tirer cette transition, à partir du moment où elle est sensibilisée par le marquage de ses places amont. La transition peut être franchie si l'horloge est continûment sensibilisée pendant une durée comprise dans l'intervalle  $[a, b]$ . De plus, nous travaillons ici sous les hypothèses de la *sémantique forte*, qui impose que la transition soit obligatoirement tirée avant la date  $b$  (à moins d'avoir été désensibilisée par le tir d'une autre transition du réseau).

Pour le TPN simple illustré figure 2.1, la transition  $t$  est sensibilisée par la présence d'un jeton dans la place  $p1$  ; elle peut être tirée à une date (non nécessairement entière) comprise dans l'intervalle  $[1, 4]$ . Le tir de la transition entraîne le transit du jeton de la place  $p1$  vers la place  $p2$ , comme pour un RdP classique.

Nous donnons ci-dessous la définition formelle des réseaux de Petri T-temporels.

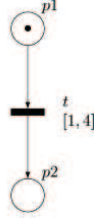


FIG. 2.1 – Exemple de réseau de Petri temporel

**Définition 1** (Réseau de Petri T-temporel) *Un réseau de Petri T-temporel est un  $n$ -uplet  $\mathcal{N} = (P, T, \bullet(\cdot), (\cdot)\bullet, a, b, M_0)$  avec :*

- $P = \{p_1, \dots, p_m\}$  un ensemble fini non vide de places ;
- $T = \{t_1, \dots, t_n\}$  un ensemble fini non vide de transitions ;
- $\bullet(\cdot) \in (\mathbb{N}^P)^T$  la fonction d'incidence amont ;
- $(\cdot)\bullet \in (\mathbb{N}^P)^T$  la fonction d'incidence arrière ;
- $a \in (\mathbb{Q}^+)^T$  et  $b \in (\mathbb{Q}^+ \cup \{\infty\})^T$  les fonctions donnant pour chaque transition ses dates de tir au plus tôt et au plus tard, respectivement ( $a \leq b$ ) ;
- $M_0 \in \mathbb{N}^P$  le marquage initial du réseau.

Un *marquage* du TPN est un vecteur de  $\mathbb{N}^P$  tel que pour toute place  $p$ ,  $M(p)$  est le nombre de jetons dans cette place. Une transition  $t$  est *sensibilisée* par un marquage  $M$  si le nombre de jetons dans chacune de ses places amont est supérieur ou égal au poids de l'arc entre la place et la transition. On note pour la suite  $t \in \text{enabled}(M)$ . Enfin, une transition  $t$  est dite *nouvellement sensibilisée* par le tir d'une transition  $t'$  si elle est sensibilisée par le marquage  $M - \bullet(t') + (t')\bullet$  mais pas par le marquage  $M - \bullet(t')$ . On notera alors  $t \in \uparrow \text{enabled}(M, t')$ .

La sémantique d'un TPN peut se donner sous la forme d'un *système de transitions temporisé*, où les états se définissent comme l'association d'un marquage  $M$  et d'un vecteur de valuations d'horloges  $v$ .

**Définition 2** (Sémantique d'un réseau de Petri T-temporel) *La sémantique d'un TPN  $\mathcal{N}$  est définie par le système de transition temporisé  $\mathcal{S}_{\mathcal{N}} = (Q, Q_0, T, \rightarrow)$  tel que*

- $Q = \mathbb{N}^P \times (\mathbb{R}^+)^T$
- $Q_0 = (M_0, \bar{0})$
- $\rightarrow \subseteq Q \times \{T \cup (\mathbb{R}^+)\} \times Q$  est la relation de transition incluant transitions continues et discrètes :
- la relation de transition continue est définie par :

$$\forall \delta \in \mathbb{R}^+ (M, v) \xrightarrow{\delta} (M', v') \text{ssi} \begin{cases} v' = v + \delta \\ \forall t_k \in T, t_k \in \text{enabled}(M) \Rightarrow v'(t_k) \leq b(t_k) \end{cases}$$

- la relation de transition discrète est définie par :

$$\forall t \in T (M, v) \xrightarrow{t} (M', v') \text{ssi} \begin{cases} t \in \text{enabled}(M) \\ a(t) \leq v(t) \leq b(t) \\ M' = M - \bullet(t) + (t)\bullet \\ \forall t_k \in T, v'(t_k) = \begin{cases} 0 & \text{si } t_k \in \uparrow \text{enabled}(M, t) \\ v(t_k) & \text{sinon} \end{cases} \end{cases}$$

Outre ces éléments classiques, le logiciel Roméo implémente trois types d'arcs supplémentaires, utilisés dans les traductions proposées ci-dessous. Ils sont tous dirigés d'une place vers une transition :

- arcs de *vidange* (*flush arcs*), terminés par un diamant plein ;
- arcs de *lecture* (*read arcs*), terminés par un diamant blanc ;
- arcs *inhibiteurs logiques* (*logical inhibitor arcs*), terminés par un rond plein.

Ces trois nouveaux types d'arcs sont illustrés figure 2.2. Le premier impose que le tir de la transition concernée *vide* la place  $p_1$  de la totalité de ses jetons. Les deux autres imposent une condition supplémentaire au tir de la transition par ailleurs sensibilisée par le marquage de ses places amont reliées par un arc « classique ». Pour l'arc de lecture, la place  $p'_1$  doit contenir au moins  $k$  jetons pour pouvoir tirer la transition  $t$  alors que pour l'arc inhibiteur, la présence d'au moins un jeton dans la place  $p''_1$  empêche le tir de la transition  $t$ . Dans les deux cas, le tir de la transition  $t$  n'affecte pas le marquage des places  $p'_1$  ou  $p''_1$ .

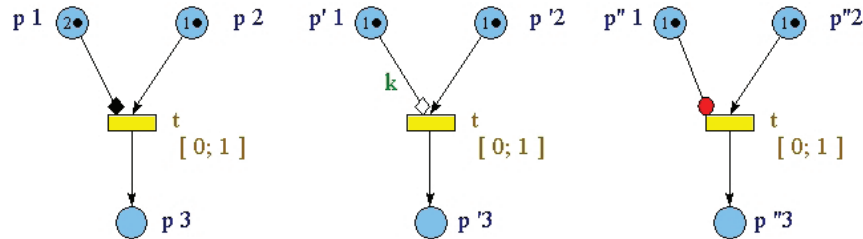


FIG. 2.2 – Exemples de TPN avec arcs de vidange (g.), de lecture (c.) et d'inhibition logique (d.)

## 2.2 Motifs de traduction des superFFBD vers les TPN

La traduction des modèles de haut-niveau se fait sous les hypothèses suivantes :

- pas d'incomplétude dans le modèle d'entrée (i.e. *pas de points non reliés, en particulier*) ;
- pas de faute syntaxique ou sémantique par rapport aux règles décrites dans le chapitre précédent.

Nous avons considéré qu'il était plus clair de présenter les traductions sous la forme d'exemples et de figures plutôt que d'algorithmes. Précisons en outre que les traductions décrites ci-dessous ne prétendent pas être optimales en terme de nombre de places, de transitions ou d'arcs. De plus, les TPN n'ont pas une expressivité suffisante pour décrire des comportements où des entités telles que des données sont différenciables : il n'est ainsi pas possible, par exemple, de modéliser des files d'attentes de données ou de fonctions selon des protocoles FIFO, LIFO, etc. Il pourrait donc s'avérer utile, par la suite, de recourir à d'autres modèles formels, tels que les réseaux de Petri colorés [Jen87].

Enfin, chaque traduction de construction fournit une transition de sortie qui constitue le point d'entrée des structures suivantes.

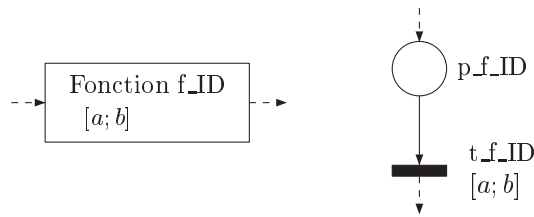


FIG. 2.3 – Fonction simple et traduction en TPN

## 2.2.1 Fonctions

### Fonctions simples

Dans le cas des fonctions simples, la traduction est immédiate : le motif en TPN comprend une place  $p\_f\_ID$ , modélisant l'activité de la fonction, et une transition  $t\_f\_ID$  (cf. figure 2.3). La fonction est en exécution lorsqu'au moins un jeton est présent dans la place  $p\_f\_ID$ .

En ce qui concerne la sémantique du motif, la transition  $t\_f\_ID$  ne peut être tirée  $a$  unités de temps avant l'arrivée d'un jeton dans la place et doit être tirée au plus tard  $b$  unités de temps après l'arrivée du jeton, modélisant ainsi une fonction dont la durée d'exécution appartient à l'intervalle  $[a; b]$ .

### Fonctions décomposées

Comme précisé au chapitre précédent, le formalisme superFFBD autorise la modélisation du système par niveaux hiérarchiques ; en revanche, la modélisation par TPN impose de mettre tout le système « à plat ».

