# Contribution à la formalisation et à la vérification des diagrammes dynamiques UML2 à base des réseaux de Petri

Aymen Louati

Université Tunis EL Manar
Ecole Nationale d'Ingénieurs de Tunis
Laboratoire du Signal, Images et Technologies de l'Information

École Doctorale d'Informatique, Télécommunications et Électronique de Paris
Conservatoire National des Arts et Métiers
Centre des Etudes et De Recherche en Informatique et Communication (CEDRIC)

# Contribution à la Formalisation et à la Vérification des Diagrammes Dynamiques UML2 à base des Réseaux de Petri

**THÈSE**

*Pour obtenir le grade de* : **Docteur Conservatoire National des Arts et Métiers de Paris et de l'Ecole Nationale d'Ingénieurs de Tunis**

*Spécialité* : **Informatique (CNAM Paris) et Génie-Electrique (ENIT Tunis)**

*Présentée et soutenue publiquement par :*

**Aymen LOUATI**

**Le 09 Décembre 2015**

**Devant le jury composé de :**

| | |
|---|---|
| Mme. Safya BELGHITH | Professeur, ENIT Tunis (**Présidente**) |
| Mr. Kamel BARKAOUI | Professeur, CNAM Paris (**Directeur de thèse**) |
| Mr. Noureddine ELLOUZE | Professeur, ENIT Tunis (**Directeur de thèse**) |
| Mme. Amel GRISSA-TOUZI | Maître de Conférences, ENIT Tunis (**Rapporteur**) |
| Mme. Thouraya BOUABANA-TEBIBEL | Professeur, ESI Alger (**Rapporteur**) |
| Mme. Isabelle COMYN-WATTIAU | Professeur, CNAM Paris (**Examinateur**) |

*A mes parents, à qui je dois tout.*
*A ma chère femme Abir, à qui je dois beaucoup.*
*A mon frère et son épouse, à ma soeur et aux mes chers petits Malek et Salim.*
*Aux mémoires de mon grand-père et mon oncle.*

*"Démarrer une thèse, c'est comme escalader une très haute montagne, sans pouvoir autant retourner en arrière."*
**Aymen LOUATI**

# Remerciements

Les résultats présentés dans ce mémoire sont l'aboutissement des travaux de recherche effectués dans le cadre d'une thèse en cotutelle entre le Laboratoire du Signal, Images et Technologies de l'Information, de l'Ecole Nationale d'Ingénieurs de Tunis et le Laboratoire CEDRIC du Conservatoire National des Arts et Métiers de Paris.

Je tiens à exprimer mes sincères remerciements à Mme. Safya BELGHITH, Professeur à l'ENIT, pour m'avoir fait l'honneur de présider le jury de ma soutenance. Qu'elle trouve ici le témoignage de mon profond respect.

Je suis très reconnaissant envers Mme. Thouraya BOUABANA-TEBIBEL, Professeur à l'ESI d'Alger, qui m'a honoré en acceptant de juger ce travail, pour l'analyse rigoureuse qu'elle a mené à ce mémoire et qui a contribué à son enrichissement. Je tiens de même à exprimer mes vifs remerciements envers Mme. Amel GRISSA-TOUZI, Maître de conférences à l'ENIT, de l'honneur qu'elle me fait en acceptant de juger mes travaux de recherche.

Je remercie Mme. Isabelle COMYN-WATTIAU, Professeur au CNAM Paris, d'avoir accepté de faire partie des membres du jury.

Je tiens à exprimer ma profonde gratitude et mes sincères remerciements à mes deux chers directeurs de thèse, Mr. Noureddine ELLOUZE, Professeur à l'ENIT et Mr. Kamel BARKAOUI, Professeur au CNAM Paris, à qui je dois beaucoup. C'était un plaisir de partager avec vous ces dernières années, vous m'avez appris le sens de la responsabilité et de la patience, vous m'avez surtout soutenu au moment du besoin. Merci pour votre générosité et votre confiance.

J'adresse mes remerciements à Mme. Chadlia JERAD, Maître-assistante à l'ENSI Tunis, pour sa disponibilité et sa collaboration qu'elle n'a cessé de me fournir dans le cadre de ces travaux. Il me tient tout particulièrement à cœur de remercier mes collègues de l'école Mahjoub et Zohra pour leur solidarité. Je remercie spécialement mon frère éternel Hechmi ainsi que ma collègue Rimah pour son enthousiasme.

# Résumé

Les systèmes informatiques envahissent de plus en plus notre quotidien, en allant de la plus simple application de lecture des fichiers audio, à la plus critique comme les voitures et les avions. Dans les systèmes critiques, la validation par vérification formelle s'impose. Cette thèse s'inscrit dans ce cadre et tend à doter le langage UML, langage de modélisation standard de facto, d'une sémantique formelle pour des finalités de vérification. En premier lieu, nous avons analysé et révisé le fondement théorique des principales approches de vérification formelle issues de la littérature et se focalisant sur le langage UML, ses profils et les concepts des réseaux de Petri (RdP). En deuxième lieu, nous avons proposé une nouvelle approche hiérarchique de formalisation pour la vérification des diagrammes globaux d'interaction (IOD). Ensuite, nous avons développé des formalismes temporels et temporisés des diagrammes de Timing UML2 (TD), appliqués par des exemples d'illustration. Sur ceux, nous avons conçu une nouvelle approche hiérarchique, s'intéressant aux Systèmes Temps Réel (STR), utilisant l'extension temporelle du langage des contraintes objets OCL/Temps Réel ($OCL^{TR}$), le profil UML MARTE et la logique temporelle temporisée (TCTL), exploitée d'une technique de vérification automatique après la transformation du modèle (Model Checking). Enfin, nous avons appliqué les formalismes proposés sur une étude de cas, afin de garantir leur efficacité logique et temporelle.

**Mots clés :** UML, MARTE, RdP, STR, IOD, TD, $OCL^{TR}$, TCTL, Model Checking.

# Abstract

The computer systems are increasingly invading our daily lives from the simplest application as audio files reading to the most critical one as cars and airplanes. For the critical systems, the validation by the formal verification is required. This thesis concerns this area of research and aims to ensure the betterment of UML language, which is the de facto standard, with formal semantics for verification finality. For the first part, we have analyzed and revised the theoretical foundations the existing formal verification approaches used UML, their profiles and the basic concepts of the Petri Nets (PN). For the second part, we have created a novel hierarchical approach of formalization in order to verify the Interaction Overview Diagrams (IOD). Then, we have developed temporal formalisms based on the UML2 Timing Diagrams (TD), applied by illustration examples. Based on these, we have proposed a novel hierarchical approach which is interested in Real Time Systems (RTS) and employed the temporal extension of the Object Constraints Language (OCL/Real Time) ($OCL^{TR}$), the UML MARTE profile and the timed computation Tree logic (TCTL), given by the Model Checking technique after the model's transformation. Finally, we have applied all the proposed formalisms using a case study, in order to ensure its logical and temporal efficiency.

**Keywords :** UML, MARTE, PN, RTS, IOD, TD, $OCL^{TR}$, TCTL, Model Checking.

# Table des matières

# Liste des tableaux

# Table des figures

# Introduction générale

**Sommaire**



Le Génie Logiciel définit la science qui regroupe les méthodes de travail et les pratiques de développement afin de produire un logiciel fiable et conforme aux besoins des utilisateurs. Ainsi, dans les phases précoces du cycle de vie logiciel et avec la complexité croissante des systèmes, il est devenu crucial de bien conduire les étapes de construction logicielle, pour bien cerner les insuffisances et éviter toutes les ambigüités, par exemple, un programme en cours d'implémentation reste difficile à le comprendre par un autre informaticien [BD09]. Toutes ces raisons sont à l'origine des contrôles successifs du produit, qui devrait satisfaire une bonne démarche de modélisation spécifique aux fonctionnalités attendues et décrites dans le cahier des charges. Un bon moyen de modélisation devra nécessairement s'adapter, à toute catégorie du système, surtout au niveau des systèmes critiques (Temps Réel, Temps Réel Embarqués, etc.) où la vérification de leurs contraintes temporelles reste difficile. Dans ce cas, il demeure indispensable d'imposer une vérification formelle sur les modèles générés.

## Contexte scientifique et problématique

Le standard UML (Unified Modeling Language) étant défini par l'Object Management Group (OMG) [1], est fortement utilisé dans le milieu industriel avec la prolifération de ses nouveaux profils, dédiés pour la modélisation des systèmes complexes. La notation simple et graphique qu'il offre, le rend un moyen efficace de communication entre les intervenants impliqués dans un système. Muni d'une diversité de diagrammes, UML a permis de modéliser un système sous plusieurs angles. Il est dépourvu d'une syntaxe claire et correcte avec une sémantique imprécise. Ses utilisateurs s'accordent à l'interprétation de plusieurs concepts bien connus, mais la précision d'autres concepts souffre d'une incohérence sémantique qui

1. https ://www.omg.org/

impose leur vérification.

Dans la littérature, plusieurs travaux ont traité l'émergence d'UML avec les langages et les formalismes mathématiques tels que les systèmes de transitions (réseaux de Petri, automates, etc.) ou bien les langages de spécification et de vérification (B, Z, etc.), afin de démontrer les aspects spécifiques désirés. Dans [Stö04a, Stö04b], les auteurs ont combiné formellement la simplicité de l'utilisation des diagrammes UML pour favoriser la conformité des propriétés fonctionnelles et non fonctionnelles. Nonobstant qu'un manque de sémantique s'approuve au niveau de la fiabilité, la sécurité de communication et le temps de réponse, surtout pour les systèmes critiques où leur complexité s'accroit de plus en plus, qu'il est devenu incontournable de corriger le côté fonctionnel et extra-fonctionnel.

Dans ce contexte, plusieurs approches de vérification formelle se sont intéressées à l'étude de la vue dynamique d'UML, mais peu de résultats ont été communiqués spécialement sur la vérification des nouveaux diagrammes définis sous UML2 (diagrammes globaux d'interaction (IOD), diagrammes de timing (TD), etc.).

En tenant compte qu'une utilisation directe d'une approche formelle devra respecter la sémantique du langage comme défini par l'OMG, nous tenons compte du choix des diagrammes qui se voit dépendant du type des systèmes que nous visons à étudier. Dans ce sens, le choix des TD comme une variante d'interaction est adéquat pour décrire et combiner l'ensemble des états des objets du système, dans un contexte temporel. Nous pensons de même que l'ensemble de ces interactions peuvent renforcer la modélisation d'un système temps réel (STR) regroupé dans une structure hiérarchique représenté par un IOD. Dans ce sens, nous décrivons formellement ces deux diagrammes en termes de formalismes de réseaux de Petri (RdP), qui se voient adéquats pour la modélisation des systèmes. Pour faciliter la tâche de la vérification des systèmes que nous visons à étudier, plusieurs autres techniques sont mises en exploitation telles que la formulation des contraintes OCL sur les STR, les logiques temporelles et temporisés, ainsi que les profils UML temps Réel proposés. Dans ce contexte, nous présentons notre travail de thèse.

## Objectifs et motivations

Dans cette thèse, notre premier défi consiste à proposer un ensemble d'approches de formalisation pour la vérification des IOD et des TD en termes de différentes extensions de RdPs. Dans ce sens, nous essayons de fusionner l'ensemble de ces formalismes dans une nouvelle approche hiérarchique, appliquée sur les STR par une technique de vérification après la transformation du modèle. Notre deuxième défi consiste à faire face aux exigences temporelles de ces types de systèmes, dans ce sens, les propriétés temporelles désirées seront exprimées sous forme de contraintes OCL temps réel $OCL^{TR}$, dédiées à augmenter

l'expressivité des diagrammes issus des STR et seront transcrites en termes de la logique temporelle temporisée TCTL sur les modèles générés. Les contraintes temporelles du système sont décrites à l'aide du profil UML MARTE [2] (Modeling and Analysis of Real-Time and Embedded Systems), à ce niveau, nous avons remarqué qu'aucune approche de vérification issue de la littérature, n'a eu les capacités d'émerger ce profil avec le formalisme des TD dans le contexte des IOD.

Plus clairement, nous viserons la mise en place d'un ensemble d'approches basé sur les IOD et les TD et combinés par de diverses extensions de RdP (colorée, hiérarchique, temporisée et temporelle). Nous définirons une première approche hiérarchique de formalisation pour la vérification d'un IOD muni de variantes d'interaction sous forme de diagrammes de séquence et de TD. Les jeux de test dynamiques ainsi que la vérification seront assurés utilisant l'outil CPN Tools [3]. Ensuite, nous définirons à base des TD, deux approches de formalisation utilisant deux extensions de RdP temporels, analysées et vérifiés utilisant les outils CPN Tools et Roméo [4] (nous présentons ces outils dans la partie Annexe). Nous exploiterons le profil UML MARTE pour spécifier les contraintes de durée et l'ensemble des contraintes temporelles du système. Par la suite, nous donnerons nos résultats de vérification exprimées à base de la logique temporelle TCTL et exploitées utilisant le model checker de Roméo. Ces formalismes proposés seront appliqués par de divers exemples d'illustration. Enfin et afin de bien formuler les propriétés temporelles, nous offrirons au concepteur UML un outil de traduction des contraintes $OCL^{TR}$ [omga] en termes de la logique TCTL. Toutes les approches proposées seront combinées dans une structure hiérarchique et appliquées sur une étude de cas (Integrated Modular Avionics(IMA)-Based AirBorne Systems), pour s'assurer de leur bon fonctionnement logique.

## Plan du mémoire

Le présent rapport est organisé en une introduction générale, six chapitres, une conclusion générale et une annexe.

Dans le premier chapitre, nous présentons un état de l'art sur le langage de modélisation UML et ses profils temps réel proposés pour la modélisation des STR, nous définissons spécialement le profil UML MARTE. Par la suite, nous décrivons les principaux langages et formalismes de spécification et de vérification.

Dans le deuxième chapitre, nous présentons un état de l'art sur la vérification formelle des diagrammes UML, les principales logiques temporelles en rapport avec nos travaux et les

2. https ://www.omgmarte.org/
3. http ://cpntools.org/
4. http ://romeo.rts-software.org/

techniques de vérification formelle, principalement la technique de vérification automatique par modèle (model checking). Par la suite, nous décrivons les principales approches de vérification combinant les diagrammes UML et les RdP, nous donnons le cas des approches basées sur les STR. Enfin, une synthèse des travaux existants et une introduction aux contributions sont établies.

Dans un troisième chapitre, nous présentons notre première contribution qui consiste en une nouvelle approche hiérarchique de formalisation pour la vérification des IOD moyennant les diagrammes de séquence (SD) et les TD en termes de RdP Colorés Hiérarchiques (HCPN). Cette approche est appliquée sur une partie du Système DAB 'Distributeur Automatique de Billets'.

Dans un quatrième chapitre, nous proposons une deuxième contribution qui consiste en deux approches de formalisation pour la vérification des TD en termes de RdP temporisés et temporels. La première approche propose une formalisation des TD en termes de RdP colorés temporisés notés TCPN, la vérification du modèle généré est assuré en utilisant l'outil CPN Tools. Cette approche est appliquée sur une étude de cas (Système de contrôle d'accès dans une salle serveur). La deuxième approche temporelle propose une formalisation des TD en termes de RdP temporels notés TPN (Transition-TPN). Cette approche est appliquée sur une étude cas (Processus de chargement d'une page web) et dont nous faisons part dans le chapitre qui suit utilisant la logique temporelle TPN-TCTL. La vérification du modèle généré est assurée en utilisant l'outil Roméo.

Dans un cinquième et sixième chapitre, nous décrivons notre troisième contribution qui consiste en une approche hiérarchique de formalisation donnée sous forme d'un IOD et un ensemble de TD et s'intéressant aux STR. Les résultats de vérification sont originaires de la formulation des propriétés décrites sous la logique temporelle TCTL et dérivées des contraintes ($OCL^{TR}$). Le profil UML MARTE est exploité afin de décrire les contraintes temporelles du système. Nous mettons en application cette contribution sur une étude de cas (IMA/Based Airborne Systems).

Enfin, une conclusion générale est destinée à résumer tous les travaux réalisés ainsi qu'un ensemble de perspectives est proposé pour planifier nos futures directions.

# Chapitre 1

# UML et les approches formelles

**Sommaire**



## 1.1 Introduction

Depuis la dernière décennie, UML demeure le standard de facto de la modélisation objet conçu par l'OMG, il vient pour exprimer visuellement une multitude de solutions à plusieurs vues et ce, par la diversité des diagrammes qu'il offre, servant à documenter des composants de larges systèmes complexes à dominante logicielle [omgc]. Les importantes révisions apportées pour UML (version 2.5 au moment de notre rédaction de ce rapport), viennent intégrer plusieurs nouveaux aspects statiques et dynamiques, représentant à la fois un objectif et un défi tenant à décrire la technologie logicielle dans des démarches MDA, MDE et SOA [SH05].

UML est défini par une sémantique ambigüe qu'une vérification formelle de ses diagrammes s'impose. Tout ceci exprime la nécessité de le déployer dans un formalisme mathématique [1]. Le choix d'un tel formalisme à appliquer est de grande importance, il convient de sélectionner la méthode correspondante selon la nature du système à étudier : choisir par exemple les langages de spécification formelle Z et B pour la modélisation des systèmes séquentiels, les RdP pour la modélisation des systèmes concurrents et distribués, etc. Dans ces dernières approches citées, la vérification est supportée, après avoir spécifié

1. pagesperso.lina.univ-nantes.fr/info/perso/permanents/attiogbe/

formellement le diagramme.

Dans ce premier chapitre, nous introduisons les principes de la modélisation UML, un court aperçu sur ses diagrammes et ses nouveaux profils proposés pour la modélisation des STR, principalement le profil UML MARTE. Ensuite, nous présentons les principaux langages et formalismes de spécification et de vérification formelle utilisés dans la littérature, à ce niveau, nous nous focalisons essentiellement à décrire le formalisme des RdP.