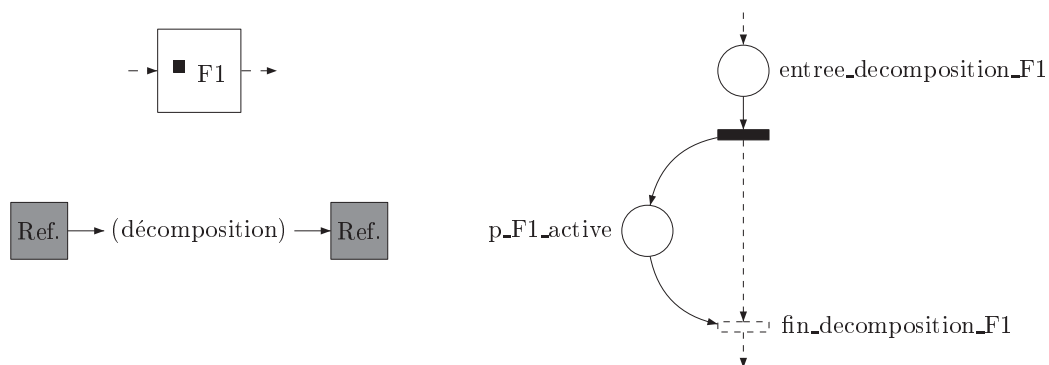


FIG. 2.4 – Fonction décomposée et traduction en TPN

Pour simplifier le suivi de l'activité d'une fonction-mère, on a choisi d'ajouter une place  $p\_f\_active$  (cf. figure 2.4) : l'entrée dans la décomposition amène un jeton dans cette place, consommé lors de la sortie de la décomposition.

### Fonctions à sorties multiples

Dans le cas où la fonction à sorties multiples n'a pas de décomposition (et par suite, aucun nœud EXIT), on peut se ramener à une structure de choix, dont la traduction est présentée plus bas. Dans le cas contraire, la traduction s'effectue comme pour une fonction décomposée « normale », les nœuds EXIT permettant de relier la branche de la décomposition en cours de traitement à la branche de sortie de la fonction correspondante. Ce lien est simplement matérialisé par un arc reliant la dernière transition de branche de la décomposition à la première place de la traduction de la branche de sortie.

### Fonctions périodiques

La figure 2.5 correspond à la traduction en TPN de la fonction périodique illustrée figure 1.3, p.14.

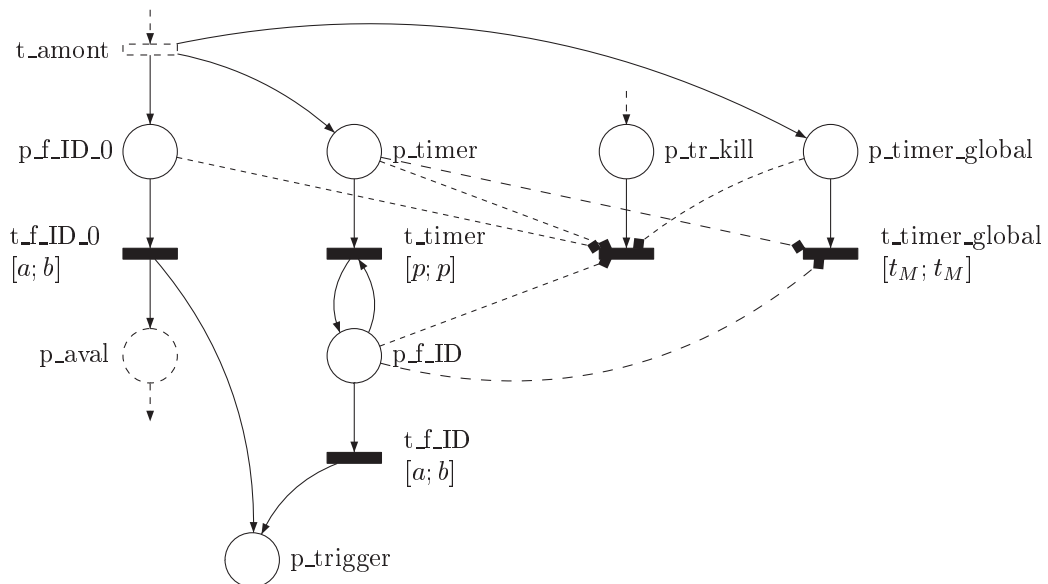


FIG. 2.5 – Traduction en TPN de la fonction périodique de la figure 1.3

La terminaison de la fonction par le trigger de terminaison `tr_kill` est réalisée au moyen d'arcs de vidange. La désactivation après  $i_M$  itérations est quant à elle obtenue avec un second timer, modélisé par la place `p_timer_global` et la transition `t_timer_global`. L'intervalle temporel correspondant à la transition est alors  $[t_M; t_M]$  avec  $t_M = i_M \times p$ .

#### 2.2.2 Flux de données et ressources

Tous les flux sont modélisés par la présence de jetons dans des places `p_flux`; une place `p_prio_flux`, dont le marquage est initialisé à 1, est attachée à chaque flux. Si plusieurs fonctions en mode d'attente « Acquire Available » font une requête sur le flux, cette construction permet de donner la priorité à celle qui attend depuis le plus longtemps.

### Émission de flux

La figure 2.6 illustre le cas d'une fonction « simple » émettant une quantité  $q$  d'un flux `flow`.

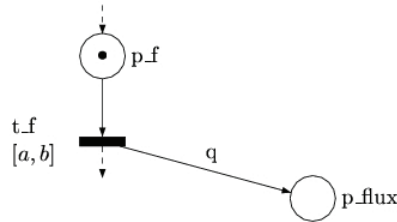


FIG. 2.6 – Fonction émettant un flux

### Réception de flux

Le motif d'attente d'un flux diffère selon la nature de la relation entre une fonction et son flux, en particulier selon la valeur du paramètre `isDestructive`. La figure 2.2.2 illustre le cas de deux fonctions F1 et F2, recevant toutes deux les flux `flowA` et `flowB`, selon les paramètres reportés dans le tableau ainsi que les quantités nécessitées. La fonction F2 est en outre affectée d'un `timeOut` d'une durée de 10 unités de temps.

	F1	F2
<code>flowA</code>	<code>isDestructive</code> « Acquire Available » $q = 2$	<code>isDestructive</code> normal $q = 3$
<code>flowB</code>	<code>!isDestructive</code> $q = 1$	<code>isDestructive</code> normal $q = 2$

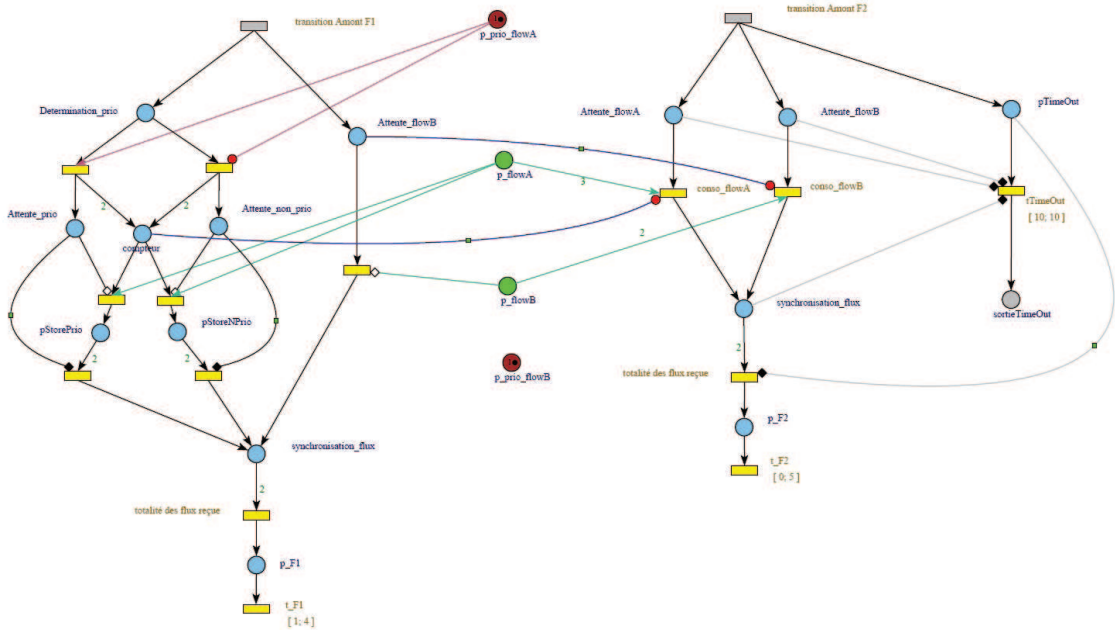
TAB. 2.1 – Paramètres des relations entre les fonctions et les flux de la figure 2.2.2

D'une manière générale, la lecture ou la consommation d'un flux se fait selon l'ordre de priorité suivant :

1. lecture du niveau de flux pour les fonctions non destructives (type trigger) ;
2. consommation de la ressource par la fonction « Acquire Available » prioritaire ;
3. consommation de la ressource par les autres fonctions « Acquire Available » ;
4. consommation de la ressource par les fonctions en mode d'attente normale ;

Cependant, une fonction non destructive et attendant un niveau  $q1$  d'un flux ne peut bloquer une autre fonction destructive attendant un niveau  $q2 < q1$  de ce même flux, sous peine de blocage du modèle.