## 1.2 Le langage de modélisation UML

Modéliser un système revient à donner une abstraction de la réalité, subjective et pertinente. La combinaison de ces caractéristiques nous permet de se poser plusieurs questions : pour réussir le processus d'élaboration selon l'esprit traditionnel, où la modélisation et l'implémentation sont traitées séparément, faudrait-il coder ou modéliser avant ? Étant donné à nos mains un programme prêt, quel chemin nous précise ses modèles conçus ? Est-il possible de forcer la dépendance entre un modèle et un programme ? Est-il possible de générer du code à partir d'un modèle ou inversement ?

Dans ce sens, plusieurs techniques s'adaptent à base de nouvelles stratégies de l'ingénierie des systèmes (Forward Engineering, Reverse Engineering, RoundTrip Engineering, etc.) [BD09]. Ces différentes techniques utilisent les diagrammes d'UML2 (deuxième version d'UML2) comme les diagrammes de classe, d'empaquetage qui sont utilisés pour montrer les aspects statiques du système, alors que d'autres comme le diagramme d'activité, de machine d'état qui sont utilisés pour décrire les aspects dynamiques, d'autres diagrammes montrent des interactions entre les différents éléments du système comme les IOD et les TD. Dans ce qui suit, nous essayons de présenter brièvement les diagrammes apportés par UML2.

### 1.2.1 Les diagrammes UML2

UML2 se veut plus productif et devient le premier support des architectures dirigées par les modèles (MDA) [omgc]. Son incohérence sémantique et sa mauvaise interprétation sont tant reprochées par l'absence des relations entre les diagrammes associés à un modèle, la difficulté de son application aux systèmes distribués, la divergence concernant l'implémentation des langages, son adoption à grande échelle par des personnes sans qualifications exigées, l'absence d'un support hiérarchique quant à la composition des diagrammes, son nombre écrasant de diagrammes et de symboles [Mar06], etc. Toutes ces raisons demeurent parmi ses faiblesses majeures le rendant un langage quelconque, incapable de proposer un support de vérification. À ceci s'ajoute une multitude de concepts qui ont compliqué son

apprentissage et ont alourdi sa familiarisation avec ses différents diagrammes. Mise à part quelques points mineurs relevant plutôt de la cohérence de l'ensemble de sa notation, tous les éléments d'UML1 ont fait partie d'UML2, étendus par de nouvelles capacités. Bien qu'il y ait beaucoup plus à la modélisation qu'à UML, la réalité est qu'il définit les artefacts de modélisation standard, quand il s'agit de la technologie objet [Amb04].

La diversité des diagrammes UML2 telle qu'illustrée par la figure 1.1 est classée en deux principales vues :

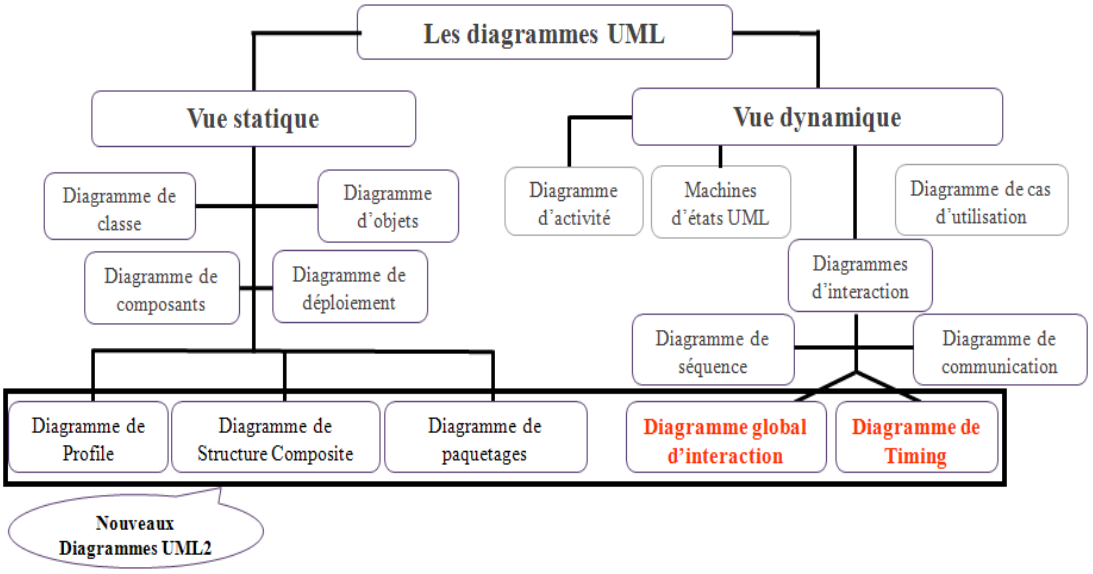



FIGURE 1.1 – Les diagrammes officiels UML2.5

Une première vue statique décrivant la structure du système et est représentée par un ensemble de diagrammes (classe, composants, déploiement, etc.). Une deuxième vue dynamique décrivant le comportement attendu du système, captée par un ensemble de diagrammes (séquence, TD, IOD, machine d'état, etc.) [Gra08]). Un ensemble de diagrammes peuvent une fois spécifier un modèle UML, la signification la plus utilisée dans une telle modélisation est :

— Le diagramme de classe qui représente la structure statique du système nommé le diagramme central de toute la modélisation objet,
— Le diagramme d'objets qui permet de mettre en évidence des liens entre les objets, représentant des instances des associations entre les classes et partageant un ensemble de caractéristiques,
— Le diagramme de composant qui décrit les modules (fichier, bibliothèque, table, etc.) pour composer le système (généralement servi pour les systèmes complexes), ainsi leur dépendance dans un environnement de réalisation,

— Le diagramme de cas d'utilisation qui décrit un ensemble de fonctionnalités du système et non pas des solutions d'implémentation. Un cas d'utilisation est une classe de fonctionnalités [BD09] fournie par le système. L'ensemble des cas d'utilisation permet de spécifier les besoins fonctionnels des utilisateurs, centrant l'expression des exigences fonctionnelles entre ces derniers et le système,
— Le diagramme de séquence qui représente le comportement du système en termes d'interaction éclatant le système. Son objectif global est d'enrichir le diagramme de classe afin d'identifier et trouver les objets manquants [BD09],
— Le diagramme de machine d'état qui représente un automate d'état finis reliés par des arcs orientés et décrivant des transitions possibles des différents états d'un objet. Un état d'un objet représente une conjonction instantanée de l'ensemble des valeurs de ses attributs [omgd],
— Le diagramme d'activité qui décrit un ensemble de fonctionnalités du système. Sémantiquement, il entrait en confusion sous UML1 avec les diagrammes d'états-transitions et partageait la même partie du méta-modèle UML. Sous UML2, ces deux derniers ont été séparés pour les décrire formellement avec l'ajout de certaines nouvelles notations [omgd],

Dans le contexte de nos travaux, nous nous intéressons essentiellement aux IOD et aux TD (marqués en rouge dans la figure 1.1), deux diagrammes nouvellement définis sous UML2 (nous fournissons leurs détails dans les chapitres 3 et 4).

### 1.2.2 Les profils UML temps réel

Vu la complexité croissante des systèmes temps réel (STR) et les STR embarqués, plusieurs profils UML temps réel (TR) ont été définis, afin de faire face à leurs exigences. Nous énumérons dans ce qui suit, les principaux profils et nous présentons spécialement le profil UML MARTE [omga].

#### 1.2.2.1 Principaux profils UML temps réel

Dans ses premières versions, UML est vue faible au niveau de la modélisation des STR, il ne possédait pas les atouts pour pallier le manque d'expression des propriétés sur ces derniers, cependant, le temps est spécifié d'une manière informelle. Par respect des contraintes temporelles, UML s'est fourni de ses propres extensions pour proposer un bon résultat [Mar06] de vérification au niveau de la sûreté de fonctionnement, la concurrence, le déterminisme, etc. Dans la littérature, plusieurs travaux ont traité l'utilisation des profils UML TR.

Dans [BBHP08], les auteurs ont prouvé l'intérêt du langage SysML (Systems Mode-

ling Language) [omgd, Sys07] présentait un champ d'application plus large qu'UML, le partageant un large sous-ensemble, héritant ses caractéristiques et sa sémantique pour la modélisation des systèmes complexes et embarqués, aux caractères distribués et TR. SysML est conçu pour modéliser et vérifier les systèmes à deux niveaux matériel et logiciel pour l'ingénierie des systèmes, il définit un mécanisme d'extension sans avoir à modifier le langage. Il est constitué par un ensemble de stéréotypes, de contraintes textuelles et un ensemble de valeurs étiquetées.

Par ailleurs, il existe plusieurs autres profils reconnus et dérivés d'UML pour la modélisation des STR, à savoir le profil QFTP [omgd] (Quality of Service and Fault Tolerance specification), le profil SPT [2] (Schedulability, Performance and Time) pour la modélisation des aspects liés aux performance et ordonnancement, le profil UML MARTE [omga], le profil DAMRTS (Dependability Analysis Model for Real Time Systems) [AAM05] pour la vérification des propriétés temporelles et probabilistes dans les STR, le profil TURTLE [3] (Timed-UML RT-Lotos Environment) et son successeur AVATAR dérivé de SysML pour la modélisation et la vérification des STR Embarqués, le profil UML+ [DBLM02], le profil OMEGA [4] pour assurer le développement correct des STR embarqués et aussi bien d'autres profils spécifiant la nature des systèmes à étudier.

#### 1.2.2.2 Le profil UML MARTE

Le profil UML MARTE [omgb] (Modeling and Analysis of Real-Time and Embedded Systems) dédié pour la modélisation des STR et STR Embarqués, est nouvellement apporté par UML [GP12]. MARTE offre de nouvelles capacités pour exprimer les propriétés temporelles (Package NFP (non-functional properties), l'allocation des ressources, les notions de temps quantifiable, etc.). MARTE est disponible sous toutes les plateformes de la modélisation unifiée. Actuellement, il semble remplacer UML pour la modélisation des concepts d'ordonnancement, de la performance et du temps. Son architecture consiste en trois principaux packages comme illustrée par la figure 1.2 [omgb] : '**MARTE Foundation package**', '**MARTE Design Model**' pour la conception des modèles issus des STR et des STR embarqués et '**MARTE Analysis Model**' pour l'analyse des modèles d'application. Ces deux derniers packages partagent des objets communs quant à la description des annotations temporelles et l'utilisation des ressources concurrentes, contenus dans 'MARTE Foundation package', ce dernier reclasse ses domaines en plusieurs autres packages à savoir le profil des éléments essentiels '**CoreElements**', le profil de la modélisation des propriétés non-fonctionnelles '**NFPs**', le profil de la modélisation du temps '**Time**', le profil de la mo-

2. http ://iaaa.cps.unizar.es/docencia/SWD/
3. http ://ttool.telecom-paristech.fr/
4. http ://www-omega.imag.fr/profile.php

délisation des ressources génériques '**GRM**' et le profil de la modélisation des allocations '**Alloc**'. La définition détaillée de chacun des packages est disponible dans [omgb].

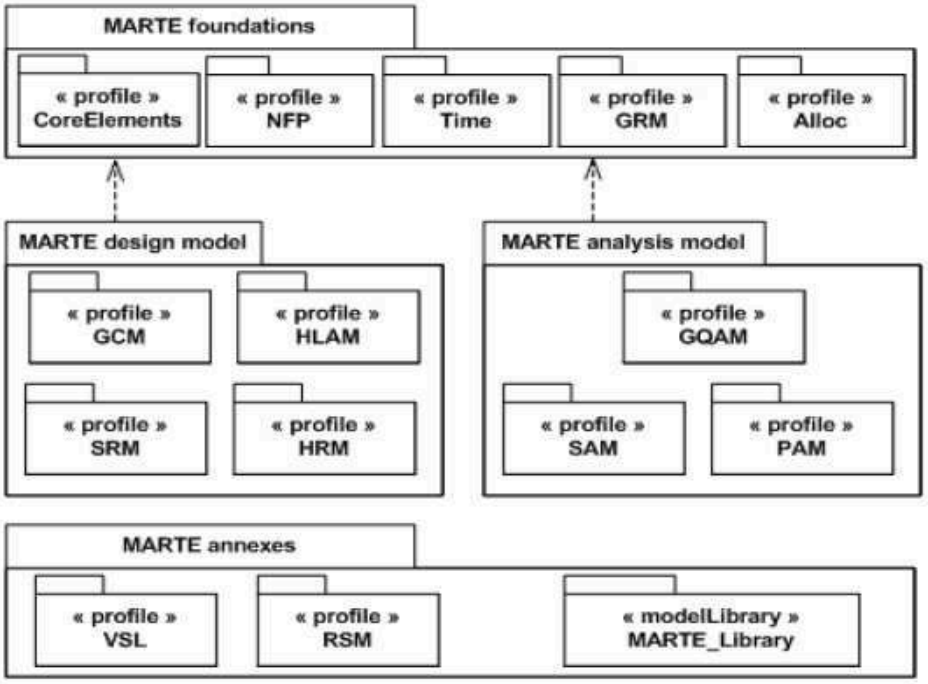



FIGURE 1.2 – Architecture du profil UML MARTE

Les caractéristiques de 'MARTE Analysis Model' se reclassent autour de trois autres packages, divisés en une partie générique fondamentale sous le profil '**GQAM**' et deux autres packages pour les domaines d'analyse spécifique (les profils '**SAM**' et '**PAM**'). Cependant, la structure de MARTE permet d'ajouter d'autres domaines d'analyse à savoir la consommation d'énergie, l'utilisation de la mémoire, la fiabilité, etc. Bien que d'autres packages sont fournis et reclassés dans son architecture issus du package annexes.

## 1.3 Panorama des principaux langages et formalismes de spécification et de vérification

Au sens informatique, une approche formelle est une méthode permettant de raisonner rigoureusement, moyennant une logique mathématique, sur un programme informatique ou un matériel électronique. Son objectif est de pouvoir démontrer une certaine validité quant à une certaine spécification attendue du système. Les langages et les formalismes issus de la littérature permettent de supporter la précision moyennant une sémantique complète. Dans ce sens, UML dispose d'une sémantique informelle qui rend difficile son utilisation directe comme un formalisme de vérification et ce, pour bien présenter les diagrammes selon

une compréhension aisée. Dans la suite, nous définissons ces principaux langages et formalismes dédiés pour la spécification et la vérification des systèmes, nous nous intéressons spécialement au formalisme des RdP pour leur fondement mathématique, leurs extensions variées et adéquates avec la diversité des diagrammes UML, ainsi que la robustesse qu'ils offrent au niveau de la modélisation et la vérification des systèmes.

### 1.3.1 Principaux langages de spécification et de vérification

Ces langages tendent à donner un aspect plus formel à la sémantique des modèles et à leur analyse automatique, nous pouvons les classifier comme suit :

#### 1.3.1.1 Les langages de spécification formelle algébrique

Les langages de spécification formelle (Z, B, Object Z, etc.) représentent des approches de génération qui tendent à définir une sémantique formelle, prenant en charge l'aspect réactif des modèles sources. Leur but est de pouvoir vérifier les exigences fonctionnelles des processus et des modèles conçus.

#### 1.3.1.2 Les langages de vérification par modèle

Les langages de vérification par modèle tendent à générer des programmes corrects, garantissant l'absence d'erreurs. En général, ces langages suivent une méthodologie d'analyse dynamique, dans laquelle les systèmes sont modélisés comme des modèles à états finis, utilisant une logique temporelle pour la vérification formelle des propriétés. Les langages les plus utilisés sont Alloy/Alloy-Analyzer, Maude/ Maude Checker, UPPAAL, SMV-SL de SMV et Promela de Spin. Ces langages prouvent l'incohérence des modèles sémantiques et des modèles d'analyses, ce qui ne garantit pas la vérification des spécifications formelles générées.

#### 1.3.1.3 Les langages de programmation

Les langages de programmation tendent à formaliser les modèles à base de leur code source généré. Plusieurs travaux de formalisation UML ont été traités à propos, se basant spécialement sur les diagrammes d'activité pour capturer les spécifications dans un cadre de langages synchrones et aboutir à des modèles exécutables. Dans [EJW02], les auteurs ont proposé un algorithme pour l'exécution des diagrammes d'activité UML1, traitant leur sémantique comme un workflow avec une grande vigueur.

### 1.3.2 Principaux formalismes de spécification et de vérification

#### 1.3.2.1 Les systèmes de transitions étiquetés

Ces systèmes renferment un ensemble d'états liés par une relation de transition dirigée et étiquetée. Deux états $s$ et $s'$ sont liés par la relation de transition si et seulement si le système pourra changer d'état de $s$ à $s'$ [EJW02]. Dans un autre sens, un système de transitions étiqueté peut être défini par la description de sa relation de sa relation de transition et de son état initial, dans ce cas, l'ensemble des états est constitué par tous les états atteignables à partir de l'état initial par la relation de transitions. Tout de même, il peut être défini par la composition des systèmes de transitions déjà construits. Généralement, ces systèmes sont généralement utilisés pour le traitement des workflow.

#### 1.3.2.2 Les automates à états finis

Ces automates représentent des machines abstraites se composant d'un ensemble fini d'états possible, un ensemble fini de symboles en entrée et une fonction de transition entre ces différents états. Les automates s'inspirent de la sémantique des machines à états représentés à l'aide d'une table de transitions et un ensemble de notations formelles. Formellement, un automate à états finis est défini comme suit :

**Definition 1.1** $A = (Q, \sum, \delta, q_0 \text{ et } F)$ où :

- $Q = \{q_1, q_2, q_3, ...\}$ représente un ensemble fini d'états,
- $\sum = \{a, b, c, d, ...\}$ représente un ensemble fini de symboles alphabétiques,
- $\delta(q_i, a) : q_j$ représente une fonction de transition prenant en paramètre un état $q_i$ et un symbole $a$ et retourne un état $q_j \backslash (\ll signature \gg \delta : Q \times \sum \to Q)$, cette fonction pourra renvoyer $\emptyset$ (exemple : $\delta(q_1, d) = \emptyset$. $\delta$ pourra prendre un mot en entrée $\alpha \backslash (\alpha \in \sum^*) : \delta(q_i, \alpha) = q_j (\ll signature \gg \delta : Q \times \sum^* \to Q)$,
- Un état initial $q_0 \backslash q_0 \in Q$,
- $F = \{f_1, f_2, ...\}$ représente un ensemble d'états finaux $\backslash F \subseteq Q$,

Entre autre, les automates temporisés représentent une variante des automates à états finis, utilisés généralement pour la vérification des modèles issus des STR. Le temps est émis sous forme de variables réelles appelées horloges, de nombre fini et admettant une même vitesse. Dès qu'une transition est sensibilisée, l'horloge mesure le temps écoulé et la transition devient tirable dans l'intervalle temporel marqué à cette transition. Deux configurations d'évolution associées à cette variante [BM06] : une première où le temps s'écoule et l'automate reste dans son même état avec une augmentation des valeurs des horloges d'un délai d'attente. Une deuxième où l'automate passe à un état suivant dès qu'il franchit une transition discrète, associe des contraintes sur les valeurs des horloges.

#### 1.3.2.3 Les Réseaux de Petri (RdP)

Un RdP standard représente un modèle formel abstrait des flux d'informations [Bal07], qui constitue un support de modélisation des activités asynchrones et concurrentes, pour les aspects basiques des systèmes distribués, parallèles, non-déterministes et/ou stochastiques. Il permet de maîtriser et d'assurer la sûreté de fonctionnement des applications complexes (aéronautique, industrie, etc.) et demeure de même un outil graphique de modélisation et d'analyse, des systèmes parfaitement adaptés à l'étude des structures de contrôle [Bas00]. Par la figure 1.3 ci-dessous, nous illustrons un exemple d'un RdP place-transition, où les places empty et holding coin sont appelées des places de marquages, reliées toutes les deux par un ensemble de transitions, qui assure le changement de l'état de marquage dans le modèle.

Informellement, un RdP est un graphe orienté biparti mettant en jeu un ensemble de places et de transitions où elles représentent respectivement une action élémentaire et une ressource munies par des arcs.

Formellement, l'utilisation d'un RdP demeure importante pour la vérification des systèmes concurrents. Dans ce sens, plusieurs travaux tel que [MDM03], où les auteurs l'ont proposé pour la détection automatique des symétries.

Dans [Bal07], les auteurs l'ont formellement défini comme suit :

**Definition 1.2** $M_{PN} = (P, T, F, W, \text{et } M_0)$, où :

- $P = \{p_1, p_2, .., p_P\}$ est un ensemble fini de places,
- $T = \{t_1, t_2, .., t_T\}$ est un ensemble fini de transitions$\backslash P \cap T = \emptyset$ et $P \cup T \neq \emptyset$,
- $F \subseteq P \times T \cup T \times P$ représente un ensemble fini d'arcs orientés,
- $W : F \rightarrow (1, 2, 3, ...)$ est une fonction d'affection des poids des arcs,
- $M_0 = \{m_{01}, m_{02}, .., m_{0P}\}$ représente le marquage initial du réseau,

La modélisation d'un système peut mener à un RdP de taille volumineuse, rendant leur manipulation et/ou leur analyse difficile. L'objectif est de trouver un moyen pour modifier la structure du modèle à générer afin de réduire la taille, afin d'obtenir un autre modèle. Cette problématique a motivé l'introduction d'une nouvelle variante des RdP, à savoir les RdP colorés notés CPN dans le reste du rapport. Par application au domaine formel, plusieurs travaux l'ont exploité tels que dans [Vau87], où les auteurs l'ont proposé pour la modélisation des spécifications dans les systèmes parallèles.

**Les RdP colorés (CPN) :** Pour mieux clarifier cette extension, nous prenons un exemple : Supposons que nous avons un stock contenant plusieurs types de pièces ; Choisissons un RdP classique pour le modéliser, nous nous rendons compte que chacun des

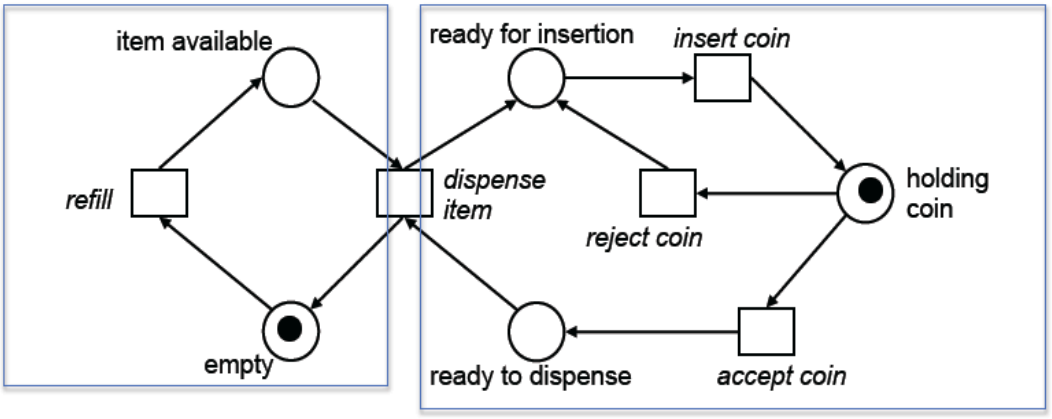



FIGURE 1.3 – Exemple d'un RdP place-transition

types de pièces, nécessite de créer une place stock, ce qui entraîne la création de places supplémentaires pour la représentation du stock. Par contre et avec un CPN, nous représentons tout le stock par une seule place. Pour distinguer les différents types de pièces, nous assignons une couleur (valeur d'un domaine) aux jetons de chaque type de pièces.

Dans [Jen07, CKZ11], les auteurs l'ont formellement défini comme suit :

**Definition 1.3** $M_{CPN} = (P, T, A, D, V, C, G, E \text{ et } I)$, où :

- P représente un ensemble fini de places,
- T représente un ensemble fini de transitions$\backslash P \cap T = \emptyset$ et $PT$,
- $A \subseteq P \times T \cup T \times P$ représente un ensemble fini d'arcs orientés,
- D représente un ensemble fini et non vide de domaines de couleurs (types),
- V représente un ensemble de variables typées$\backslash v \in V$, $type[v] \subseteq D$,
- $C : P \rightarrow D$ représente une fonction assignant un ensemble de couleurs typées à chaque place,
- $G : T \rightarrow Expressions$ représente une fonction booléenne assignant une condition par transition$\backslash t \subseteq T$, $[Type(G(t)) = Bool$ et $Type(Var(G(t)) = D]$. Par défaut, la garde sur un arc est vraie, Var (G(t)) représente l'ensemble des variables de G(t),
- $E : A \rightarrow Expressions$ représente une fonction assignant une expession par arc $\backslash a \subseteq A$ et $[Type(E(a)) = C(p)$ et $Type(Var(E(a)) = D]$,
- $I : P \rightarrow Expressions$ représente une fonction d'initialisation assignant un marquage initial à chaque place p $\backslash p \subseteq P$ et $[Type(I(p)) = C(p)]$,

**Les CPN hiérarchiques temporisés (THCPN) :** Le temps et les CPN sont introduits par [Jen07]. La sémantique proposée est fortement exploitée par l'outil CPN Tools [5]. Les auteurs ont formellement défini les THCPN comme suit :

**Definition 1.4** $T_{HCPN} = (CPN, R, r_0)$ où :

- CPN représente un RdP coloré,
- R représente l'ensemble des valeurs inclus dans $R_0^+$,
- $r_0$ est la valeur initiale dans R,

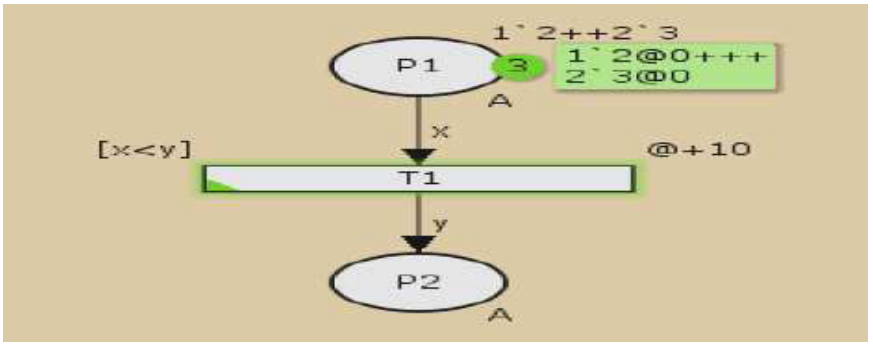



FIGURE 1.4 – Une transition dans un THCPN

Les THCPN utilisent les transitions de substitution comme montrée par la figure 1.4. Pour définir cette transition, $t1$ est tirée dans un temps de $10(@+10)$ et qui va être ajoutée au jeton produit par $P1$. Le marquage de la place $P2$ avec le jeton est noté par un opérateur spécifique $@+10$ (par exemple $1'2@+10$). Ce délai détermine la durée pendant laquelle, le jeton reste indisponible (temporisation du jeton). Ce jeton ne pourrait être disponible dans $P2$ qu'après dix unités de temps, une fois que la transition est franchie.

## 1.4 Conclusion

Tout au long de ce chapitre, nous avons présenté le langage UML, ses nouveaux profils temps réel définis pour renforcer la modélisation des STR et des STR embarqués, spécialement, nous avons mis le point sur le profil UML MARTE. Ensuite, nous avons décrit les principaux langages et formalismes de spécification et de vérification utilisées dans la littérature, nous nous sommes focalisés spécialement à décrire le formalisme des RdP. Dans le chapitre suivant, nous donnerons un état de l'art sur les principales approches de vérification formelle qui utilisent ces derniers formalismes et les combinant avec le langage UML.

5. http ://cpntools.org/

# Chapitre 2

# État de l'art sur la vérification formelle des diagrammes UML

**Sommaire**



## 2.1 Introduction

Dans ce chapitre, nous présentons les principales logiques temporelles et temporisées en rapport avec le contexte de nos travaux ainsi que les principales techniques de vérification. Par la suite, nous essayons de donner un état de l'art général sur les principales approches de vérification formelle sur les diagrammes UML et ses profils, spécialement celles les combinant avec le formalisme des RdP, nous étudions de même celles développées sur les STR et utilisant les IOD. Enfin, nous donnons une synthèse des travaux analysés et une introduction aux contributions à accomplir.

## 2.2 Les techniques de vérification formelle

### 2.2.1 Les preuves de théorèmes

Les preuves de théorèmes nécessitent la construction d'une partie de la preuve au minimum par l'utilisateur. Les spécifications peuvent être données en termes de logiques ou de spécifications algébriques et les preuves sont faites sur les modèles de la spécification en utilisant des prouveurs. Si le model checking peut conduire à une explosion combinatoire du nombre d'états (trop coûteuse en termes du temps), cette technique aura la possibilité de prendre en compte des données complexes. Son inconvénient majeur est la relative complexité des preuves manuelles.

### 2.2.2 Les tests d'équivalence

Les outils de test d'équivalence utilisent des relations d'équivalence particulières pour vérifier que deux modèles partagent une propriété donnée. Il s'agit souvent de prouver une propriété relativement abstraite, avec peu de données. Comme pour la vérification automatique par modèle, il est nécessaire que les deux systèmes soient finis, même s'il existe des techniques où la vérification se fait simultanément avec la construction des systèmes.

### 2.2.3 Animation de spécifications

Une animation de spécifications permet de valider une séquence (dans le cadre de systèmes dynamiques) ou une construction (dans le cadre de spécifications algébriques). Elle permettra d'avoir une sorte de prototype de la spécification ou du système spécifié. De nombreux outils dédiés à l'animation de spécifications sont omniprésents (exemple de l'outil CPN Tools pour les CPN).

Plusieurs autres techniques existent dans la littérature (les BDD, les assistants des preuves, les tests, etc.). Les BDD demeurent une représentation de formules booléennes utilisée surtout pour la vérification des circuits numériques, qui admettent l'avantage d'être parfois exponentiellement plus compacte que ses formules explicites. Les assistants de preuves permettent à un utilisateur de démontrer des théorèmes sur le programme, avec une aide et une vérification par la machine. Par ailleurs, la vérification automatique par modèle **(model checking)** demeure la technique la plus remarquable dans la littérature, nous la décrivons comme suit :

### 2.2.4 Vérification automatique par modèle (Model Checking)

Le model checking (figure 2.1) se base essentiellement sur la construction de l'espace d'états accessibles, permettant de vérifier le respect des propriétés attendues du système, plus particulièrement les propriétés temporelles temporisées, en parcourant l'ensemble de ces états. Une formule d'une logique temporelle est généralement évaluée soit sur une trace (logique linéaire), soit sur un arbre de branchement (logique arborescente ou de branchement). L'évaluation à base des traces indique que la formule est évaluée sur une des exécutions possibles du système tandis que dans l'évaluation sur un arbre de branchement pourrait raisonner sur toutes les évolutions futures possibles.

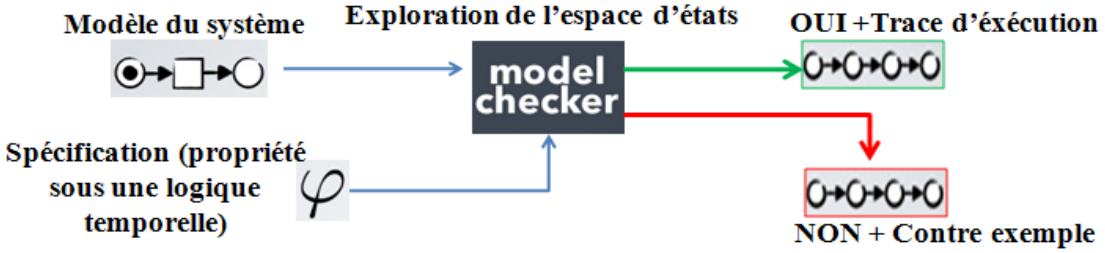



FIGURE 2.1 – Processus du model checking

## 2.3 Les logiques temporelles et temporisées

Les logiques temporelles sont utilisées généralement pour exprimer et spécifier formellement un comportement sur un modèle d'exécution, utilisant des propositions qui associent des valeurs sur les variables afin de définir les états de ce modèle [Nas07]. Un comportement est défini par un séquencement d'états, exploité par des opérateurs logiques au sein d'une propriété linéaire ou arborescente, suivant une syntaxe et une sémantique bien adoptées. La formule propositionnelle utilise une seule ou double implication logique des opérateurs logiques de conjonction ou de disjonction, les négations ainsi que des valeurs booléennes évaluant le résultat de cette formule. Une propriété est exploitée par l'exploration d'un système de transitions fini. Les logiques temporelles et temporisées peuvent être réparties en deux classes : une première classe de logiques qualitatives et une deuxième classe de logiques quantitatives.

### 2.3.1 Les logiques qualitatives

Ces logiques telles que (LTL, CTL, etc.) sont reconnues par leur pouvoir limités d'expression, avec un aspect temporel abstrait. Dans ce sens, les durées exactes d'apparition

des états et leur succession sont ignorées. Dans cette section, nous nous limitons seulement à présenter la logique temporelle CTL.

**Logique temporelle CTL :** L'ensemble des opérateurs temporels issu de la logique CTL [CE81, Nas07] est composé de quantificateurs de chemins utilisés pour exprimer des formules en logiques arborescentes, suivis d'opérateurs temporels exprimant des propriétés sur le comportement. A ce niveau, nous pouvons classifier deux quantificateurs A et E. A (**pour tout chemin**) exprime qu'une propriété est vérifiée par tous les séquencements d'états du modèle, commençant à base de l'état courant. Le quantificateur E (**Existe**) pour indiquer qu'une propriété est vérifiée par au moins un seul séquencement partant de l'état courant du modèle en simulation. D'autres opérateurs comme G(**Globally**), F(**Eventually**), U(**Until**) et X(**Next**) sont exploités par les quantificateurs suivant le contexte de la formule propositionnelle demandé.

- L'opérateur G (**toujours**) vient exprimer des propriétés pour les quelles une formule propositionnelle est vraie dans tous les états d'une simulation,
- L'opérateur F (**finalement**) permet d'exprimer qu'un état pourra vérifier une propriété dans le futur, à titre d'exemple $G(p \Rightarrow q)$ signifie qu'à partir d'un état vérifiant $p$, nous aurons finalement dans le futur un état satisfaisant la propriété $q$,
- L'opérateur U (**jusqu'à**) exprime qu'une propriété sera vérifiée tant qu'une autre ne l'est pas, à titre d'exemple $pUq$ signifie que $p$ sera vérifiée sur tous les états jusqu'à ce que $q$ le soit, par conséquent, $q$ devra être obligatoirement satisfaite dans le futur,
- L'opérateur X (**Suivant**) indique qu'une propriété sera vérifiée dans l'état suivant, partant de l'état courant, à titre d'exemple, $G(p \Rightarrow Xq)$ signifie que si un état vérifie la propriété $p$ alors l'état suivant devra satisfaire la propriété $q$,

**Syntaxe de CTL :** Syntaxiquement, nous pouvons considérer la logique CTL par un ensemble de propositions, définie ci-dessous par :

$$\varphi ::= T \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid AX\varphi \mid E(\varphi U \varphi) \mid A(\varphi U \varphi) \tag{2.1}$$

Chaque formule CTL est évaluée dans les états d'une structure arborescente ainsi que chaque chemin devra commencer du point de l'état de l'évaluation de la formule. L'opérateur E indique **'il existe un chemin tel que'** alors que l'opérateur A indique **'tout chemin vérifie'**.

Exemple : $EX\varphi$ signifie qu'il existe un chemin partant de l'état courant, le successeur de l'état courant satisfait $\varphi$.

$A(\varphi_1 U \varphi_2)$ désigne que le long de tout chemin partant de l'état courant, tel que $\varphi_2$ sera vraie dans un état futur et tous les états intermédiaires satisfaient $\varphi_1$.