On notera enfin que, sous cette modélisation, l'arrivée à échéance d'un `timeOut` conduit à l'abandon de la zone d'attente des flux ; les ressources qui ont éventuellement été prises avant cette échéance sont perdues.



### 2.2.3 Structures de contrôle

#### Structures d'exécutions parallèles

La figure 2.7 donne la traduction d'une structure par branches parallèles simple ; on notera à l'issue de chaque branche de la structure la présence d'une place permettant la synchronisation finale de toutes les branches entre elles.

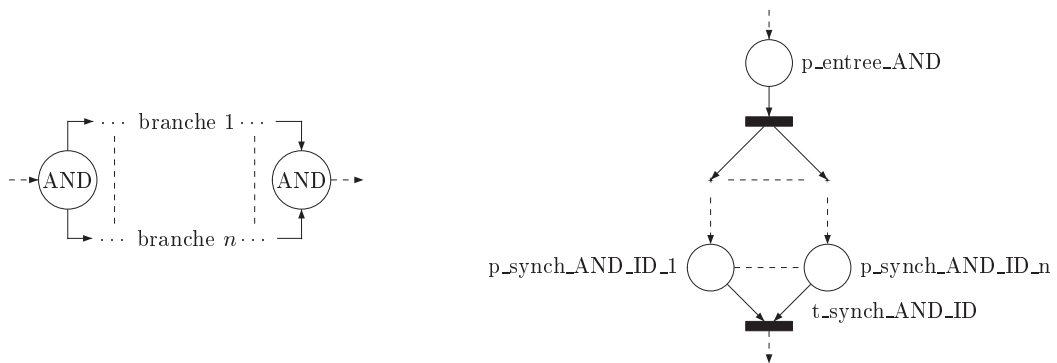


FIG. 2.7 – Branches parallèles et traduction en TPN

La traduction d'une structure de branches parallèles contenant au moins un modificateur kill est assez semblable au cas précédent. Elle est donnée figure 2.8 pour une seule branche de terminaison ; ce motif fait intervenir des arcs de vidange entre la transition `fin_branche_k` (modélisant la fin de la dernière construction de la branche de terminaison) et toutes les places des autres branches. Cette traduction est extensible au cas où plusieurs branches sont marquées du modificateur ; on ne la représente pas ici, pour des raisons de lisibilité.

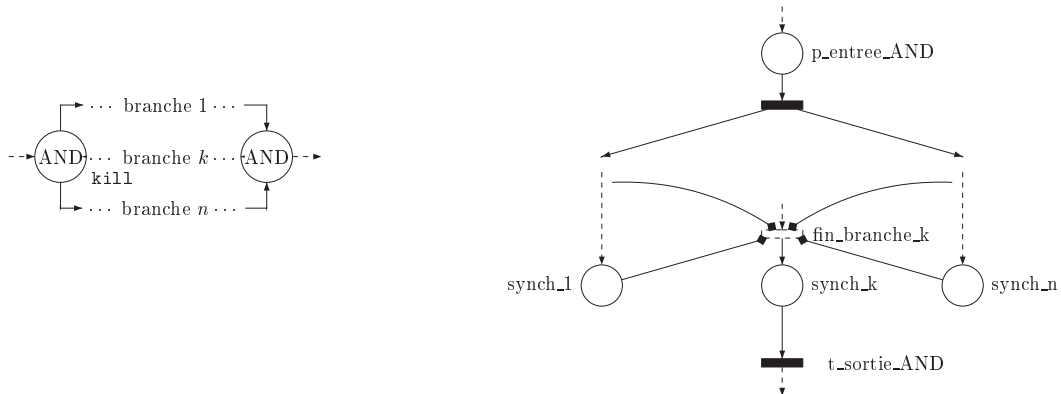


FIG. 2.8 – Branches parallèles avec une branche de terminaison

Enfin, nous avons montré que la structure de répliation pouvait se ramener à une structure par branches parallèles.

### Branches de choix

La traduction de la structure de sélection est assez immédiate ; elle est illustrée figure 2.9.

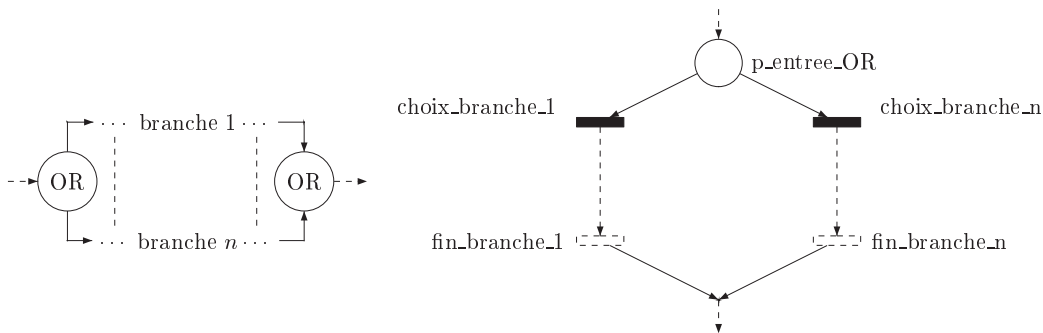


FIG. 2.9 – Branches de choix et traduction en TPN

Le choix de la branche à exécuter correspond au tir de l’une des transitions `choix_branche_i` dans la traduction du motif. Le choix de cette transition est externe au réseau, et fait partie des contrôles que l’utilisateur (utilisateur physique ou programme externe) aura à sa disposition lors de la simulation des superFFBD *via* leur modèle TPN correspondant (*thème de travail qui sera développé dès le début de notre thèse*).

### Itérations et boucles

La traduction en TPN de la structure d’itération est donnée figure 2.10. Elle met en œuvre deux compteurs :

- `counter_nl` (*no less*) : ce compteur permet d’effectuer *au moins*  $i$  itérations ;
- `counter_nm` (*no more*) : il permet d’effectuer *au plus*  $i$  itérations.

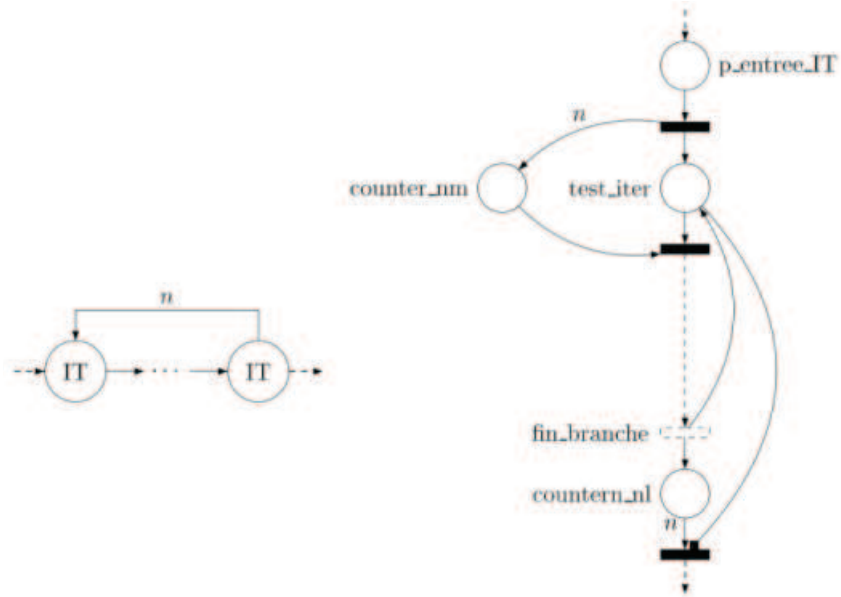


FIG. 2.10 – Structure d’itérations et traduction en TPN

Enfin, le motif correspondant à une boucle contenant un nœud LE est donné figure 2.11. L’arrivée dans la place  $p_{LE}$  et le tir de la transition  $t_{LE}$  vide la structure de ses jetons, se qui se traduit par l’ajout d’arcs de vidange entre toutes les places de la structure et la transition  $t_{LE}$ .

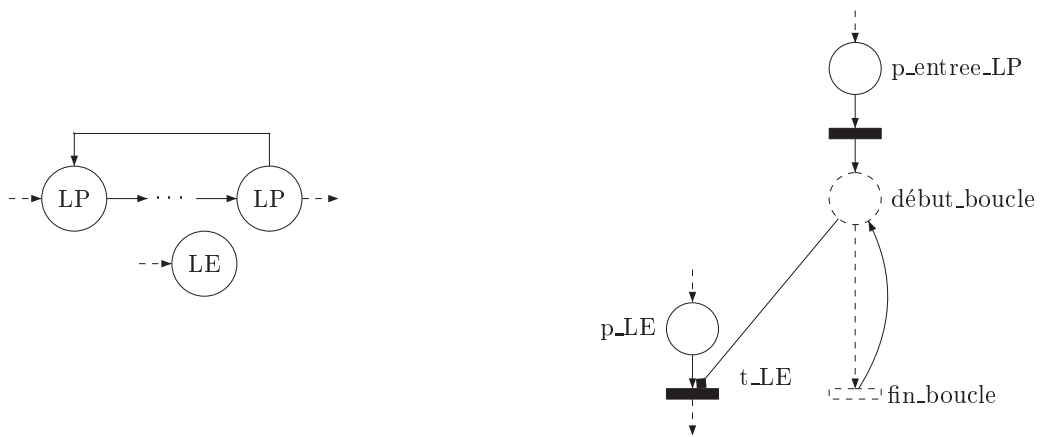


FIG. 2.11 – Structure de boucle avec nœud de sortie



## Chapitre 3

# Méta-modélisation et transformation de modèles

Les chapitres précédents ont permis de jeter les bases théoriques des traductions des modèles de haut-niveau vers les réseaux de Petri. Ce chapitre présente les principes qui ont guidé l'implémentation, sous forme de code Java, des traductions.