D'autres opérateurs sous CTL pourraient être exploités tel que : $EF\varphi$, $EG\varphi$, $AF\varphi$ et $AG\varphi$ :

Exemple :

— $EG\varphi = \neg AF\neg\varphi$ désigne qu'il existe un chemin dans le quel $\varphi$ est toujours satisfaite,

— $AG\varphi = \neg EF\neg\varphi$ signifie que $\varphi$ est toujours vraie pour tout état accessible,

**Sémantique de CTL :** Une formule CTL est définie sur une séquence d'éxécution des LTS (Systèmes de transitions étiquetées) $ST = (\sum, Q, q_0, Tr, \rho)$ tel que $\sum$ est l'alphabet d'étiquettes de transitions, $Q$ est l'ensemble des états, $q_0$ est l'état initial, $Tr \subseteq Q \times \sum \times Q$ représente l'ensemble des transitions étiquetées et $\rho : Q \to 2^P$ est une fonction qui associe à chaque état un ensemble de propositions. Une structure S est définie par un ensemble tel que $S = (ST, \prod)$, où ST est un système de transitions étiquetées et $\prod$ est l'ensemble des chemins d'exécutions. Un chemin est défini par un séquencement d'états fini ou infini de la forme $q_0q_1q_2...q_n...$ stable par suffixe, de la sorte que :

— Pour tout i, il existe une étiquette l tel que $(q_i, l, q_{i+1}) \in Tr$,

— Si la séquence est finie et se termine en $q_n$, alors $q_n$ n'a pas de successeurs,

Soit $P$ un ensemble de propositions. Nous définissons par induction sur la structure des formules CTL. La relation "$S, q \models \varphi$", abrégée en $q \models \varphi$ à partir des règles suivantes :

— $q \vDash T$,

— $q \vDash p$ si et seulement si $p \in \rho(q)$, p est une proposition de $P$,

— $q \vDash \neg\varphi$ si et seulement si $q \nvDash \varphi$,

— $q \vDash \varphi_1 \wedge \varphi_2$ si et seulement si $q \vDash \varphi_1$ et $q \vDash \varphi_2$,

— $q \vDash AX\varphi$ si et seulement si pour tous les chemins $(q_0q_1...) \in \Pi$ tels que$(q_0 = q)$ , nous avons $q \vDash \varphi$,

— $q \vDash EX\varphi$ s'il existe un chemin $(q_0q_1...) \in \Pi$ tel que$(q_0 = q)$ et $q_1 \vDash \varphi$,

— $q \vDash A(\varphi_1 \cup \varphi_2)$ si et seulement si pour tous les chemins $(q_0q_1...) \in \Pi$ tels que$(q_0 = q)$ et il existe $i \geq 0$ ,$q_i \vDash \varphi_2$ et pour tout $0 \leq j < i$, $q_j \vDash \varphi_1$,

— $q \vDash E(\varphi_1 \cup \varphi_2)$ si et seulement si pour tous les chemins $(q_0q_1...) \in \Pi$ tels que$(q_0 = q)$ et il existe $i \geq 0$ ,$q_i \vDash \varphi_2$ et pour tout $0 \leq j < i$, $q_j \vDash \varphi_1$,

Exemple : Considérons une ressource et son processus demandant avec deux états demande et satisfait, nous voulons exprimer la propriété rassurant la vivacité bornée du modèle. Toute demande de ressource par un processus finira par être toujours satisfaite. Sous la logique CTL, nous pouvons exprimer cette propriété comme suit (équation 2.2) :

$$AG(demande(p) \Rightarrow EFsatisfait(p)) \tag{2.2}$$

### 2.3.2 Les logiques quantitatives

Contrairement aux logiques qualitatives, les logiques quantitatives telles que MITL [AFH96], TLTL [Zha94], TCTL [BGR09] modélisent l'aspect temporel partageant une certaine qualité. Dans ce sens, les durées d'apparition des états et leurs successions sont déterminées. Dans cette section, nous nous limitons à présenter la logique temporelle TCTL, celle en rapport avec nos travaux.

**Logique temporelle TCTL :** La logique TCTL [Yov98, BGR09] représente une variante temporisée de la logique CTL, par l'ajout d'une notion de temps quantifiée dans la syntaxe et la sémantique. Ce qui dit que la temporisation est affectée sur les opérateurs logiques de la formule. Elle ajoute à ce cadre des informations quantitatives sur les délais séparant les actions, construites à partir de propositions atomiques, de connecteurs logiques et de combinateurs temporisés qui utilisent la modalité U (**Until**).

**Syntaxe de TCTL :** La logique TCTL combine un domaine de temps $\top$ et un ensemble de propositions $P$ [Nas07]. La syntaxe des formules TCTL est définie sur l'ensemble $P$ par :

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid A(\varphi U_{\sim k}\varphi) \mid E(\varphi U_{\sim k}\varphi) \tag{2.3}$$

où : $k \in \top$ et $\sim \in \{=, <, >, \leq, \geq\}$

Nous pouvons utiliser les règles suivantes pour exprimer d'autres opérateurs :

— $EF_{\sim k}\varphi = E(\top U_{\sim k})$,

— $EG_{\sim k}\varphi = \neg AF_{\sim k}\neg\varphi$,

— $AF_{\sim k}\varphi = A(\top U_{\sim k})$,

— $AG_{\sim k}\varphi = \neg EF_{\sim k}\neg\varphi$,

Exemple : $EF_{\leq 3}\varphi$ signifie qu'il est possible d'atteindre dans au moins 3 unités de temps, un état vérifiant $\varphi$.

**Sémantique de TCTL :** Une formule TCTL s'interprète sur les états d'un système de transitions temporisé $STT = (\sum, Q, q_0, Tr, \rho)$ tel que $\sum$ est l'alphabet d'étiquettes de transitions, $Q$ est l'esemble d'états, $q_0$ est l'état initial, $Tr \subseteq Q \times T \times (\sum \cup \{\in\}) \times Q$ est l'ensemble des transitions munies de durées et d'étiquettes. Enfin, $\rho$ est une fonction permettant d'étiquetter chaque état par des propositions atomiques. Dans [Nas07], une structure arborescente temporisée est définie par $S_t = (STT, \Pi^t)$, tel que $STT$ est un système de transitions temporisé et $\Pi^t$ est un ensemble de chemins temporisés. Un chemin temporisé de $\Pi^t$ est une séquence finie ou infinie d'éléments de $Q \times T$ de la forme $(q_0, t_0), ...(q_n, t_n)...$ où :

— Pour chaque indice $i$, il existe une étiquette $l$ telle que $(q_i, t_i, l, q_{i+1}) \in Tr$ et $t_i \leq t_{i+1}$,
— La séquence est finie et se termine par $(q_n, t_n)$ si $q_n$ n'a pas de successeur,

Soit $q \in Q$ et $t \in \top$. La relation de satisfaction $(q,t) \vDash \varphi$ est définie inductivement par :

— $(q,t) \vDash \top$,
— $(q,t) \vDash p$ si et seulement si $p \in \rho(q)$,
— $(q,t) \neg \vDash$ si et seulement si $(q,t) \nvDash \varphi$,
— $(q,t) \vDash \varphi_1 \wedge \varphi_2$ si et seulement si $(q,t) \vDash \varphi_1$ et $(q,t) \vDash \varphi_2$,
— $(q,t) \vDash A(\varphi_1 U_{\sim k} \varphi_2)$ si et seulement si pour tous les chemins $((q_0,t_0)...(q_n,t_n)...) \in \Pi^t$ tels que $q_0 = q$ et $t_0 = t$, il existe $t'$ tel que $(t' \sim t + k \wedge t' \geq t, (q_n, t') \vDash \varphi_2)$, et pour tout $0 \leq j \leq n$ et $t'' \in [t, t']$, $(qi, t'') \vDash \varphi_1$,
— $(q,t) \vDash E(\varphi_1 U_{\sim k} \varphi_2)$ si et seulement si pour tous les chemins $((q_0,t_0)...(q_n,t_n)...) \in \Pi^t$ tels que $q_0 = q$ et $t_0 = t$, il existe $t'$ tel que $(t' \sim t + k \wedge t' \geq t, (q_n, t') \vDash \varphi_2)$ et pour tout $0 \leq j \leq n$ et $t'' \in [t, t']$, $(qi, t'') \vDash \varphi_1$,

Exemple : Test de la propriété de sûreté du modèle : toute ressource occupée par un processus doit être libérée dans k unités de temps. Sous la logique TCTL, nous pouvons exprimer cette propriété comme suit (équation 2.4) :

$$AG(passe(p) \Rightarrow AF_{\leq k} libere(p)) \tag{2.4}$$

## 2.4 Panorama des principaux travaux de vérification UML : Cas des RdP

La migration des versions d'UML a prouvé une description plus concise au niveau de la clarté sémantique de ses aspects statiques. Cependant et jusqu'à aujourd'hui, aucune approche de vérification formelle n'a eu les capacités d'unifier ses aspects dynamiques qui restent informellement décrits.

La littérature est riche en termes de travaux ayant pour objectif la vérification des diagrammes UML. Quelques travaux tels que [SKM01, Mos07], où les auteurs ont utilisé le model checker Promela de SPIN pour vérifier les comportements des diagrammes d'états-transitions sous forme d'automates de Büchi, respectivement sous la forme d'une structure de Kripke et en termes d'opérateurs SMV. Dans [EBSC04], les auteurs ont formellement défini les diagrammes UML en termes d'opérateurs SMV. Aussi et dans [SB06b, SB06a, APS03], les auteurs ont formellement décrit les diagrammes dynamiques UML en termes de spécifications formelles algébriques Z et B. Dans [HKL+10], les auteurs ont essayé de traduire les diagrammes UML dans un langage de spécification de processus algébrique. Dans [SMCC13], les auteurs ont vérifié le comportement des systèmes (blo-

cage, synchronisation) utilisant les diagrammes de machines d'état et les diagrammes de cas d'utilisation par génération automatique de cas de test.

Entre autre, plusieurs autres approches ont combiné les diagrammes dynamiques d'UML et les RdP. Dans [Stö04a, Stö04b, Stö04c, Stö04d, FTJR07], les auteurs ont formellement défini les diagrammes d'activité, de séquence et de cas d'utilisation en termes de CPN et de CPN hiérarchiques utilisant l'outil CPN Tools. Dans [DL03], les auteurs ont essayé de formaliser et de vérifier les diagrammes dynamiques UML à base des RdP. Dans [BBB14], les auteurs ont formellement décrit et vérifié les diagrammes d'activité à base des RdP transactionnels. Dans [BDM02, GMC04, CM06, MH08], les auteurs ont formellement défini les diagrammes dynamiques en termes de classes instanciables et stochastiques de RdP. Dans [ZS04], les auteurs ont proposé une approche en émergeant les diagrammes de machines d'état et les diagrammes de communication avec les CPN. Dans [YS06], les auteurs ont exploité ce dernier travail et ont proposé une approche hiérarchique pour optimiser le modèle d'exécution, utilisant les états composites. Dans [BT06], les diagrammes d'activité et de séquence ont été exploités pour la modélisation des échanges de messages en termes de RdP à objets. Dans [Sta07], les auteurs ont formellement défini les diagrammes de séquence en termes des modèles de réseaux à processus. Dans [KM10], les auteurs ont exploité des techniques de modèles dirigés par la vérification. Dans [ES09], les auteurs ont utilisé différentes variantes de RdP pour formaliser les diagrammes de cas d'utilisation et de séquence.

## 2.5 Travaux de vérification formelle UML pour les STR

Les approches de vérification formelle sur les STR et les STR embarqués ont prouvé une bonne conduite au cours du processus de développement logiciel. Sauf qu'elles restent toujours dépendantes des exigences temporelles définies dans leur modélisation, constituant une fonction assez délicate pour assurer leur fiabilité et leur sûreté.

Dans ce contexte, plusieurs profils UML TR [omgc, omgd] (chapitre1) ont été proposés pour agir sur la performance des systèmes et capter leurs aspects temporels. Dans ce qui suit, nous décrivons les principales approches de vérification qui les utilisent. Dans [AMCN09, KZJK10], les auteurs ont essayé d'exprimer les architectures et/ou les comportements du système pour la vérification des exigences temporelles et non fonctionnelles. Dans [DBLM02], les auteurs ont combiné le profil TR UML+ et les automates temporisés utilisant le model checker de Kronos pour la vérification des comportements du système et ce, en exploitant les tendances temporelles des diagrammes d'états-transitions pour assurer une haute qualité des spécifications à faible coût. Dans [EBC03], les auteurs ont développé un outil de vérification (TABU : Tool for the Active Behavior UML) utilisant un assistant

d'écriture pour la vérification des propriétés (logique LTL) et ce, par une transformation des diagrammes de machines d'état et des diagrammes d'activité en termes d'opérateurs SMV pour minimiser les erreurs systèmes et leurs coûts de production. Dans [BACTHTM07], les auteurs ont proposé une approche de vérification sur les STR en termes d'algèbre de processus (CSP+T) afin d'assurer leur fiabilité et leur ordonnancement. Dans [SA09], les auteurs ont proposé une approche TTOOL (TURTLE Toolkit) pour la vérification des exigences temporelles sur les STR embarqués à base d'un profil UML TR générique, définissant une sémantique permettant de l'utiliser avec les outils de développements des STR (RTL, CADP, UPPAAL, etc.). Dans [AAM05], les auteurs ont formellement défini les diagrammes de machines d'état en termes d'automates temporisés probabilistes, définissant un nouveau profil DAMRTS. Dans [MLG01], les auteurs ont proposé une nouvelle approche pour la conception des fonctionnalités, utilisant la notion de l'encapsulation fonctionnelle, dans ce sens, ils ont émergé un nouveau profil UML embarqué assurant une combinaison matérielle et logicielle des systèmes.

## 2.6 Synthèse des travaux et contributions

Dans cette partie, nous proposons de limiter les approches de vérification formelle, spécifiquement de décrire celles liées au contexte de nos travaux. Nous entamons par une introduction aux différentes contributions à accomplir.

Suite à une analyse dans la littérature, nous nous sommes focalisés spécialement sur les travaux de [HKL+10, AMCN09, KZJK10, BT09, BBB14, MH08, BMMR10, Stö04c, CM06, BCD05], qui ont marqué les approches de vérification liées au contexte de nos contributions. Cependant, nous avons remarqué que minime sont les travaux qui ont abordé la vérification formelle des IOD [AMCN09, BT09, BMMR10] et presqu'aucune approche n'a utilisé le formalisme des TD pour développer son approche sur les STR.

En fait, les IOD (chapitre3) ont prouvé une nature hiérarchique dans leur modélisation. De plus, ils ont défini un intérêt particulier lors de la vérification automatique à travers l'intégration de différentes variantes d'interaction. Par conséquent, il semble inefficace de représenter tout le comportement d'un large système à travers un seul diagramme ; puisqu'il prendra l'image d'une boîte noire, donnant à l'analyste l'embarras de travailler sur plusieurs niveaux d'abstraction.

Dans [BMMR10], les auteurs ont traduit les éléments d'un IOD en termes de logiques temporelles. Par une analyse des travaux de [AMCN09], les auteurs ont proposé une simple contribution à la formalisation et à la vérification des IOD et des diagrammes d'activité à base des RdP temporels. Dans [BTB07, BT09], les auteurs ont formellement défini respectivement les IOD à base des HCPN et les diagrammes d'activité partitionnés à base des

CPN afin de modéliser les échanges des messages et des signaux, utilisant CPN Tools. Les propriétés à vérifier ont été exprimées utilisant la logique CTL et dérivées des invariants OCL.

Les dernières approches analysées se sont intéressées essentiellement à la vérification des IOD et de ses variantes dans un seul niveau hiérarchique, ce qui n'exploite pas les performances de ces diagrammes et de plus, leurs nœuds d'interaction font référence seulement aux diagrammes de séquence.

Dans l'approche que nous proposons et tout d'abord, nous essayons d'étendre ces derniers travaux toute en considérant une hiérarchie multi-niveaux, nous proposons une première approche hiérarchique multi niveaux de formalisation des IOD à base des HCPN, utilisant les TD (chapitre4) comme des nœuds d'interaction **(contribution1)**.

Notre première contribution est définie par une étude comparative avec les travaux similaires et est donnée comme suit par le tableau 2.1 :

TABLE 2.1 – Etude comparative des travaux **(contribution 1)**

| **Contribution** | **RdP hiérarchique** | **Model Checker CPN Tools** | **Approche multi-niveaux** | **Proposition d'interaction temporelle** |
|---|---|---|---|---|
| [AMCN09] | Oui | Non | Non | Non |
| [BT09] | Oui | Oui | Non | Non |
| [LJB13b] | Oui | Oui | Oui | Oui |

Dans une seconde étape et vu qu'aucune approche n'a eu les capacités d'exploiter les performances des TD dans le contexte hiérarchique des IOD, nous proposons deux approches de formalisation pour la vérification des TD à base de classes temporelles de RdP (TPN, TCPN). A ce niveau, nous modélisons les contraintes temporelles à base du profil UML MARTE **(contribution2)**.

Par la suite, nous cherchons à approfondir les spécifications par la proposition d'une formalisation d'invariants OCL dans son extension temporelle notée $OCL^{TR}$ en termes de la logique temporelle TCTL et TPN-TCTL (le résumé des travaux similaires est donné dans le chapitre5). Dans ce contexte, nous proposons la description de ces contraintes sur les diagrammes UML utilisés ainsi que la description des contraintes temporelles utilisant le profil UML MARTE, appliquées sur les TD dans une structure hiérarchique représentée par un IOD. Le model checker de Roméo est utilisé pour générer les traces de résultats d'analyse et de vérification sur les modèles TPN générés **(contribution3)**.

## 2.7 Conclusion

Tout au long de ce chapitre, nous avons donné un aperçu sur les principales logiques temporelles et les techniques de vérification utilisées dans la littérature. Ensuite, nous avons introduit les différents travaux de vérification formelle combinant le langage UML et les RdP, nous avons accentué sur celles utilisant la modélisation des STR et les nouveaux profils UML TR définis. Par la suite, nous avons synthétisé ces approches à travers une étude comparative avec les différents formalismes que nous comptons proposer.

Dans le prochain chapitre, nous mettrons en pratique notre première contribution, nous ferons part à une approche hiérarchique de formalisation pour la vérification des IOD.

# Chapitre 3

# Formalisation des IOD en HCPN

**Sommaire**



## 3.1 Introduction

Dans ce chapitre, nous présentons notre première contribution qui consiste en une approche hiérarchique de formalisation pour la vérification des IOD à base des HCPN. Dans une première étape, nous définissons formellement les concepts de base de ces modèles utilisés, ensuite, nous décrivons leurs formalismes hiérarchiques. Dans une seconde étape, nous donnons leurs différentes règles de transformation. Enfin, nous appliquons notre approche sur une partie du système DAB (Distributeur Automatique de Billets), afin de vérifier l'exécution du modèle. D'une vue générale, le contexte de cette contribution peut être résumé comme suit (figure 4.1) :

## 3.2 Concepts de base

Un IOD [omga] représente un élément global de la modélisation dynamique UML. Il dispose d'un pouvoir très expressif, lui permettant de montrer l'interaction des composants d'un système dans un haut niveau d'abstraction. Un nœud d'interaction d'un IOD peut jouer le rôle soit d'une interaction faisant référence vers un diagramme existant (appelée interaction use) et est marquée sur le coin gauche $Ref\langle nom - diagramme\rangle$, soit d'une interaction complètement modélisée (appelée inline interaction) et est marquée sur le coin

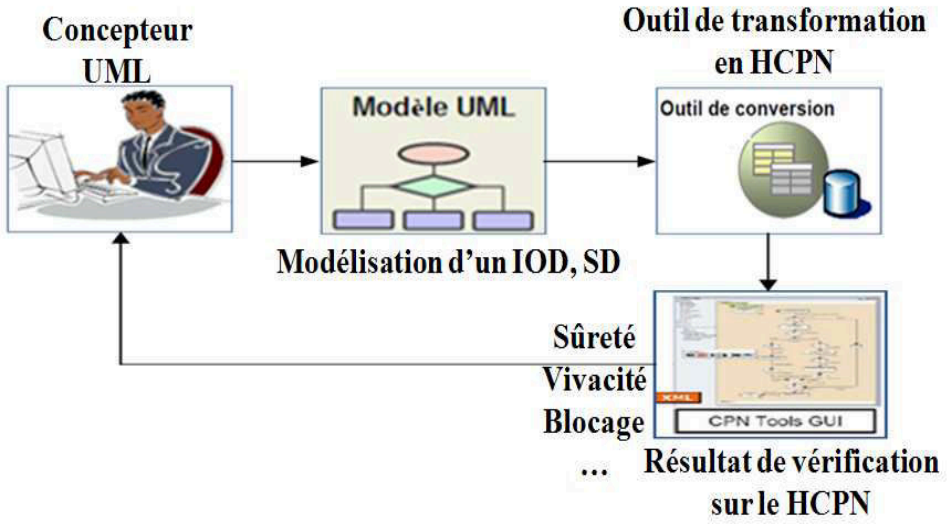



FIGURE 3.1 – Contribution 1 : Contexte spécifique et applications

gauche par $SD\langle nom - diagramme\rangle$ ou $TD\langle nom - diagramme\rangle$. La forte similitude et le caractère hiérarchique qu'offre un IOD avec un HCPN [BT09] nous ont poussé à les combiner ensemble dans notre premier travail.

### 3.2.1 Formalisation des IOD et des SD

Les IOD définissent une combinaison d'interaction suivant un processus workflow. Ils cherchent à mettre en évidence une vue globale du flux de contrôle dans une hiérarchie à plusieurs niveaux et ce, pour faciliter la description des systèmes complexes. Les IOD sont largement inspirés des diagrammes d'activité UML2 [VV13], fusionnant de différentes variantes d'interaction. Par conséquent, les IOD admettent la même forme dans certains éléments d'activité (état initial, fourche, jointure, etc.), mais avec une sémantique différente dans leur interprétation [omga]. Cependant, ils partagent certains aspects comme les scénarios séquentiels et les scénarios parallèles, les mêmes notations des éléments de modélisation, un nœud initial sous forme d'un cercle plein, un ou plusieurs nœuds finaux, un nœud de jointure, un nœud de fourche, etc. Par la figure 3.2, nous ilustrons graphiquement le modèle d'un IOD où sa notation utilisée intègre un ensemble de diagrammes d'interaction avec un état ou nœud initial, un état ou nœud final, des nœuds de contrôle pour la jointure, le parallélisme, la décision et la fusion. Entre autre, les nœuds d'interaction sont supposés des composantes primordiales alors que d'autres éléments tels qu'une contrainte de durée peut lui être affectée.

Sémantiquement, les IOD ne donnent pas de spécification quant à l'interprétation exacte des flots de contrôle, chaque nœud d'interaction doit être interprété dans le contexte d'une

interaction et non d'une activité au niveau syntaxique.

Dans [BT09], les auteurs ont formellement défini les IOD comme suit :

**Definition 3.1** $M_{IOD} = (n_0, Nf, I, B, D, E$ et $Ed)$, où :

- $n_0$ représente un nœud initial (état initial),
- $Nf = nf_1, .., nf_n$ représente un ensemble fini des nœuds finaux,
- I = $\{in_1, .., in_n\}$ représente un ensemble fini de nœuds d'interaction,
- B = $\{b_1, .., b_n\}$ représente un ensemble fini de nœuds de jointure et de fourche,
- D = $\{d_1, .., d_n\}$ représente un ensemble fini de nœuds de décision et de fusion,
- E = $\{e_1, .., e_n\}$ représente un ensemble fini de liaisons pour connecter les nœuds d'un IOD,
- Ed = $n_0 \cup I \cup B \cup D \times NF \cup I \cup B \cup D \rightarrow E$ représente une fonction reliant les nœuds d'un IOD aux liaisons définies,

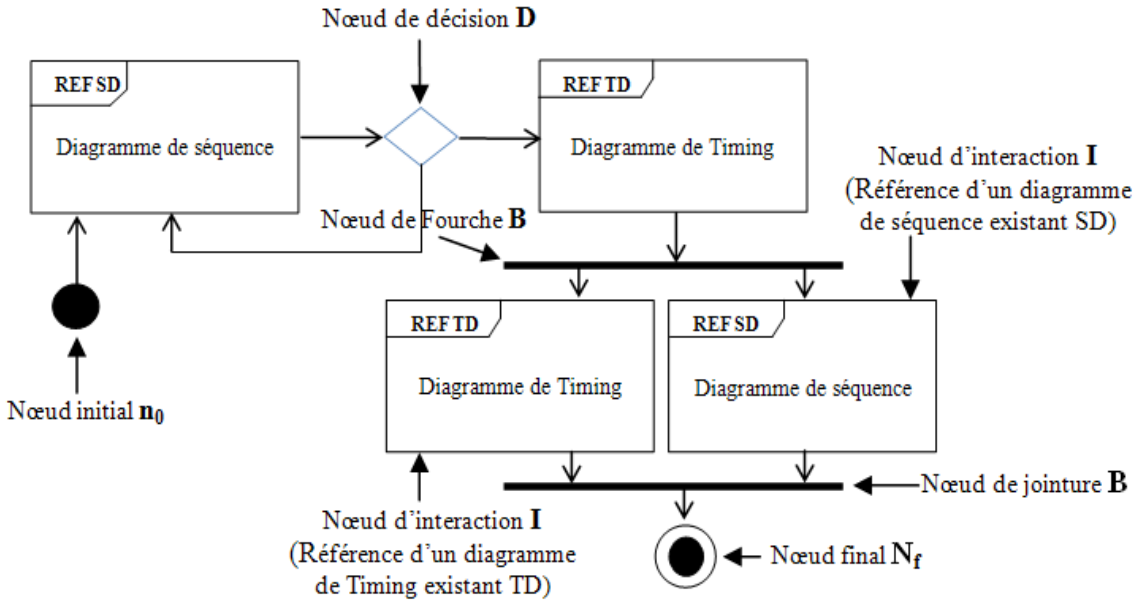



FIGURE 3.2 – Exemple d'un diagramme global d'interaction

Notament, les mêmes auteurs ont formellement défini les diagrammes de séquence (SD) comme suit :

**Definition 3.2** $M_{SD} = (Lf, Msg, Beg, End, Ptx, Find, Lost, Alt, Op, Par, Loop, In, Out$ et $Str)$, où :

- $Lf = \{lf_1, .., lf_n\}$ représente un ensemble fini de lignes de vie,
- $Msg = \{msg_1, .., msg_n\}$ représente un ensemble fini de messages échangés entre les lignes de vie,

- $Beg = \{beg_1, .., beg_n\}$ représente un ensemble fini de points d'interaction dans une ligne de vie débutant un message,
- $End = \{end_1, .., end_n\}$ représente un ensemble fini de points d'interaction dans une ligne débutant un message asynchrone,
- $Ptx = \{ptx_1, .., ptx_n\}$ représente un ensemble fini de points d'interaction dans une ligne débutant un message synchrone,
- $Find = \{f_1, .., f_n\} \rightarrow Msg$ représente un sous-ensemble des messages trouvés,
- $Lost = \{l_1, .., l_n\} \rightarrow Msg$ représente un sous-ensemble des messages perdus,
- $Alt = \{alt_1, .., alt_n\}$ représente un ensemble fini des fragments combinés alternatifs,
- $Op = \{op_1, .., op_n\}$ représente un ensemble fini des fragments combinés optionnels,
- $Par = \{par_1, .., par_n\}$ représente un ensemble fini des fragments combinés parallèles,
- $Loop = \{loop_1, .., loop_n\}$ représente un ensemble fini des fragments combinés itératifs,
- $In : Msg \rightarrow Beg$ représente une fonction qui renvoie le point d'interaction relevant de l'émission d'un message à partir d'une ligne de vie,
- $Out : Msg \rightarrow End$ représente une fonction qui renvoie le point d'interaction relevant de la réception d'un message à une de vie,
- $Str : End \cup Ptx \rightarrow Beg$ représente une méthode qui renvoie pour chaque point d'interaction les points de départ et d'arrivée d'un message,

### 3.2.2 Les HCPN

Les HCPN [Jen97] représentent des RdP de haut niveau analysés sous CPN Tools. Par ailleurs, ils définissent des modèles dans lequel une partie peut être représentée par une transition de substitution et ce, pour faire une abstraction d'un autre modèle, marqué comme un sous-réseau. Cette transition notée $TS$ (figure 3.3) est détaillée à part. La hiérarchie sert à subdiviser un modèle en différentes sous-parties, ce qui autorise une modélisation modulaire.

**Definition 3.3** $M_{HCPN} = (Pg, P, T, SubTr, A, C, Pre, Post, Pl, Tr, TrPg$ et $M_0)$, où :

- $Pg = \{pg_0, pg_1, .., pg_i\}$ représente un ensemble fini de pages $\backslash pg_0$ est la page principale,
- $P = \{p_0, p_1, .., p_i\}$ représente un ensemble fini de places,
- $T = \{ts_0, ts_1, .., ts_i)$ représente l'ensemble des transitions $\backslash (P \cap T = \emptyset)$,
- $SubTr = \{subtr_0, subtr_1, .., subtr_i\}$ représente l'ensemble des transitions de substitution,
- $A \subseteq P \times T \cup T \times P$ représente l'ensemble fini des arcs,

— $C = \{c_1, .., c_i\}$ représente l'ensemble des couleurs,
— $Pre = P \times T \rightarrow P(C)$ représente la pré-condition d'un tir d'une transition \ $Pre(p_i, t_j) = \{c_1, c_2, c_3, .., c_k\}$,
— $Post = T \times P \rightarrow P(C)$ représente la post-condition d'un tir d'une transition \ $Post(t_i, p_j) = \{c_1, c_2, c_3, .., c_k\}$,
— $Pl : Pg \rightarrow P(P)$ représente une fonction qui renvoie l'ensemble des places d'une page,
— $Tr : Pg \rightarrow P(T)$ représente une fonction qui renvoie l'ensemble des transitions d'une page,
— $TrPg : SubTr \rightarrow Pg$ représente une fonction associant une page à une transition de substitution,
— $M_0 : P \rightarrow C$ représente la fonction de marquage initial $\backslash M_0(p_i) = \sum c_k$, $k = \{1, .., i\}$,

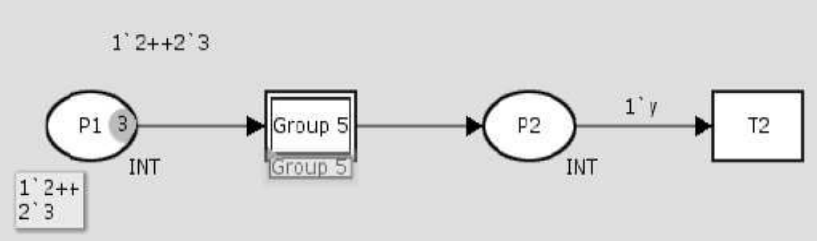



FIGURE 3.3 – Un exemple d'une transition de substitution

Chaque TS remplace toute une page avec un ensemble de places fusionnées qui est fonctionnellement identique et admettant un même marquage à tout instant. Dans la figure 3.3, $T2$ représente une transition puits qui retire un jeton y de $P2$. Quant à la transition Group5, elle fait référence à une TS, à laquelle est associée une sous-page au même nom. $Group5$ représente un CPN possédant deux places notées $P1 - 1$ et $P2 - 1$. Ces deux dernières doivent admettre les mêmes types que $P1$ et $P2$ et doivent avoir le même nombre de jetons. Les places $P1$ et $P1 - 1$ respectivement $P2$ et $P2 - 1$ sont appelées des places fusionnées. $P1$ et $P2$ représentent les ports d'entrée et de sortie et sont étiquetées respectivement **in** et **out**, associées à la transition $Group5$. Par analogie à cette définition formelle proposée, nous notons que :

— $Group5 \in SN$,
— $(P1, P2) \in PN$,
— $(InP1, OutP1, InP2, OutP2) \in PT$,
— La fonction d'interconnexion entre les places fusionnées et la $TS \in PA$, ainsi que la page qu'elle représente,
— $(P1, P1 - 1) \in FS$,

— $(P2, P2-1) \in FS$,

## 3.3 Approche hiérarchique de formalisation des IOD

Afin de décrire formellement nos modèles, nous nous focalisons sur la hiérarchie de la formalisation des IOD en termes d'éléments HCPN. Dans ce sens, nous proposons une première approche de formalisation d'un IOD à plus qu'un niveau hiérarchique. Nous utilisons de même les TD (chapitre4) pour capter les aspects temporels des interactions dans la structure hiérarchique et démontrer leur cohésion entre les différents composants du système.

### 3.3.1 Définition formelle des IOD hiérarchiques

Les IOD ont été définis naturellement par leur caractère hiérarchique [omgd], nous les décrivons formellement d'une façon fidèle à leur nature. Dans ce sens, nous proposons une transformation de ses différents éléments en termes d'éléments d'un HCPN. Pour commencer, prenons un modèle UML $M$ composé d'un ensemble d'IOD, un ensemble de SD et un ensemble de TD, tous reliés d'une manière hiérarchique comme ci-dessous défini :

**Definition 3.4** $M = M_{IOD} \cup M_{SD} \cup M_{TD}$ tel que :

- $M_{IOD}$ représente un ensemble fini d'IOD,
- $M_{SD}$ représente un ensemble fini de SD,
- $M_{TD}$ représente un ensemble fini de TD (la formalisation des TD sera traitée dans le prochain chapitre),

Dans un premier lieu, nous définissons formellement un ensemble d'IOD comme suit :

**Definition 3.5** $M_{IOD} = (S_{IOD}, Ni, NF, I, B, D, E, IODcomp, Ed,$ et $Ref)$, où :

- $S_{IOD} = \{IOD_0, .., IOD_i\}$ représente un ensemble fini d'occurrences des IOD,
- $Ni = \{ni_1, .., ni_n\}$ représente un ensemble fini de nœuds initiaux,
- $NF = \{nf_1, .., nf_i\}$ représente un ensemble fini de nœuds finaux,
- $I = \{in_1, .., in_i\}$ représente un ensemble fini des nœuds d'interaction,
- $B = \{b_1, .., b_i\}$ représente un ensemble fini des nœuds de jointure et de fourche,
- $D = \{d_1, .., d_i\}$ représente un ensemble fini des nœuds de décision et de fusion,
- $E = \{e_1, .., e_i\}$ représente un ensemble fini d'interconnections des éléments d'un IOD,
- $IODcomp : S_{IOD} \rightarrow partition(Ni \cup NF \cup I \cup B \cup D)$ représente une fonction qui associe pour chaque IOD son nœud initial, ses sous-ensembles de nœuds d'interaction, de jointure, de fourche, de décision et de fusion dans un niveau n $\backslash n > 0$,

— $E \rightarrow Ni \cup I \cup B \cup D \times NF \cup I \cup B \cup D$ est une application connectant les différents nœuds d'un IOD,
— $Ref : I \rightarrow (M_{IOD} \cup M_{SD} \cup M_{TD})$ est une fonction injective qui associe pour chaque nœud d'interaction et respectivement, un modèle $M_{IOD}, M_{SD}$ ou $M_{TD} \backslash \exists$ un seul IOD $IOD_i \backslash Ref^{-1}(IOD_i) = \emptyset$,

Dans la définition du modèle $M$, chaque nœud d'interaction fait référence à une seule interaction (SD, TD ou un IOD). Chacune d'elle doit être transformée exactement vers un élément moyennant la fonction Ref. Cette dernière capte la structure hiérarchique du modèle, qui associe pour chaque nœud son diagramme correspondant. Si la fonction Ref référence un IOD, alors la fonction IODcomp renvoie tous les éléments constituant ce modèle. Nous déduisons dans un premier lieu que Ref est une fonction injective, car la page d'un IOD principal $IOD_0$ ne sera jamais référencée. Dans un deuxième lieu, l'image de Ref et son co-domaine Ref ne sont pas égaux, la fonction Ref est aussi surjective. Ensuite et afin de définir les règles de transformation du modèle $M$ vers un modèle $M_{HCPN}$, nous définissons les diagrammes ainsi que les différents nœuds dans des niveaux hiérarchiques. D'une façon récursive, nous définissons dans un premier lieu le niveau hiérarchique n d'un IOD tel que $n \in N$ comme ci-dessous défini :

**Definition 3.6** Un IOD $IOD_i$ est dans un niveau hiérarchique n $\backslash n > 0$, si et seulement si $Ref^{-1}(IOD_i)$ appartient à un IOD d'un niveau hiérarchique n-1. L'IOD $IOD_i \backslash Ref^{-1}(IOD_i) = \emptyset$ est propre au niveau hiérarchique $i = 0$.