Nous y présentons ainsi les concepts d'architecture dirigée par les modèles (*Model Driven Architecture* ou MDA) et de transformation de modèles par l'écriture de règles de traduction sur leurs méta-modèles. Nous présentons ensuite les méta-modèles associés aux diagrammes superFFBD et aux réseaux de Petri tels qu'ils sont actuellement implémentés dans Roméo. Enfin, nous décrivons les principales classes Java qui ont permis la réalisation pratique d'un sous-ensemble de transformations des superFFBD vers les TPN.

### 3.1 Généralités

#### 3.1.1 Principes de la méta-modélisation

L'approche MDA, apparue en 2000, s'articule autour du paradigme selon lequel « tout est modèle » [Fra03, KWB03]. Elle se structure selon une architecture pyramidale en quatre niveaux d'abstraction croissants :

- **niveau M0** : le monde réel, comprenant les éléments que l'on cherche à modéliser ;
- **niveau M1** : les modèles, qui forment une représentation – simplifiée et donc souvent partielle – des objets du niveau M0 ;
- **niveau M2** : les méta-modèles, ou les « modèles de modèles », qui peuvent être vus comme des *langages de modèles*. Ils décrivent l'ensemble des règles et des propriétés régissant l'écriture d'un modèle, lequel est alors dit *conforme* à son méta-modèle ;
- **niveau M3** : les méta-méta-modèles, qui décrivent l'ensemble des règles et propriétés définissant l'écriture d'un méta-modèle ; à noter que le niveau M3 s'auto-définit et qu'il n'existe donc pas de niveau d'abstraction supérieur.

Considérons par analogie une navette en orbite autour de la Terre. Sa trajectoire réelle (niveau M0) est modélisée par des équations (M1), qui ont été écrites à partir de lois mathématiques et physiques (M2), elles-mêmes écrites en utilisant des symboles graphiques ou littéraux (M3).

La figure 3.1 illustre ces notions dans leur application au domaine de l'ingénierie logicielle. Le niveau M0 représente les instances du monde réel (des clients, des commandes,...), le niveau M1 la modélisation de ces instances sous formes de classes UML. Le niveau M2

décrit les « briques » de la modélisation par UML (classes UML, associations UML, attributs UML, etc. ) alors que le niveau M3 définit des concepts plus généraux, indépendants du langage utilisé au niveau M2.

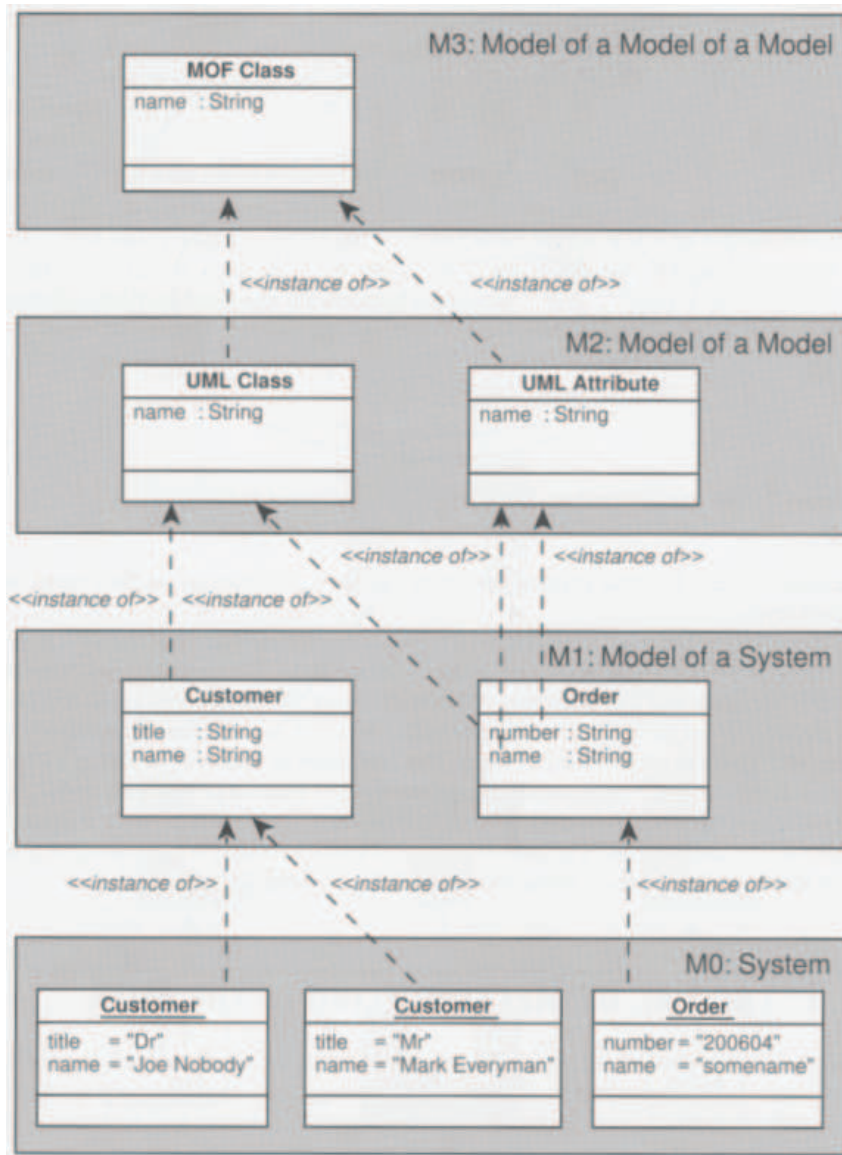


FIG. 3.1 – Modélisation d’une application informatique en niveaux d’abstraction

Plusieurs langages de niveau M3 ont été développés dans les domaines de l’ingénierie logicielle ou système ; les deux langages les plus communément utilisés sont :

- le *Meta Object Facility* (MOF), standard défini par l’OMG ;
- ECore, défini et employé avec le projet Eclipse EMF (*Eclipse Modeling Framework*), et qui s’appuie en partie sur la variante Essential MOF, développée par l’OMG.

C’est ce dernier langage qui a été utilisé pour la description et la manipulation des méta-modèles décrits plus bas.

### 3.1.2 Transformation et génération de modèles

L'intérêt majeur de l'approche MDA est de permettre une transformation « automatisée » de modèles en manipulant leurs méta-modèles. La transformation s'effectue par l'écriture de règles de transformation mettant en correspondance les concepts des deux méta-modèles. La figure 3.2 illustre le cas d'une transformation d'un modèle PIM (*Platform Independent Model*, décrit avec UML, par exemple), vers un modèle PSM (*Platform Specific Model*, écrit par exemple en C#).

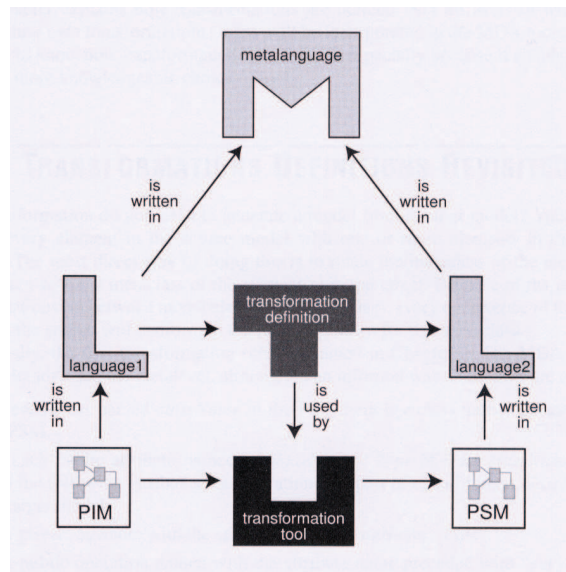


FIG. 3.2 – Transformation de modèles par l'approche MDA

Un certain nombre de langages dédiés à l'écriture de ces règles de transformation ont été développés ; citons en particulier QVT (*Queries/Views/Transformations*), standard défini par l'OMG, et ATL (*ATLAS Transformation Language*), développé à l'INRIA. Des outils logiciels ont également été créés pour la mise en œuvre de ces règles de transformation, comme l'ensemble MIA-Transformation – MIA-Generation <sup>1</sup> ou MDWorkbench <sup>2</sup>.

Dans un premier temps, nous avons choisi d'exprimer nos règles de transformation sous forme de classes Java ; une implémentation sous MDWorkbench devrait être prochainement à l'étude.