Ensuite, nous essayons de définir le niveau hiérarchique d'un nœud d'interaction $in$ d'un SD et d'un TD, proposés respectivement comme suit :

**Definition 3.7** Un nœud d'interaction $in$ d'un niveau hiérarchique $n \backslash n \in N$, si et seulement si, $in \in IOD_n$.

**Definition 3.8** Un diagramme de séquence $SD_j$ d'un niveau hiérarchique n, si et seulement si, $Ref^{-1}(SD_j) = IOD_{n-1}$ (niveau hiérarchique n-1).

**Definition 3.9** Un diagramme de timing $TD_j$ d'un niveau hiérarchique n, si et seulement si, $Ref^{-1}(TD_j) = IOD_{n-1}$ (niveau hiérarchique n-1).

### 3.3.2 Transformation des IOD hiérarchiques en HCPN

La construction des CPN peut s'hiérarchiser en éléments plus petits grâce à l'utilisation des TS et le raffinement de ces formalismes avec des couches multiples de détails. Une TS d'un réseau simplifié donne un aperçu général du système en substituant les transitions des sous-réseaux, nous apercevons plus de détails que nous pouvons intégrer dans le modèle. Par conséquent, l'idée derrière la théorie des HCPN est de permettre la construction d'un

grand modèle, en utilisant un certain nombre de sous-réseaux CPN reliés les uns aux autres d'une manière bien définie. Dans notre travail, nous nous inspirons de la définition formelle proposée dans [BT09], nous définissons formellement les HCPN hiérarchiques comme suit :

**Definition 3.10** $M_{HCPN} = (Pg, P, T, A, C, Pre, Post, Pl, Trord, Trsub, TrPg$ et $M_0)$, où :

- $Pg = \{pg_0, pg_1, .., pg_i\}$ représente un ensemble fini de pages $\backslash pg_0$ est la page principale,
- $P = \{p_0, p_1, .., p_i\}$ représente un ensemble fini de places,
- $T = Ts \cup SubTr = \{t_0, t_1, .., t_i\}$ représente un ensemble fini de toutes les transitions $\backslash P \cap T = \emptyset$ et où :
  - $Ts = \{ts_0, ts_1, .., ts_i)$ représente l'ensemble des transitions ordinaires,
  - $SubTr = \{subtr_0, subtr_1, .., subtr_i\}$ représente l'ensemble des TS,
- $A \subseteq P \times T \cup T \times P$ représente l'ensemble fini des arcs,
- $C = \{c_1, .., c_i\}$ représente l'ensemble des couleurs définissant les jetons,
- $Pre = P \times T \rightarrow partition(C)$ représente la pré-condition d'un tir d'une transition $\backslash Pre(p_i, t_j) = \{c_1, c_2, c_3, .., c_k\}$,
- $Post = T \times P \rightarrow partition(C)$ représente la post-condition d'un tir d'une transition $\backslash Post(t_i, p_j) = \{c_1, c_2, c_3, .., c_k\}$,
- $Pl : Pg \rightarrow partition(P)$ représente une fonction qui renvoie l'ensemble des places d'une page,
- $Trord : pg \rightarrow partition(Ts)$ représente une fonction qui renvoie l'ensemble des transitions ordinaires d'une page,
- $Trsub : pg \rightarrow partition(subtr)$ représente une fonction qui renvoie l'ensemble des TS d'une page,
- $TrPg(subtr, pg)$ représente une fonction associant une page à une TS,
- $M_0 : P \rightarrow C$ représente la fonction de marquage initial $\backslash M_0(p_i) = \sum c_k$, $k = \{1, .., i\}$,

Au niveau de notre transformation, un IOD renferme une page principale du HCPN $Pg_0$, tous les nœuds d'interaction sont transformés en termes de sous-pages HCPN, propre à un ensemble de transitions de substitution. Au moment de la création de la page, les paramètres d'entrée et de sortie sont pris en charge, le sous-réseau dérivant un SD ou un TD montre la fin d'une branche de l'hiérarchie d'un IOD. Chaque sous-page crée est débutée par des transitions In et Out appelées aussi In-transition et Out-transition. Afin de procéder à la transformation de notre modèle, les travaux de [BT09] ont proposé une fonction $\Omega$ dérivant les éléments de modélisation d'un IOD donné en termes d'éléments HCPN. Nous donnons cette fonction comme ci-dessous définie (équation 3.1) :

$$\Omega : n_0 \cup NF \cup B \cup D \cup I \cup E \rightarrow Pg \cup P \cup T \cup SubTr \cup A \cup C \tag{3.1}$$

Cette fonction n'est pas applicable où le modèle $M$ d'un système est composé d'un ensemble d'IOD, d'un ensemble de SD et d'un ensemble de TD. Ici, nous nous inspirons fortement de la fonction $\Omega$ et nous proposons une redéfinition avec une certaine modification, une nouvelle fonction $\Omega_H$. Dans ce cas, nous ne transformons pas directement un IOD vers une page principale HCPN, mais nous appellons un IOD hiérarchique de niveau 0. Entre autre, tous les autres IOD sont référencés d'une manière hiérarchique à base d'un niveau n $\backslash n > 0$ et sont transformés en termes de sous-pages HCPN. Les interactions référençant un SD et un TD du modèle source se transforment de même en termes de pages de niveau hiérarchique n. La fonction $\Omega_H$ prendra en charge le niveau hiérarchique du diagramme. Nous décrivons cette fonction comme ci-dessous défini (équation 3.2) :

$$\Omega_H : S_{IOD} \cup Ni \cup NF \cup B \cup D \cup I \cup E \rightarrow Pg \cup P \cup Ts \cup SubTr \cup A \cup C \tag{3.2}$$

Par les tableaux 3.1 et 3.2, nous donnons respectivement les différentes règles de transformation ainsi que les liaisons connectant les éléments d'un IOD. Les correspondances entre les IOD sont obtenues en appliquant la fonction $\Omega_H$. Les transitions dérivées des nœuds de fourche ou de jointure sont sémantiquement respectées, ainsi que leurs pré et post-conditions. Les sous-réseaux dérivés des SD en résultent de l'application de la fonction $\Omega$ et de la fonction de translation $\Phi$. Dans ce sens, nous définissons une nouvelle fonction prenant en charge la transformation des éléments d'un TD (fonction $\theta$ qui sera traitée dans le chapitre 4). Comme suit, nous décrivons la fonction $\Phi$ (équation 3.3).

$$\Phi : Lf \cup Msg \cup Beg \cup End \cup Ptx \cup Find \cup Lost \cup Alt \cup Op \cup Par \cup Loop \longrightarrow Pg \cup P \cup T \cup A \cup C \tag{3.3}$$

La fonction $\Phi$ est prise sans aucune modification, à titre d'exception la description des fonctions $Pl$, $Trord$ et $Trsub$, modifiées avec l'ajout respectif de certaines places, transitions ordinaires et TS.

## 3.4 Exemple d'illustration et vérification du modèle

Afin de garantir le bon fonctionnement logique de notre approche, nous l'appliquons sur une partie du distributeur automatique de billets (Etape de l'authentification d'un client).

TABLE 3.1 – Règles de transformation des éléments d'un IOD en HCPN

| **Règles** | **Elément IOD** | $\Omega_H$ **:Règles de translation** | **Élément HCPN** |
|---|---|---|---|
| 1 | $IOD_n$ de niveau hiérarchique 0 | if $IOD_n$ and $level(IOD) = 0$ then $pg_0 = Create\_Page()$; | Page principale |
| 2 | $IOD_n$ de niveau hiérarchique $(n > 0)$ | if $IOD_n$ and $level(IOD) > 0$ then $pg_i = Create\_Page(); \backslash (pg_i \in Pg)$. | Page |
| 3 | Nœud d'interaction $in$ de niveau $n$ | $\forall(in \in I)$ and $(in \subseteq leveln)$, $subtr_i = Create\_SubstitutionTransition(); \backslash (subtr_i \in Subtr)$ and $(subtr_i \subseteq pg_n)$ ; $pg_i = Create\_Page(); \backslash pg_i \in Pg_{n+1}$ ; $\Omega_H(Ref(in_i)) = subtr_i$ ; $Trpg(subtr_i, pg_i)$ ; $Trsub(\Omega_H(IODcomp^{-1}(in_i))) = Trsub(\Omega_H(IODcomp^{-1}(in_i))) \cup subtr_i$ ; | Transition de substitution |
| 4 | Nœud initial $ni$ de niveau $n$ | $\forall(ni_i \in Ni)$ and $(ni_i \subseteq leveln)$, $p_i = Create\_Place(); \backslash p_i \in pg_i$ ; $Pl(\Omega_H(IODcomp^{-1}(ni_i))) = Pl(\Omega_H(IODcomp^{-1}(ni_i))) \cup p_i$ ; | Place |
| 5 | Nœud final $nf$ de niveau $n$ | $\forall(nf_i \in NF)$ and $(nf_i \subseteq leveln$ ; $p_i = Create\_Place()$ ; $\backslash p_i \in pg_i$ ; $Pl(\Omega_H(IODcomp^{-1}(nf_i))) = Pl(\Omega_H(IODcomp^{-1}(nf_i))) \cup p_i$ ; | Place |
| 6 | Nœud de jointure \ nœud de fourche $jfn_n$ | $\forall(jfn_i \in B)$ and $(jfn_i \subseteq leveln)$, $ts_i = Create\_OrdinaryTransition \backslash ts_i \in pg_i$ ; $Trord(\Omega_H(IODcomp^{-1}(jfn_i))) = Pl(\Omega_H(IODcomp^{-1}(jfn_i))) \cup ts_i$ ; | Transition |
| 7 | Nœud de fusion \ nœud de décision $mdn_n$ | $\forall(mdn_i \in D)$ and $(mdn_i \subseteq leveln)$, $p_i = Create\_Place()$ ; $\backslash p_i \in pg_i$ ; $Pl(\Omega_H(IODcomp^{-1}(mdn_i))) = Pl(\Omega_H(IODcomp^{-1}(mdn_i))) \cup p_i$ ; | Place |
| 8 | Connexion de sous-réseaux avec des in-transition | $\forall(lf_i \in Lf)$ and $(lf_i \subseteq leveln)$ ; $p_{lf} = Create\_Place()$ ; $\backslash p_{lf} \in pg_i$ ; $Pl(\Omega_H(IODcomp^{-1}(lf_i))) = Pl(\Omega_H(IODcomp^{-1}(lf_i))) \cup p_{lf}$ ; $a = Create\_Arrow(p_{lf}, t_{in})$ ; $\backslash a \in A$ ; | Place + Arc |
| 9 | Connexion de sous-réseaux avec des out-transition | $\forall(lf_i \in Lf)$ and $lf_i \subseteq leveln)$, $p_{lf} = create\_Place()$ ; $\backslash p_{lf} \in pg_i$ ; $Pl(\Omega_H(IODcomp^{-1}(mdn_i))) = Pl(\Omega_H(IODcomp^{-1}(mdn_i))) \cup p_{lf}$ ; $a = Create\_Arrow(t_{out}, p_{lf})$ ; $\backslash a \in A$ | Place + Arc |
| 10 | Inter-connexion des éléments IOD | tableau 3.2 | |

TABLE 3.2 – Règles de transformation des liaisons

| **Règles** | **Nœud source de la transition** | **Nœud destination de la transition** | $\Omega_H$ **:Règles de translation** | **Élément HCPN** |
|---|---|---|---|---|
| 10.1 | Initial Fusion Décision | Interaction Jointure Fourche | $\forall e = Ed(i,j) \backslash (i \in (N_i \cup D)) \wedge (j \in (I \cup B))$ then $a = Create\_Arrow()$ ; $\backslash a = ((\Omega(i), \Omega(j)) \wedge (a \in A))$ | Arc |
| 10.2 | Initial Fusion Décision | Fusion Décision Final | $\forall e = Ed(i,j) \backslash (i \in (N_i \cup D))$ and $(j \in (D \cup NF))$ then $ts = Create\_OrdinaryTransition()$ ; $\backslash ts \in T$ ; $Tr(\Omega_H(Ref(in))) = Pl(\Omega_H(Ref(in))) \cup t_s$ ; $a_i = Create\_Arrow()$ ; $\backslash a_i = ((\Omega(i), t_s) \wedge (a_i \in A))$ ; $a_j = Create\_Arrow()$ ; $\backslash a_j = ((t_s, \Omega(i)) \wedge (a_j \in A)$ ; | Arc + Transition + Arc |
| 10.3 | Interaction Jointure Fourche | Fusion Décision Final | $\forall e = Ed(i,j) \backslash (i \in (I \cup B))$ and $(j \in (D \cup NF))$ then $a = Create\_Arrow()$ ; $\backslash a = (\Omega(i), \Omega(j)$ and $(a \cup A))$ | Arc |
| 10.4 | Interaction Jointure Fourche | Interaction Jointure Fourche | $\forall e = Ed(i,j) \backslash (i \in (I \cup B))$ and $(j \in (I \cup B))$ then $p = Create\_Place()$ ; $\backslash p \in P$ ; $Pl(\Omega_H(Ref(in))) = Pl(\Omega_H(Ref(in))) \cup p$ ; $a_i = Create\_Arrow()$ ; $\backslash a_i = ((\Omega(i), p)$ and $(a_i \in A))$ ; $a_j = Create\_Arrow()$ ; $\backslash a_j = (p, \Omega(i)$ and $(a_j \in A))$ ; | Place +2 Arcs |

### 3.4.1 Description de l'exemple et modélisation UML

L'authentification via le DAB peut être décrite comme suit : le client insère sa carte de crédit, saisit son code secret, le système vérifie les paramètres d'authentification. Une fois est valide, un menu s'affiche, sinon le système demande de reprendre cette opération. La figure 3.4 illustre un aperçu sur la structure du système à étudier par un diagramme de classe, décrivant les différentes relations statiques (Client, Banque, Système). Nous donnons un $IOD_0$ (IOD de niveau 0) comme illustré par la figure 3.5(a). D'une certaine façon hiérarchique, nous raffinons le comportement du nœud d'interaction par la figure 3.5(b) par un $IOD_1$ (IOD de niveau 1). Dans les diagrammes qui suivent, nous proposons trois nœuds d'interaction (Pin Test, Eject Card et Welcome Message). Ces différentes interactions sont présentées à base d'un SD ou d'un TD dans un niveau plus raffiné.

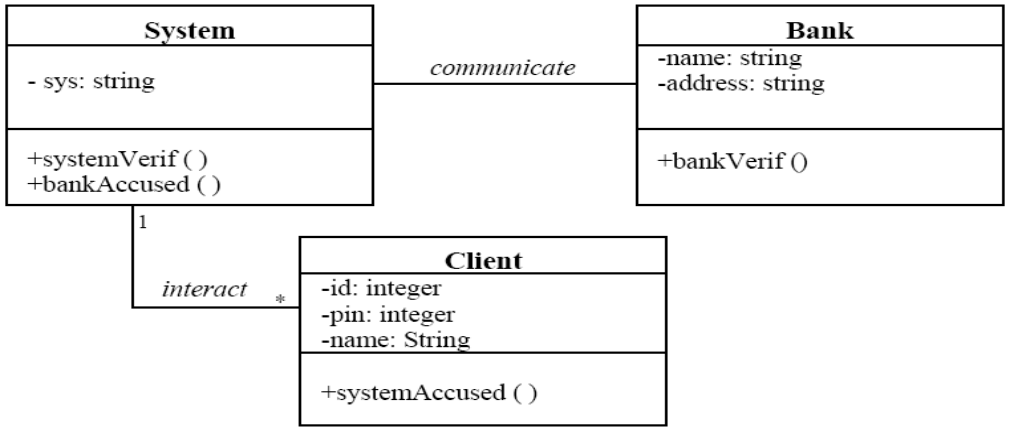



FIGURE 3.4 – Système DAB : diagramme de classe

Afin de bien décrire le comportement hiérarchique du système et les interactions possibles des différentes entités, nous modélisons par la figure 3.5 un IOD illustrant les deux niveaux hiérarchiques.

### 3.4.2 Transformation et vérification du modèle

La simulation d'un modèle demeure une contrainte pour la vérification, afin d'apporter un élément de réponse aux différentes situations que peuvent apparaître. CPN Tools nous a assuré la génération et l'analyse du modèle HCPN.

Ensuite et dans un premier lieu, la vérification des CPN sous CPN Tools nous a offert un bon moyen pour l'exploitation des propriétés génériques et spécifiques, après l'inspection du graphe d'états. Ce dernier nous a assuré l'évolution dynamique du modèle. Dans un deuxième lieu, il nous a permis de rattacher des segments de codes décrits sous le standard

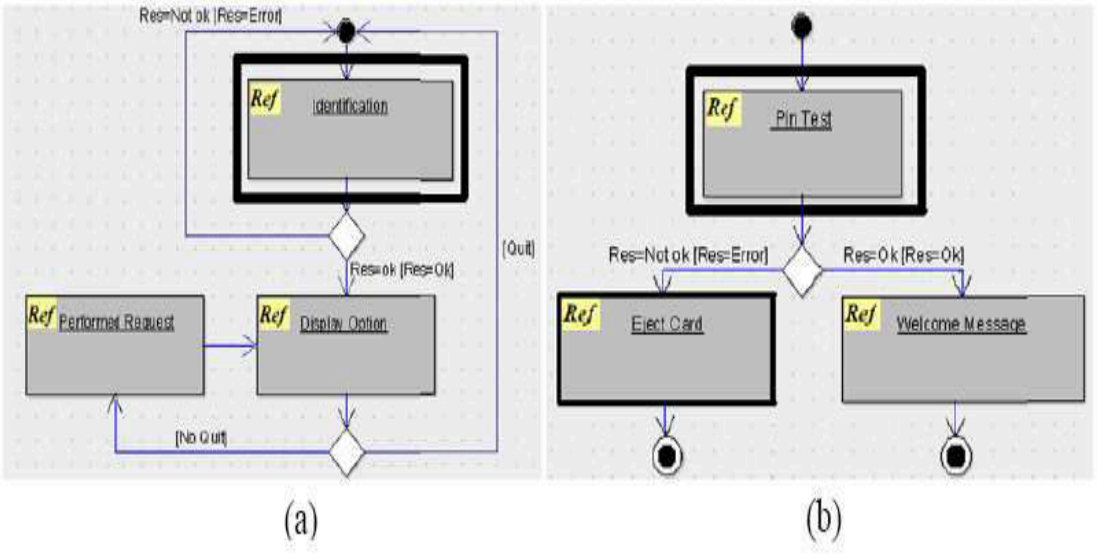



FIGURE 3.5 – Modélisation du comportement dynamique du système

ML aux transitions du modèle. La procédure de transformation des IOD en HCPN s'est basée essentiellement sur deux formats d'échange XML. Le modèle IOD est exporté dans un premier schéma XML avant d'être implémenté avec les règles de transformation sous le langage java. Ce dernier sera importé sous CPN Tools afin de pouvoir générer le modèle HCPN correspondant. Comme suit, nous illustrons le modèle HCPN généré et la trace de son exécution (figure 3.6) :

Dans ce qui suit, nous démontrons le bon fonctionnement du modèle HCPN. Nous aurons à formuler sous le standard ML (figure 3.6) les propriétés de réinitialisation, de présence de transitions mortes et du blocage pour la vérification du modèle.

1. **Test de réinitialisation :** une propriété du modèle permettant de nous confirmer, qu'à n'importe quel moment de son exécution, le modèle peut se rendre à son état initial. Sous CPN Tools, nous nous offrons la possibilité d'invoquer la requête **InitialHomeMarking()**. Le résultat approprié est à valeur booléenne. Dans le cas d'un retour réussi, le résultat fourni est **Vrai** (figure 3.6).
2. **Test de présence de transitions mortes :** Une transition morte est toute transition qui n'est franchie à aucun moment de l'exécution du modèle, où la détection de transitions mortes prouve l'existence de places non marquées. Ceci est assuré en invoquant la requête **ListDeadTIs()**. Cette dernière montre l'ensemble des transitions mortes issu du modèle.
3. **Test de blocage :** Dès que la simulation s'achève, un blocage est forcément recensé en invoquant la requête **ListDeadMarkings()**. Cette dernière renvoie la liste des

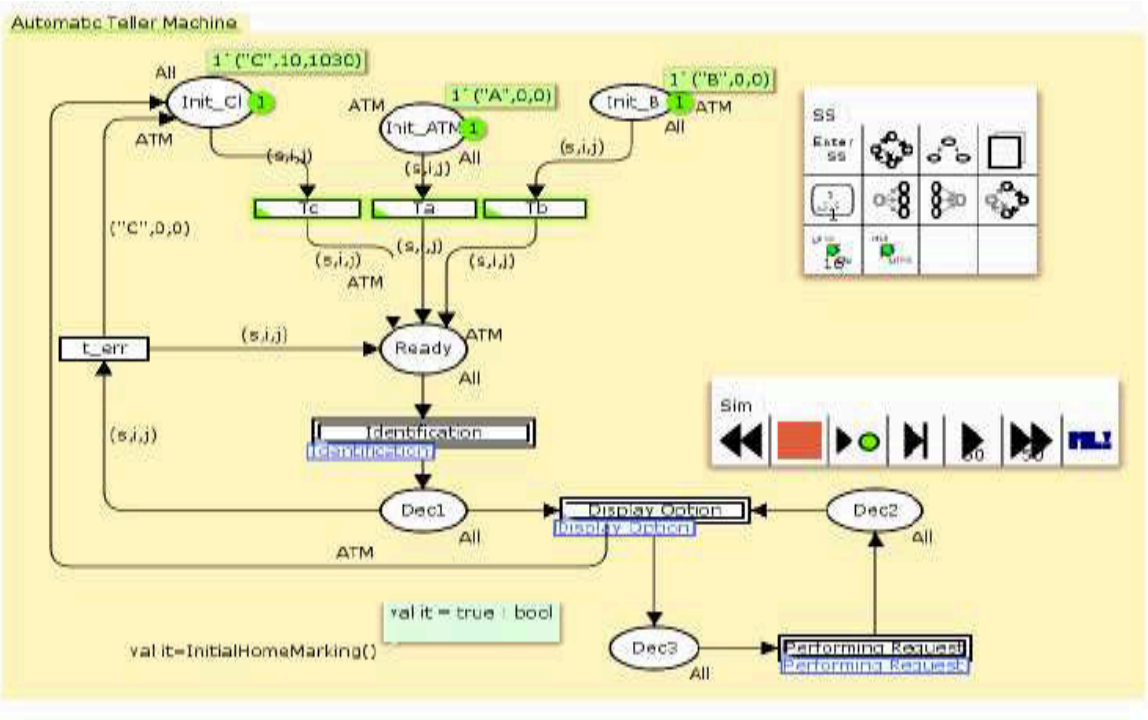



FIGURE 3.6 – Comportement du modèle HCPN généré

nœuds où le blocage est détecté. En effet, nous calculons la taille de cette liste qui restitue une valeur supérieure à zéro dans le cas d'un blocage.

## 3.5 Conclusion

Tout le long de ce chapitre, nous nous sommes focalisés essentiellement à proposer une approche hiérarchique de formalisation des IOD en HCPN. Dans un premier côté, nous les avons défini formellement et nous avons proposé un ensemble de règles de transformation de ces deux modèles. Dans un deuxième côté et afin de garantir l'efficacité de notre approche, nous l'avons appliquée sur la phase d'authentification du système DAB. Utilisant CPN Tools, nous avons analysé et vérifié notre modèle.

À ce niveau, notre approche proposée traite essentiellement la hiérarchie des IOD sans pouvoir exploiter les éléments de modélisation des TD. Dans le prochain chapitre, nous tenterons de proposer deux approches de formalisation pour la vérification des TD qui seront exploités dans un contexte temps réel.

# Chapitre 4

# Formalisation des TD en TCPN et en TPN

**Sommaire**



## 4.1 Introduction

Dans ce chapitre, nous proposons une nouvelle contribution au niveau de deux approches de formalisation pour la vérification des TD à base des RdP temporisés et temporels.

La première approche consiste en une formalisation des TD en termes des RdP colorés temporisés (TCPN). Les jeux de tests dynamiques et la vérification du modèle généré sont assurés utilisant CPN Tools. Un outil de transformation automatique est mis en place. Cette approche est appliquée sur une étude de cas (Système de contrôle d'accès dans une salle serveur). La deuxième approche consiste en une formalisation des TD en termes des RdP temporels (T-TPN ou Transition-TPN), nous les notons TPN dans la suite du rapport. Dans ce sens, une transformation des modèles est définie, la simulation et la vérification sont assurées utilisant Roméo et appliquées sur une étude de cas (Processus de chargement d'une page web). Ce dernier exemple et les propriétés désirées à vérifier seront exprimés sous la logique temporelle TPN-TCTL et seront traitées dans le chapitre5.

D'une vue générale, le contexte de ces deux approches est résumé comme suit (figure 4.1 et figure 4.2) :

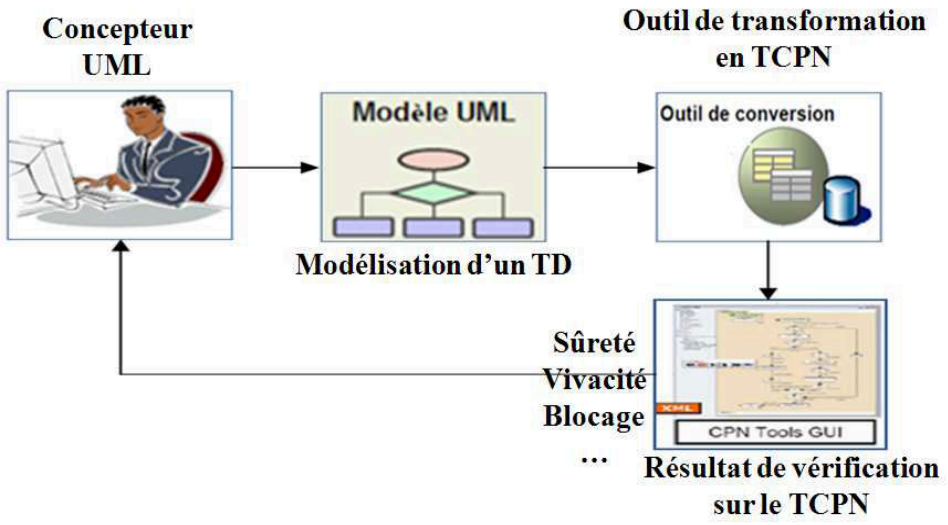



FIGURE 4.1 – Contribution 2.1 : Contexte spécifique et application

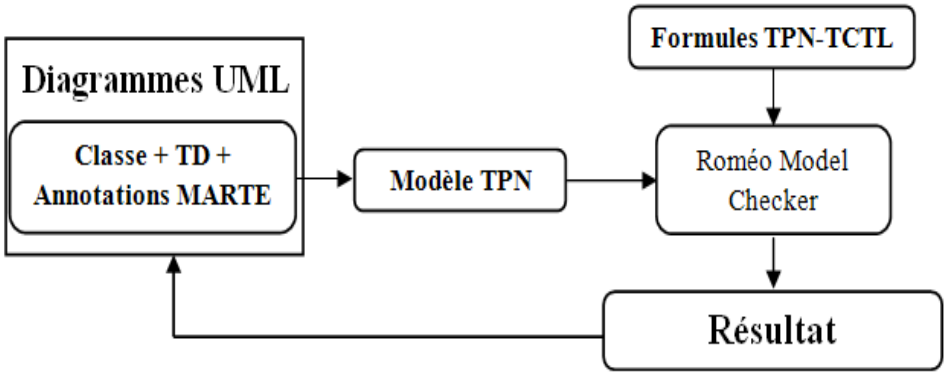



FIGURE 4.2 – Contribution 2.2 : Contexte spécifique et application

## 4.2 Concepts de base

### 4.2.1 Les TD, les TCPN et les TPN

#### 4.2.1.1 Les TD

Le diagramme de timing (TD) [omga] est nouvellement défini sous UML2 et peut se présenter dans une vue classique ou dans une vue compacte. Il dérive des techniques comprenant l'ingénierie des systèmes et les diagrammes d'interaction, combinant un diagramme de séquence et les diagrammes de machine d'état, afin de décrire les états et leurs transitions par des messages asynchrones, entre l'ensemble des objets composant le diagramme et ce, dans un contexte purement temporel.

Par conséquent, modéliser un TD revient à combiner plusieurs éléments (voir figure 4.3) à savoir les lignes de vie qui sont inversées de gauche à droite, séparées et disposées horizontalement pour l'écoulement du temps, les états, les messages, les contraintes temporelles et les contraintes de durée, nous les décrivons comme suit :

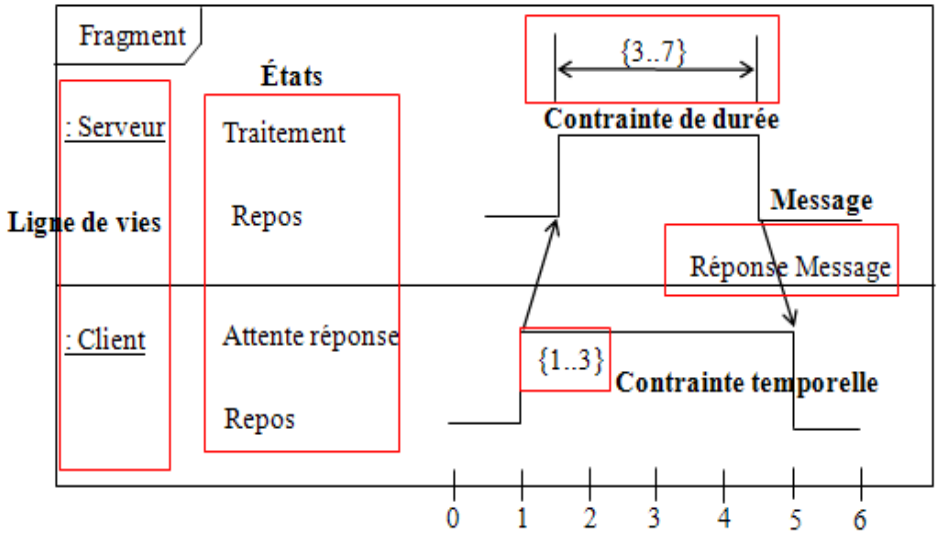



FIGURE 4.3 – Un exemple d'un diagramme de timing

- La ligne de vie : un élément représenté par une participation individuelle d'un objet dans une interaction. Elle est caractérisée par un nom et disposée horizontalement par rapport au diagramme de séquence,
- Un état : un élément représenté par un ensemble de valeurs d'attributs propre à un objet. Par conséquent, modifier l'une des valeurs des attributs d'un objet, modifie tout son état,
- Un message : un élément représenté par un lien de communication entre deux objets. Il est affiché d'une façon asynchrone au moment d'appel ou de retour,
- La contrainte temporelle : un élément représenté par la durée allouée pour qu'une transition puisse se produire,
- La contrainte de durée : un élément représenté par une durée totale sur un intervalle de temps utilisée pour maintenir et garder un état dans un objet,

Dans [Joo10], les auteurs ont formellement défini les TD comme suit :

**Definition 4.1** $M_{TD} = (Lf, Pt, Msg, State, Tc \text{ et } Dc)$, où :

- $Lf = \{lf_1, .., lf_k\}$ représente un ensemble fini de lignes de vie,
- $Pt = \{pt_1, .., pt_k\}$ représente un ensemble fini de points d'interaction entre les lignes de vie et les messages asynchrones,

— $Msg = \{m_1, .., m_k\}$ représente un ensemble fini de messages asynchrones échangés entre les objets,
— $State = \{st_1, .., st_k\}$ représente un ensemble fini des états possibles correspondants aux objets,
— $Tc = \{tc_1, .., tc_k\}$ représente un ensemble fini de temps propre à l'achèvement d'une transition,
— $Dc = \{dc_1, .., dc_k\}$ représente un ensemble fini de contraintes de durée,

#### 4.2.1.2 Les TCPN

Les TCPN ont été trop traités dans la littérature, surtout pour la modélisation des protocoles de communication. Les différents exemples manipulant les TCPN sont disponibles dans le tutorial de CPN Tools [1]. Dans ces types de modèles, la temporisation est affectée aux éléments de modélisation (place, arc, transition). Dans notre travail, nous traitons la temporisation affectée aux places, dans ce cas, le temps est représenté par un délai exact pour permettre l'occurrence d'un événement.

Dans [Boz12], les auteurs ont formellement défini les TCPN comme suit :

**Definition 4.2** $M_{TCPN} = (P, T, A, E, V, C, G, Temp, E \text{ et } If)$, où :
— $P = \{p_0, p_1, .., p_i\}$ représente un ensemble fini de places temporisées,
— $T = \{t_0, t_1, .., t_i\}$ représente un ensemble fini de transitions $\backslash P \cap T = \emptyset$,
— $A \subseteq (P \times T) \cup (T \times P)$ représente un ensemble fini d'arcs,
— $E = \{c_0, c_1, .., c_i\}$ représente un ensemble fini non vide de couleurs, où chaque couleur $c \in E$ est temporel,
— $V = \{v_0, .., v_i\}$ représente un ensemble fini de variables typées $\backslash v \in V$ et $type[v] \in E$,
— $C : P \rightarrow E$ représente une fonction des ensembles de couleurs. C assigne une couleur pour chaque place. $p \in P$ ssi, $c(p) \subseteq E$,
— $G : T \rightarrow ExpGF$ représente une fonction de garde, assignant une contrainte pour une transition t,
— $Temp = \{tm_0, tm_1, .., tm_i)$ représente un ensemble fini pris par une transition produite,
— $E : A \rightarrow EXP$ représente une fonction assignant une expression sur un arc $\backslash Type[E(a)] = C(p)$ et p est connectée à l'arc a,
— $If : P \rightarrow EXP$ représente une expression d'initialisation d'une place $\backslash [If(p) = C(p)$,

1. http ://cpntools.org/

#### 4.2.1.3 Les TPN

Les TPN [BGR09, BBB14] définissent une extension temporelle des RdP. Ils ont été conçus pour l'étude des problèmes de recouvrement au niveau des protocoles de communication. Dans les TPN, les informations temporelles peuvent être placées soit sur les transitions, le modèle TPN est appelé T-TPN (TPN), soit sur les places, le modèle TPN est appelé P-TPN ou sur les arcs et dans ce cas, le modèle TPN est appelé A-TPN.

Dans le contexte de nos travaux, nous exploitons les T-TPN pour bien modéliser les contraintes temporelles et de durée, ainsi que tous les échanges temporels entre les états des objets. Une horloge implicite qui mesure le temps écoulé à partir du déclenchement d'une transition t et un intervalle réel non négatif de temps explicite à bornes rationnelles $I_s$, peuvent être associés sur ses éléments, jouant le rôle d'une contrainte temporelle non ponctuelle. Les bornes associées représentent les temps d'attente minimal et maximal pour franchir cet élément.