## 3.2 Description des méta-modèles

### 3.2.1 Méta-modèle d'entrée M2\_Process

Ce méta-modèle décrit les modèles superFFBD, objets `M1_Process` ; un module d'édition – encore en développement – permet de créer et de manipuler les objets du modèle en conception. À l'heure actuelle, ce méta-modèle est encore assez proche de celui offert par le logiciel CORE. Développé sous la forme de diagrammes UML, il s'articule autour de 11 *packages* principaux, 73 classes et 20 diagrammes de classes. Trois de ces diagrammes,

<sup>1</sup> Logiciels développés par Sodifrance.

<sup>2</sup> Logiciel développé par Sodus.

présentant un intérêt pour la traduction vers les TPN, sont détaillés plus bas ; pour des raisons de lisibilité, les classes ne portent généralement pas la totalité des attributs, méthodes et relations qui leur sont attachés.

Le but du travail présenté dans ce document n'était pas d'établir ce méta-modèle ; celui-ci a été construit de manière collégiale par les différentes équipes impliquées dans le projet Kimono, selon un processus itératif et en s'appuyant sur le document d'annexe au Dossier de Définition de Kimono. En revanche, un certain nombre de points soulevés durant l'étude des traductions vers les TPN ont permis d'affiner le méta-modèle `M2.Process`, en s'attachant à préciser la syntaxe et la sémantique de ses éléments. Nous avons pu par exemple ajouter à la classe décrivant les fonctions du modèle les attributs `durationMin` et `durationMax` pour préciser les durées d'exécution de ces fonctions (paramètres non disponibles sous CORE).

La figure 3.3 correspond au diagramme `Behavior Relations`, qui capture les relations entre les éléments décrivant la *hiérarchie* d'un modèle `M1.Process`. Celui-ci comprend un objet `Network` principal, entité contenant l'ensemble des branches et constructions du modèle. Il comprend en particulier une `NetworkBranch`, laquelle contient la *liste ordonnée* des constructions (objets `NetworkConstruct`) qui se succèdent sur cette branche.

La classe `NetworkConstruct` est de type abstrait car regroupant tous les types de constructions possibles ; la figure 3.4, correspondant au diagramme `Behavior Type Hierarchy`, illustre les différentes constructions héritant de ce type abstrait. De plus, chaque construction se compose d'un certain nombre de branches (une pour les structures d'itération et de boucle, deux pour les réplifications et  $n$  pour les structures de fonction<sup>1</sup>, de choix ou les structures parallèles).

Enfin, les relations entre les flux (classe `Flow`) et les fonctions (classe `Function`) sont modélisées comme représenté par le diagramme `Functions and Flows`, illustré figure 3.5. Elles font apparaître des classes intermédiaires, `TriggerByTriggersRelation`, `InputsInputsToRelation` et `OutputsOutputsFromRelation`. Ce sont ces classes qui permettent de caractériser la nature de la relation entre un flux et la fonction qui le produit ou le reçoit, comme présenté partie 1. Ces classes permettent également de préciser la quantité de flux manipulée.

Comme évoqué plus haut, le méta-modèle (`M2.Process`) est encore en évolution. Ainsi, un certain nombre de structures dont la sémantique a été donnée plus haut n'y figurent pas encore ; il s'agit en particulier des fonctions périodiques et des triggers de terminaison. En revanche, le méta-modèle mentionne l'existence d'un attribut `loopCondition` dans la classe `LoopConstruct`. Cet attribut n'est pas encore pris en compte dans l'éditeur `M1.Process` mais sera vraisemblablement utilisé par la suite pour attacher à une structure de boucle une condition d'arrêt.

En ce qui concerne l'implémentation du méta-modèle, elle s'est effectuée en plusieurs étapes :

- export du méta-modèle UML vers un document XMI (*XML Metadata Interchange*) ;
- transformation au format ECore<sup>2</sup> ;
- génération, avec les outils EMF, de l'ensemble des packages, interfaces et classes Java permettant la manipulation des entités du méta-modèle.

<sup>1</sup> Dans ce cas,  $n = 0$  signifie que la fonction n'est pas à sorties multiples.

<sup>2</sup> À l'heure actuelle, cette transformation se fait avec MDWorkbench.

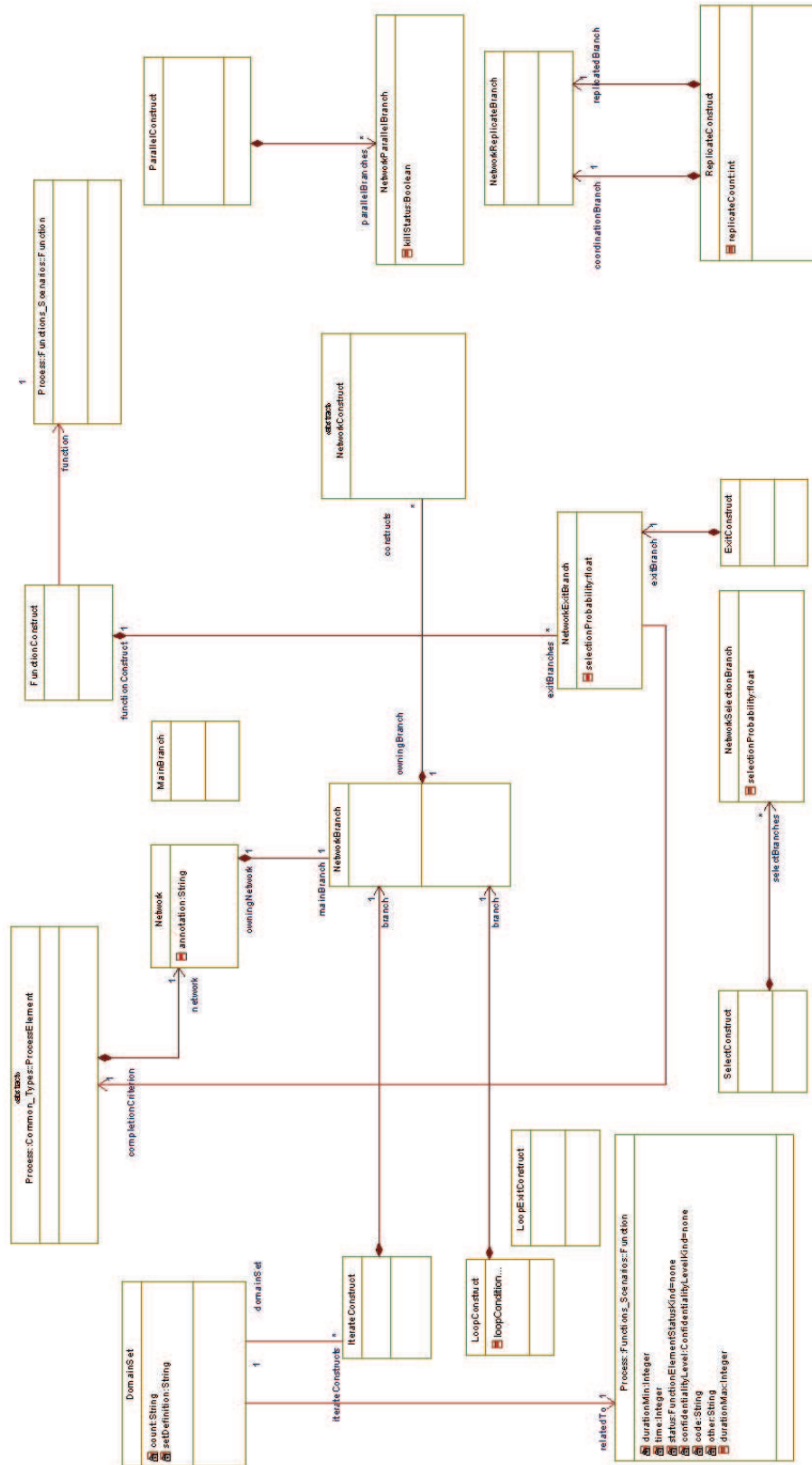


FIG. 3.3 – Diagramme Behavior Relations (M2\_Process)

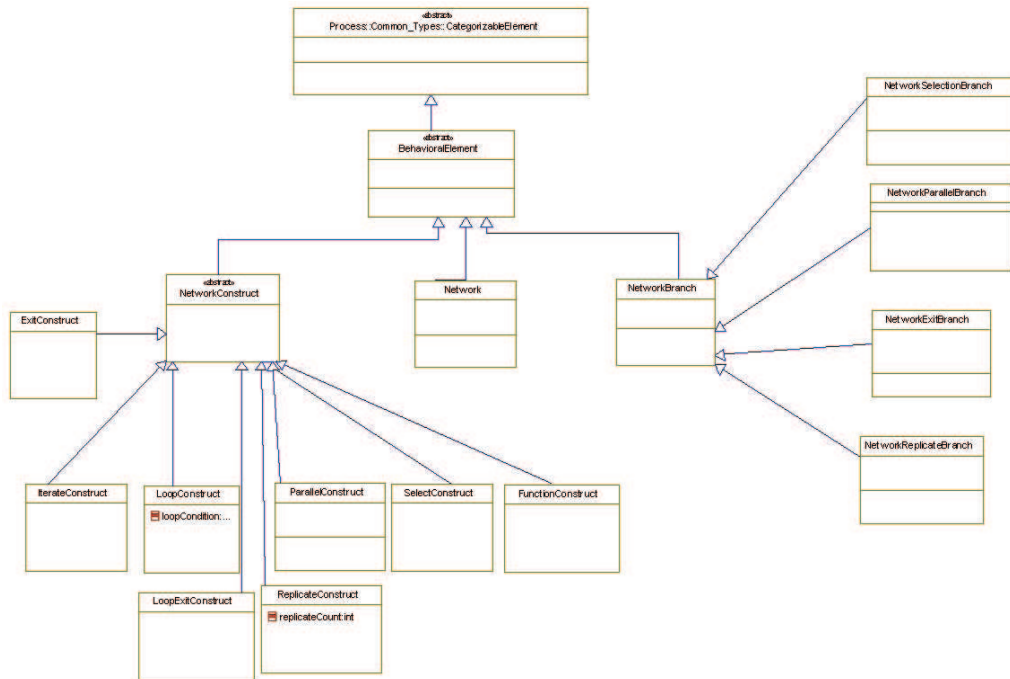


FIG. 3.4 – Diagramme Behavior Type Hierarchy (M2.Process)

### 3.2.2 Méta-modèle de sortie M2.Romeo

Plusieurs solutions se sont présentées pour la construction du méta-modèle de sortie M2.Romeo et la génération des objets Java correspondants :

- construction « manuelle » d'un méta-modèle au format UML puis transformation vers ECore (comme pour le méta-modèle d'entrée) ;
- transformation vers ECore du fichier de DTD<sup>1</sup> `romeo.dtd` donnant la syntaxe des fichiers XML interprétables par Roméo (cette transformation s'effectuant de manière automatique *via* les outils EMF) ;
- transformation du fichier `romeo.dtd` en fichier XSD<sup>2</sup> puis transformation par les outils EMF vers ECore.

La première solution a rapidement été éliminée car, bien que le méta-modèle de Roméo (présenté figure 3.6), soit d'une complexité bien inférieure au méta-modèle M2.Process, il nous a paru plus efficace d'utiliser d'une part le document DTD, qui constitue la référence du point d'entrée des modules de calculs de Roméo, et d'autre part les outils permettant d'automatiser la construction du fichier ECore à partir de cette description.

Par ailleurs, l'utilisation « brute » du fichier de DTD (seconde solution) ne permet pas de spécifier les *types* des attributs des éléments qui y sont décrits. Il nous est donc apparu plus pertinent de transformer dans un premier temps le document DTD en fichier XSD pour ensuite effectuer la transformation vers ECore (troisième solution). Cette manipulation intermédiaire a ainsi permis de typer les attributs des éléments du méta-modèle, offrant ainsi une gestion plus « propre » en terme d'objets Java. Le fichier XSD résultant est fourni en annexe, p.43.

<sup>1</sup> Document Type Definition.

<sup>2</sup> XML Schema Description.

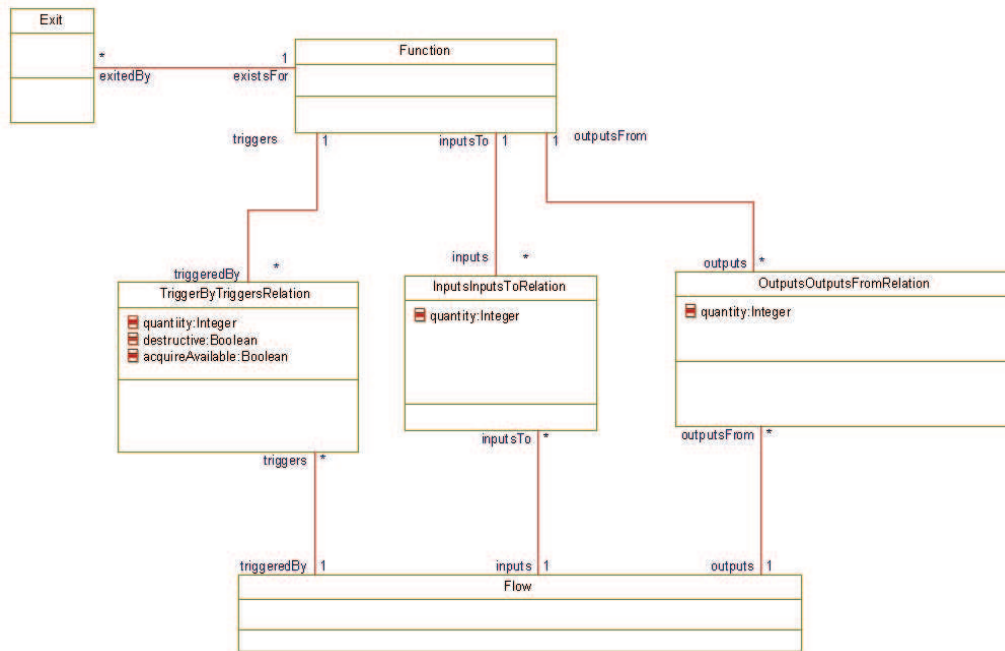


FIG. 3.5 – Diagramme Functions and Flows (M2\_Process)

Le méta-modèle met en jeu quatre classes principales :

- classe TPN : il s’agit de l’élément de plus haut niveau, contenant tous les autres ;
- classe Place ;
- classe Transition ;
- classe Arc ;

On notera que la classe Arc possède un attribut `type`, dont la valeur est un élément de l’énumération `ArcType` et qui reprend les cinq types d’arcs implémentés dans Roméo à ce jour.

Le méta-modèle décrit également les classes permettant de gérer la disposition graphique des éléments du réseau. Pour des raisons de simplicité, nous avons choisi de ne pas traiter ces aspects dans nos traductions.

Dans un premier temps, nous nous sommes attachés à produire un outil capable de fournir un fichier interprétable par Roméo, *i.e.* un fichier XML conforme au document `romeo.dtd`, et donc de limiter la communication entre les modèles dans le sens superFFBD  $\rightarrow$  Roméo. À ce titre, il a semblé plus intéressant de définir « manuellement » les classes et méthodes Java réalisant l’écriture dans un fichier XML du modèle TPN issu de la transformation. Il pourra bien entendu être intéressant de mettre en œuvre par la suite des techniques de génération automatisée de modules lecteur/écrivain de modèles TPN.

Le code de la classe `RomeoXMLWriter.java`, réalisant l’écriture du fichier XML, est donné en annexe p.47. On notera qu’il est assez similaire au code Tcl/Tk de Roméo, permettant l’écriture du fichier après son édition dans l’interface graphique.

### 3.3 Écriture des règles de transformation

Les règles de transformation ont été implémentées sous la forme de classes Java que, pour des raisons de lisibilité, nous avons choisi de ne pas reproduire dans ce document.

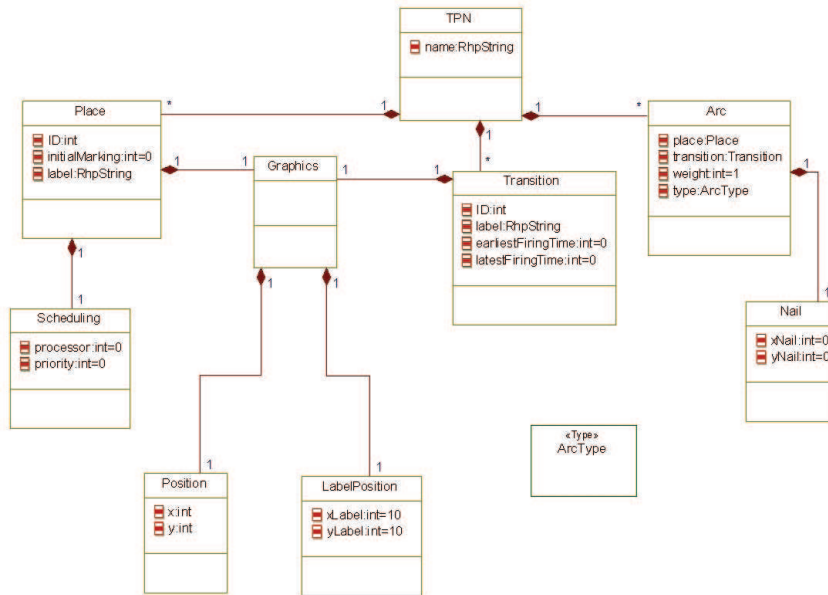


FIG. 3.6 – Méta-modèle M2\_Romeo

Une intégration à un logiciel tel que MIA ou MDWorkbench sera très certainement réalisée par la suite.

Le principe général suivi pour la transformation d'un modèle superFFBD s'articule en 5 étapes :

1. chargement du modèle d'entrée et de son scénario de plus haut niveau hiérarchique ;
2. construction des motifs de flux ;
3. parcours itératif de la branche principale du scénario chargé et appel de la méthode réalisant le motif correspondant à la construction rencontrée ;
4. ajout des arcs de vidange et d'inhibition logique entre les motifs créés à l'étape précédente ;
5. écriture dans un fichier XML du réseau obtenu.

Les opérations réalisées à l'étape 4 permettent en particulier de gérer les priorités entre les fonctions accédant à un même flux.

À l'heure actuelle, les constructions suivantes sont entièrement implémentées :

- branches parallèles (cf. algorithme donné annexe C) ;
- répliquations ;
- structures de choix ;
- itérations ;
- boucles (infinies) ;
- fonctions sans scénario de décomposition ni sorties multiples mais pouvant recevoir ou émettre des flux de nature quelconque ;

Un exemple de réseau obtenu par l'utilisation des programmes Java est donné au chapitre suivant.

# Chapitre 4

## Résultats

Nous présentons dans cette partie quelques résultats théoriques et pratiques sur les transformations présentées dans les chapitres précédents.

### 4.1 Résultats théoriques

Un objectif majeur de ce stage pratique étant de produire un outil opérationnel de traduction des superFFBD vers les TPN, nous avons volontairement laissé de côté un certain nombre de problèmes théoriques, en particulier pour ce qui concerne la décidabilité de propriétés telles que la bornitude ou l'accessibilité de marquages sur les modèles issus des transformations. Ces thématiques seront très vraisemblablement développées dès le début du travail de thèse.

Néanmoins, deux conjectures ont pu être formulées ; une idée de preuve est donnée pour la seconde.

**Conjecture 1** [bornitude des TPN obtenus] *La traduction d'un modèle superFFBD ne comportant pas de motif de flux conduit à un TPN borné.*

**Conjecture 2** [non multi-sensibilisation des constructions] *En l'absence de fonctions périodiques dans le modèle, une construction ne peut être ré-entrée par un flux de contrôle tant que cette structure est active (i.e. qu'un flux de contrôle s'y trouve déjà).*

**Idée de preuve de la conjecture 2 :** de par la nature fortement séquentielle des diagrammes de type EFFBD, le flux de contrôle est généralement unique à un instant donné, mis à part dans les constructions avec branches parallèles (AND). Dans ce dernier cas, cependant, les propriétés d'imbrications strictes des structures de contrôle interdisent que deux flux de contrôle indépendants (i.e. situés sur deux branches parallèles distinctes) puissent activer une même construction.

### 4.2 Résultats pratiques

L'éditeur de modèles `M1_Process` n'étant pas encore disponible à l'issue du stage, il ne nous a pas été possible de tester de manière efficace les traductions et leurs implémentations. Cependant, un exemple assez simple de diagramme superFFBD a été construit « manuellement » à partir des classes Java implémentant le méta-modèle `M2_Process`. Ce diagramme est illustré figure 4.1. Il fait intervenir en particulier une construction par

branches parallèles, une structure d'itération et un échange de flux (non destructif) entre deux fonctions. Le choix des durées d'exécution des fonctions, décrites tableau 4.1, a été fait de manière arbitraire et ne correspond pas à un système particulier.

Fonction	Intervalle	Fonction	Intervalle
f1	[1; 4]	f4	[0; 5]
f2	[2; 2]	f5	[1; ∞[
f3	[1; 4]	f6	[4; 5]

TAB. 4.1 – Durées d'exécution des fonctions

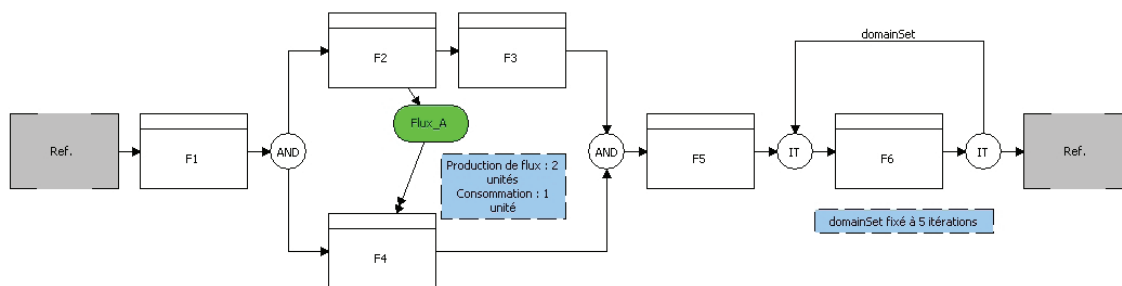


FIG. 4.1 – Exemple de diagramme superFFBD

Le réseau issu de la traduction est présenté figure 4.2. Comme précisé dans le chapitre précédent, la disposition graphique des éléments a été réalisée manuellement. Les places correspondant à des exécutions de fonctions (places  $p_{fX}$ ) sont sur fond brun, les places modélisant le flux  $flux_A$  sont sur fond vert et les places initiale et finale sont sur fond gris.

D'une manière générale, le modèle de sortie possède une structure similaire au modèle d'entrée. Aux 11 structures (*en incluant le flux*) du modèle d'entrée correspondent 19 places, 14 transitions et 36 arcs. Il semble ainsi raisonnable de supposer que la taille des réseaux obtenus reste de l'ordre de celle des modèles d'entrée, en terme de nombre de constructions.





# Conclusion

Nous avons présenté dans ce mémoire du stage pratique de Master les principes et méthodes qui nous ont permis de traduire des modèles de haut-niveau, exprimés dans un formalisme issu du monde de l'ingénierie des systèmes, vers les réseaux de Petri temporels. L'utilisation de la méthode MDA et la transformation des modèles *via* leur méta-modèle ont ainsi conduit à la réalisation d'un premier outil, proposé actuellement sous forme de classes Java, et qui devrait prochainement être intégré à l'application Kimono.

Le travail décrit dans ce document n'a pas la prétention d'être exhaustif et laisse de nombreuses pistes de réflexions et d'études pour les travaux qui seront menés au cours de notre thèse industrielle avec l'IRCCyN et la société Sodius.

Ainsi, un certain nombre de points concernant l'implémentation des règles de transformation restent à traiter : outre l'ajout des structures dont la traduction a été donnée au chapitre 2 mais qui ne sont pas encore disponibles dans l'outil, il est nécessaire d'assurer une certaine « robustification » du code créé (notamment par l'emploi de structures de gestion des exceptions), ce qui n'a été que partiellement réalisé jusqu'ici. Nous rappelons également que la validation du code est en grande partie subordonnée à la réalisation d'une série de tests, ce qui sera possible après réception de l'éditeur de modèles superFFBD.

Enfin, une grande partie de nos efforts dans les travaux futurs seront orientés vers l'étude théorique et la réalisation pratique d'outils de simulation et de vérification des modèles superFFBD au moyen de leur traduction en réseaux de Petri temporels. En particulier, la vérification pourra s'effectuer en exprimant des propriétés de *sûreté* (« *Un état mauvais n'est jamais atteint* ») ou de *vivacité* (« *Un état bon finira toujours par être atteint* »), au moyen d'une logique temporelle quantitative dédiée, TPN-TCTL<sup>1</sup>, implémentée dans les modules de calcul de Roméo [Gar06].

---

<sup>1</sup> *Temporal Computational Tree Logic for TPN.*



## Annexe A

# Fichier romeo.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Romeo XSD Specification of Time Petri Nets -->
<!-- Version: 1.0 -->
<!-- Author : C. Seidner -->
<!-- Created from the romeo.dtd file , v0.0 beta -->

<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  xmlns:romeo="http://romeo.rts-software.org"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://romeo.rts-software.org">

  <!-- Definition of a TIME PETRI NET -->
  <xsd:complexType name="TPN">
    <xsd:sequence>
      <xsd:element ecore:reference="romeo:Place" maxOccurs="unbounded"
        minOccurs="0" name="place" type="xsd:anyURI"/>
      <xsd:element ecore:reference="romeo:Transition"
        maxOccurs="unbounded" minOccurs="0" name="transition"
        type="xsd:anyURI"/>
      <xsd:element ecore:reference="romeo:Arc" maxOccurs="unbounded"
        minOccurs="0" name="arc" type="xsd:anyURI"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>

  <!-- Definition of a PLACE -->
  <xsd:complexType name="Place">
    <xsd:sequence>
      <xsd:element ecore:reference="romeo:Scheduling" name="scheduling"
        type="xsd:anyURI"/>
      <xsd:element ecore:reference="romeo:Graphics" name="graphics"
        type="xsd:anyURI"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:int"/>
    <xsd:attribute name="label" type="xsd:string"/>
    <xsd:attribute name="initialMarking" type="xsd:int"/>
  </xsd:complexType>

  <!-- Definition of a TRANSITION -->
```

```

<xsd:complexType name="Transition">
  <xsd:sequence>
    <xsd:element ecore:reference="romeo:Graphics" name="graphics"
      type="xsd:anyURI"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:int"/>
  <xsd:attribute name="label" type="xsd:string"/>
  <xsd:attribute name="eft" type="xsd:int"/>
  <xsd:attribute name="lft" type="xsd:int"/>
  <!-- Rem: Latest Firing Time can be infinite; this will be coded-->
  <!-- by (-1) in the metamodels and coded by "infini" in the -->
  <!-- TPN/XML writer-->
</xsd:complexType>

<!-- Definition of an ARC -->
<xsd:complexType name="Arc">
  <xsd:sequence>
    <xsd:element ecore:reference="romeo:Nail" name="nail"
      type="xsd:anyURI"/>
  </xsd:sequence>
  <xsd:attribute ecore:reference="romeo:Place" name="place"
    type="xsd:anyURI"/>
  <xsd:attribute ecore:reference="romeo:Transition" name="transition"
    type="xsd:anyURI"/>
  <xsd:attribute ecore:reference="romeo:ArcType" name="type"
    type="xsd:anyURI"/>
  <xsd:attribute name="weight" type="xsd:int"/>
</xsd:complexType>

<!-- Definition of the SCHEDULING parameters -->
<xsd:complexType name="Scheduling">
  <xsd:attribute name="gamma" type="xsd:int"/>
  <xsd:attribute name="omega" type="xsd:int"/>
</xsd:complexType>

<!-- Definition of the GRAPHICS informations -->
<xsd:complexType name="Graphics">
  <xsd:sequence>
    <xsd:element ecore:reference="romeo:Position" name="position"
      type="xsd:anyURI"/>
    <xsd:element ecore:reference="romeo:DeltaLabel" name="deltaLabel"
      type="xsd:anyURI"/>
  </xsd:sequence>
</xsd:complexType>

<!-- X-Y coordinates of element -->
<xsd:complexType name="Position">
  <xsd:attribute name="x" type="xsd:double"/>
  <xsd:attribute name="y" type="xsd:double"/>
</xsd:complexType>

<!-- X-Y coordinates of an element's label -->
<xsd:complexType name="DeltaLabel">
  <xsd:attribute name="deltax" type="xsd:double"/>

```

```

    <xsd:attribute name="deltay" type="xsd:double"/>
  </xsd:complexType>

  <!-- X-Y coordinates of an arc nail -->
  <xsd:complexType name="Nail">
    <xsd:attribute name="xnail" type="xsd:double"/>
    <xsd:attribute name="ynail" type="xsd:double"/>
  </xsd:complexType>

  <!-- Definition of ArcType (string enumeration) -->
  <xsd:simpleType name="ArcType">
    <xsd:restriction base="xsd:string">
      <!-- Rem: the string values are exactly those used by Romeo -->
      <!-- so as to avoid another conversion-->
      <xsd:enumeration value="PlaceTransition"/>
      <xsd:enumeration value="TransitionPlace"/>
      <xsd:enumeration value="flush"/>
      <xsd:enumeration value="read"/>
      <xsd:enumeration value="logicalInhibitor"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>

```



## Annexe B

# Fichier RomeoXMLWriter.java

```
package org.software.rts.romeo.io;

import java.io.*;
import java.util.Iterator;

import org.software.rts.romeo.*;

public class RomeoXMLWriter {

    static PrintWriter printWriter=null;

    public static void writeTPN(TPN tpn, File file){
        // Main method for writing an XML file
        // Convert places, transitions and arc in the TPN using
        // the appropriate writeXXX methods
        try{
            printWriter=new PrintWriter(new FileWriter(file, false));
        } catch (IOException e){}

        /* Write prolog*/
        printWriter.println("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
        printWriter.println("<!-- Created from a superFFBD model -->");
        printWriter.println("<TPN name=\"" +tpn.getName()+"\">");

        /*Parse the tpn and write the places, transitions and arcs*/
        for (Iterator iter = tpn.getPlace().iterator(); iter.hasNext();) {
            Place place = (Place) iter.next();
            writePlace(place);
        }
        for (Iterator iter=tpn.getTransition().iterator(); iter.hasNext();){
            Transition transition = (Transition) iter.next();
            writeTransition(transition);
        }
        for (Iterator iter = tpn.getArc().iterator(); iter.hasNext();) {
            Arc arc = (Arc) iter.next();
            writeArc(arc);
        }

        /* Write epilog and close file*/
        printWriter.println("</TPN>");
        printWriter.close();
    }
}
```

```

/*Place writer*/
private static void writePlace(Place place){
    printWriter.println("\t<place id=\"" + place.getId() +
        "\" label=\"" + place.getLabel() + "\" initialMarking=\"" +
        place.getInitialMarking() + "\">");
    printWriter.println("\t\t<graphics>");
    printWriter.println("\t\t\t<position x=\"" +
        place.getGraphics().getPosition().getX() +
        "\" y=\"" + place.getGraphics().getPosition().getY() +
        "\" />");
    printWriter.println("\t\t\t<deltaLabel deltax=\"" +
        place.getGraphics().getDeltaLabel().getDeltax() +
        "\" deltay=\"" +
        place.getGraphics().getDeltaLabel().getDeltay() +
        "\" />");
    printWriter.println("\t\t</graphics>");
    printWriter.println("\t\t<scheduling gamma=\"" + "0" + " omega=\"" + "0" + " /> ");
    printWriter.println("\t</place>");
}

/*Transition writer (with translation of infinite
latest firing time from -1 to "infini")*/
private static void writeTransition(Transition transition){
    String stringLft;

    if (transition.getLft() == -1) {stringLft = "infini";
    } else {stringLft = Integer.toString(transition.getLft());}

    printWriter.println("\t<transition id=\"" + transition.getId() +
        "\" label=\"" + transition.getLabel() + "\" eft=\"" +
        transition.getEft() + "\" lft=\"" + stringLft + "\">");
    printWriter.println("\t\t<graphics>");
    printWriter.println("\t\t\t<position x=\"" +
        transition.getGraphics().getPosition().getX() +
        "\" y=\"" + transition.getGraphics().getPosition().getY() +
        "\" />");
    printWriter.println("\t\t\t<deltaLabel deltax=\"" +
        transition.getGraphics().getDeltaLabel().getDeltax() +
        "\" deltay=\"" +
        transition.getGraphics().getDeltaLabel().getDeltay() +
        "\" />");
    printWriter.println("\t\t</graphics>");
    printWriter.println("\t</transition>");
}

/*Arc writer*/
private static void writeArc(Arc arc){
    printWriter.println("\t<arc place=\"" + arc.getPlace().getId() +
        "\" transition=\"" + arc.getTransition().getId() +
        "\" type=\"" + arc.getType().getName() + "\" weight=\"" +
        arc.getWeight() + "\">");
    printWriter.println("\t\t<nail xnaill=\"" +
        arc.getNail().getXnaill() + "\" ynaill=\"" +
        arc.getNail().getYnaill() + "\" />");
    printWriter.println("\t</arc>");
}
}
}

```

## Annexe C

# Algorithme de transformation des structures parallèles

À titre d'illustration, on donne dans cette partie l'algorithme mis en œuvre pour la traduction des motifs de branches parallèles sans modificateur `kill`. On suppose disposer de fonctions permettant d'ajouter au réseau en cours de construction des places, des transitions et des arcs. Les paramètres de ces fonctions correspondent aux attributs précisés dans le méta-modèle (pour des raisons de simplicité, les paramètres graphiques et ceux concernant l'ordonnancement ne sont pas repris ici) :

- `createPlace(label, initialMarking)`;
- `createTransition(label, earliestFiringTime, latestFiringTime)`;
- `createArc(place, transition, type, weight)`;

Par ailleurs, on dispose pour la structure de la liste ordonnée de ses branches et d'une fonction effectuant la traduction d'une telle branche (cette fonction renvoie la dernière transition créée, afin de connecter le motif de la branche à la sortie de la structure parallèle).

```
pEntering ← createPlace("p_AND_in",0);
arc1 ← createArc(pEntering, transitionAmont, Transition2Place,1);
tEntering ← createTransition("t_AND_in",0,0);
arc2 ← createArc(pEntering,tEntering,Place2Transition,1);
tExiting lbl ← createTransition("t_AND_out",0,0);
for all parallelBranch ∈ parallelBranchesList do
  lastTransition ← createParallelBranch(parallelBranch);
  pSynchro ← createPlace("p_Synch",0);
  arc3 ← createArc(pSynchro, lastTransition,Transition2Place,1);
  arc4 ← createArc(pSynchro,tExiting,Place2Transition,1);
end for
```



# Bibliographie

- [AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.
- [Fra03] David S. Frankel. *Model Driven Architecture : Applying MDA to Enterprise Computing*. John Wiley & Sons, OMG Press, 2003.
- [Gar06] G. Gardey. *Contribution à la vérification et au contrôle des systèmes temps réel – Application aux réseaux de Petri temporels et aux automates temporisés*. PhD thesis, École Centrale Nantes – Université de Nantes, 2006.
- [GLMR05] G. Gardey, D. Lime, M. Magnin, and O.H. Roux. Romeo : A tool for analyzing time Petri nets. In *Proceedings of the 17<sup>th</sup> International Conference on Computer-Aided Verification (CAV’05)*, Lecture Notes in Computer Science, 2005.
- [Her04] E. Herzog. *An approach to Systems Engineering tools data representation and exchange*. Linköpings Universitet, 2004.
- [IEE94] IEEE. IEEE-1220 : Application & management of systems engineering process, 1994.
- [Jen87] K. Jensen. Coloured Petri nets. In *Advances in Petri nets 1986, part I on Petri nets : central models and their properties*, pages 248–299. Springer-Verlag, 1987.
- [KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [Lon95] J. Long. Relationships between common graphical representations in System Engineering. In *5<sup>th</sup> International Symposium of the INCOSE*, 1995. *Mise à jour juillet 2002*.
- [Mer74] P. Merlin. *A study of recoverability of computing systems*. PhD thesis, Dpt. of Computer Science, University of California, Irvine, CA, 1974.
- [Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, 1962.