Par la figure 4.4, nous présentons un exemple d'un modèle TPN. Une transition t n'est une fois déclenchée, si et seulement si, l'évaluation de son horloge appartient à $I_s$. Si t est sensibilisée de façon continue durant au moins $a$ unités de temps, elle peut être tirée. Par contre, si elle est sensibilisée par $b$ unités de temps de façon continue, elle doit être tirée à moins qu'elle ne soit désensibilisée par un autre franchissement.

Dans [BGR09], les auteurs ont formellement défini les TPN comme suit :

**Definition 4.3** $M_{TPN} = (P, T, A, W, m_0 \text{ et } I)$, où :

- $P = \{p_1, p_2, .., p_n\}$ représente un ensemble non vide de places,
- $T = \{t_1, t_2, .., t_n\}$ représente un ensemble non vide de transitions $\setminus(P \cap T = \emptyset)$,
- $A$ représente un ensemble fini d'arcs ou de liaisons $\setminus A \subseteq (P \times T) \cup (T \times P)$,
- $W : A \to N^*$,
- $m_0$ représente le marquage initial du modèle,
- $I : T \to [min, max]$ représente une contrainte temporelle où min représente le temps de franchissement minimal de la transition T et max le temps au plus tard possible de son déclenchement.

### 4.2.2 Fonctionnement et classes d'états d'un TPN

Un état d'un TPN est un marquage m associé à une fonction I, qui associe un intervalle réel positif à chaque transition sensibilisée par m. Initialement, $s_0 = (m_0, I_0)$ avec $I_0(t) = I_s(t)$ pour chaque transition t sensibilisée par $m_0$. Les bornes temporelles $Min(I(t))$ et $Max(I(t))$ désignent respectivement, les limites inférieure et supérieure d'un intervalle I.

— **Fonctionnement d'un TPN :** soit t une transition sensibilisée dans un état $s =$

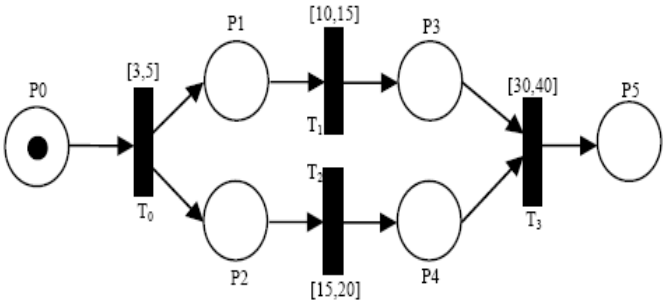



FIGURE 4.4 – Exemple d'un TPN

$(m, I)$ pour la dernière fois à une date $\beta$, t ne peut jamais être tirée depuis s avant la date $\beta + Min(I(t))$ et doit l'être au plus tard à la date $\beta + Max(I(t))$ sous contrainte d'un tir d'une autre transition désensibilise avant que celle-ci ne soit tirée. Cette règle définit sur l'ensemble des états une relation d'accessibilité temporisée notée $t@$. $s = (m, I) \rightarrow^{t@\theta} s'$ si la transition peut être tirée depuis l'état s à la date relative $\theta \backslash \theta \in I(t))$. L'ensemble des états d'un TPN est l'ensemble des états accessibles depuis l'état initial $s_0$, muni de la relation d'accessibilité temporisée $t@\theta$. Le domaine de tir d'un état $s = (m, I)$ est l'ensemble des vecteurs avec leurs composantes indexées par les transitions sensibilisées.

— **Etats et classes d'états d'un TPN :** Nous appelons l'ensemble des transitions qui peuvent être franchises à toute date dans leur intervalle temporel I. Généralement, les états d'un TPN admettent une infinité de successeurs. Toute représentation finie de l'espace d'état passe par des groupements d'états. Ces groupements sont appelés classes d'états.

## 4.3 Formalisation des TD en TCPN

Dans cette section, nous proposons la transformation des TD en termes d'élément TCPN ainsi que la vérification du modèle généré, par la suite, nous appliquons notre approche à base d'un exemple d'illustration.

### 4.3.1 Transformation

Dans un premier travail, nous nous focalisons à proposer les règles de transformation des TD en TCPN (tableau 4.1) à base de la fonction $\theta$ définie et décrite ci-dessous (équation 4.1) :

$$\theta : Lf \cup Pt \cup Msg \cup State \cup Dc \cup Tc \longrightarrow Pg \cup P \cup T \cup A \cup V \cup E \cup C \cup G \cup Temp \cup E \cup If \quad (4.1)$$

TABLE 4.1 – Règles de transformation des éléments d'un TD

| **Règles** | **Eléments TD** | **$\theta$ :Règles de translation** | **Élément TCPN** |
|---|---|---|---|
| 1 | Ligne de vie | $get(lf_i \in Lf)$ ; $p_i = Create\_Place()$ ; $\backslash p_i \in P$ ; $\theta_1 : Lf \rightarrow P$ ; $\theta_1(lf_i) = p_i$ ; | Place (un état initial) |
| 2 | Point d'interaction | $get(pt_i \in Pt)$ and $(lf \in Lf)$ ; $assign(pt_i, T_i)$ ; $\theta_2 : Pt \rightarrow (P, A, T)$ ; $\theta_2(pt_i) = assign(pt_i, T_i)$ ; | Transition |
| 3 | Message | $get(msg \in Msg) \backslash msg = (lf_1, pt_1, lf_2, pt_2)$ ; $p_m = Create\_Place()$ ; $\backslash p_m \in P$ ; $Create\_Arrow = (getTransition(pt_1), p_m)$ ; $Create\_Arrow = (p_m, getTransition(pt_2))$ ; $\theta_3 : Msg \rightarrow (A, P, A)$ ; $\theta_3(msg) = Create\_Sequence(A_{in}, P_m, A_{out})$ ; | Arc + place + Arc |
| 4 | Etat | $get(st \in State)$ and $(p \in P)$ and $(A_{in} \in A)$ ; $getColor(p, c) \backslash c(p) \subseteq E$ ; $assign(E, A_{in}, p)$ ; $\theta_4(st) = p(withcolor)$ ; | Couleur place + Expression arc |
| 5 | Contrainte de durée | $get(dc \in Dc)$ ; $a = getArrowOutTransition(tr)$ ; $Timestamp(a, uniform(x, y))$ ; $\theta_5(dc) = temp$ ; | Contrainte sur la transition de sortie |
| 6 | Contrainte temporelle | $get(tc \in Tc)$ and $(tr \in T)$ and $(p \in P)$ ; $Create\_Sequence(tr, p)$ ; $setDestArrow(getArrowIn(tr), tr)$ ; $a = Create\_Arrow(p, tr)$ ; $Timestamp(a, uniform(x, y))$ ; $\theta_6(tc) = Create\_Sequence(tr, p)$ ; | Transition + Place |

Ci-dessous, nous illustrons par les figures 4.5 et 4.6 respectivement, la transformation d'une ligne de vie en termes d'une place marquée et d'un message en termes d'une place liant la transformation de deux lignes de vie.

Sur cette approche, nous avons développé un outil automatique (3TD : TD To TCPN) sous le langage Java, utilisant des formats d'échange XML pour l'exportation d'un TD dans notre éditeur UML Visual Paradigm [2] et importer le modèle TCPN sous CPN Tools

2. http ://www.visual-paradigm.com/

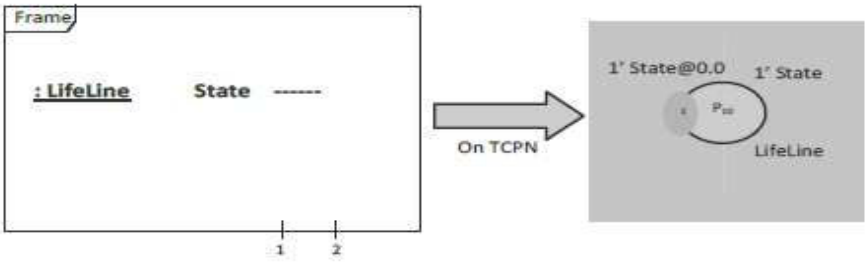



FIGURE 4.5 – Transformation d'une ligne de vie

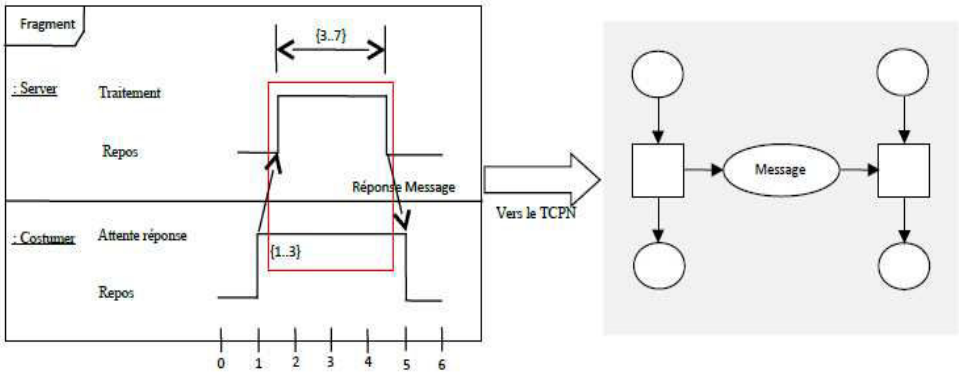



FIGURE 4.6 – Transformation d'un message

(certaines règles de transformation sont détaillées sous le langage java dans l'annexe (A.2)). Au moment de notre développement, nous avons fait appel à la bibliothèque JDOM comme source d'API Java ouverte, pour représenter et manipuler le document XML d'une façon intuitive. Dans ce sens, une machine virtuelle Java (JVM) est définie.

### 4.3.2 Exemple d'illustration et vérification du modèle

#### 4.3.2.1 Description de l'exemple et modélisation UML

Dans cette partie, nous nous intéressons à décrire un système de contrôle d'accès dans une salle serveur, affichant l'ensemble des interactions possible avec l'utilisateur. Après avoir inséré sa carte, ce dernier est invité à saisir son code d'accès, le système vérifie la validité de la carte et l'exactitude du code. Une fois authentifié, l'utilisateur est autorisé à accéder dans la salle, sinon, il est invité à retirer sa carte et relancer la procédure d'authentification.

Nous supposons que notre interaction est spécifiée par deux diagrammes (figure 4.7), un diagramme de cas d'utilisation pour la description des besoins fonctionnels et un diagramme de classe afin de donner la structure statique du système. L'interaction décrite est modélisée par un TD et est illustrée par la figure 4.8.

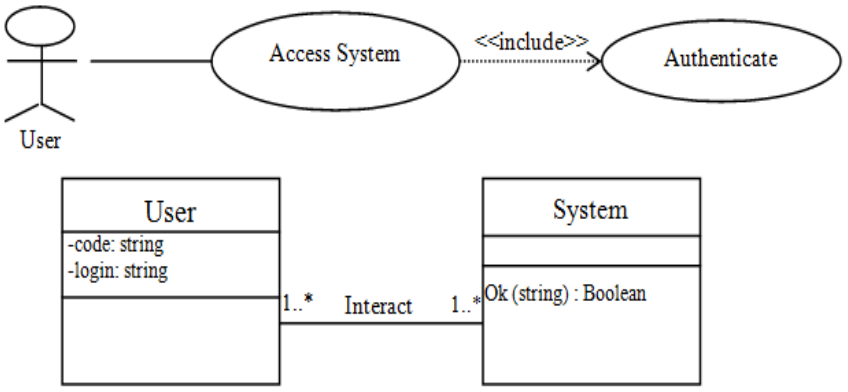



FIGURE 4.7 – Architecture de l'exemple

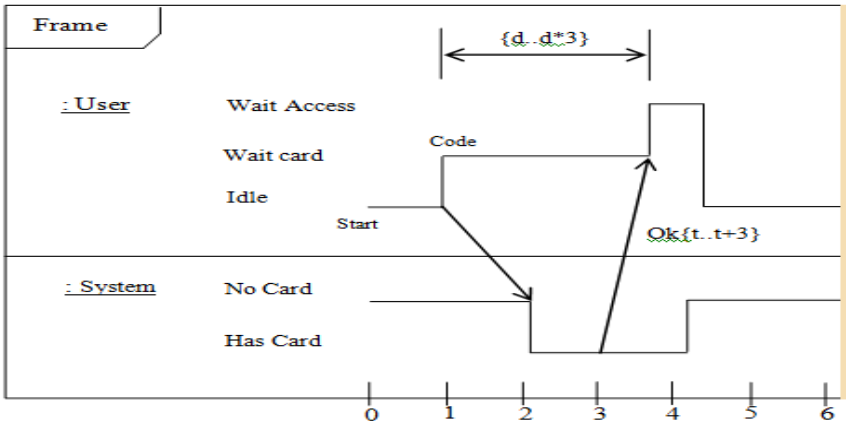



FIGURE 4.8 – Diagramme de timing de l'exemple

Par la figure 4.9, nous illustrons le modèle TCPN généré. L'évaluation des propriétés est assurée à base d'un rapport textuel, décrivant les aspects du modèle (taille du graphe, marquage, réinitialisation, présence de cycles, états finaux, transitions mortes).

#### 4.3.2.2 Vérification du modèle

Suite à l'analyse du comportement du modèle, nous nous sommes dirigés à générer l'espace d'état afin de pouvoir le vérifier. Dans ce contexte, nous avons maintenu le test de réinitialisation, de blocage et de vivacité.

1. **Test de réinitialisation :** nous invoquons la requête **InitialHomeMarking()**, comme le montre la figure 4.10, le résultat retourné est à valeur booléenne **Vrai**, signifiant le retour réussi au marquage initial, issu du n'importe quel état du modèle.
2. **Test de blocage :** Dès l'analyse du modèle est achevée, un blocage est forcément

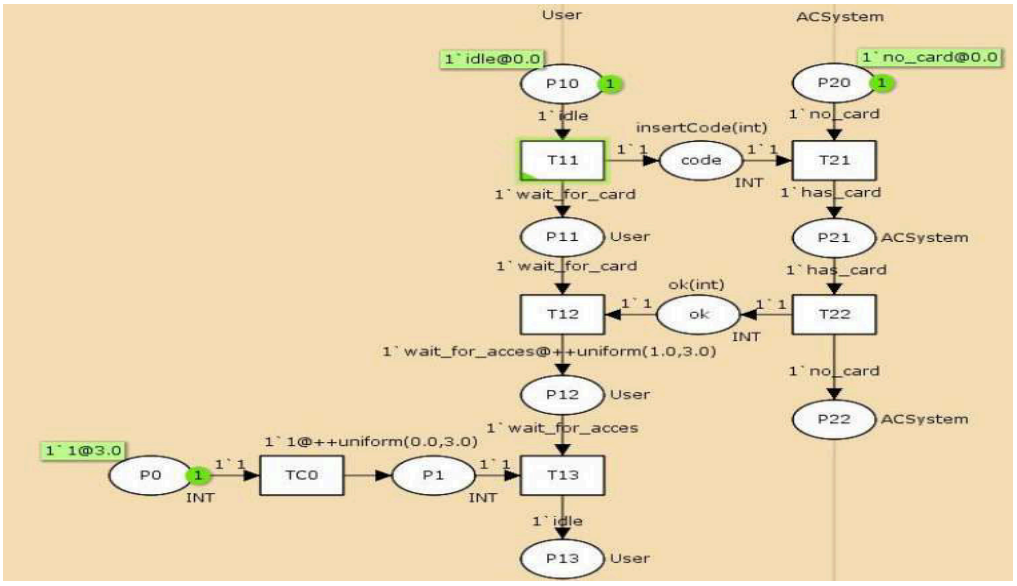



FIGURE 4.9 – Modèle TCPN généré

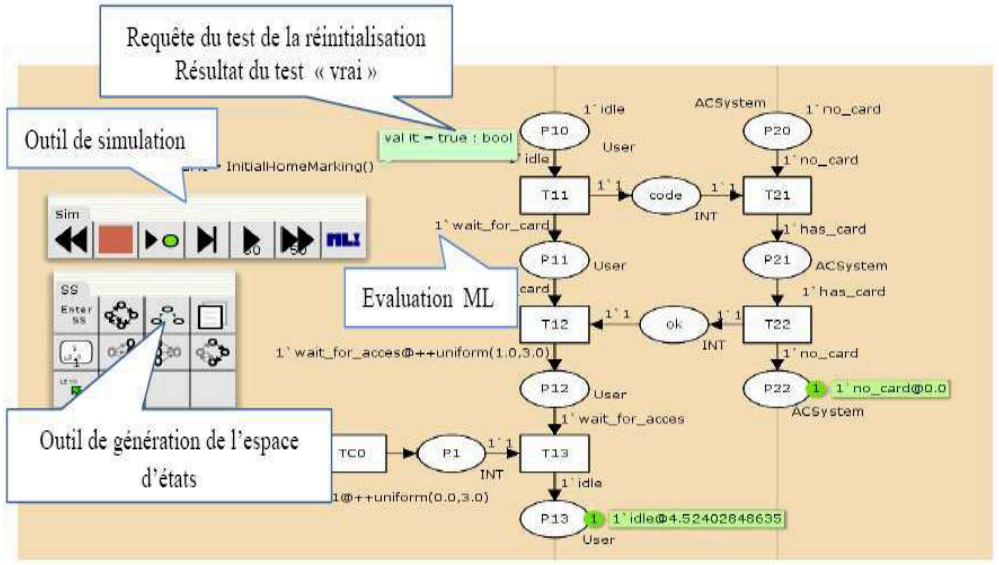



FIGURE 4.10 – Test de réinitialisation du modèle

recensé en invoquant la requête **ListDeadMarkings()** (figure 4.11). Cette dernière renvoie la liste des nœuds où le blocage sera détecté. De ce fait, nous calculons la taille de cette liste qui restitue une valeur supérieure à zéro en cas de blocage.

3. **Test de vivacité :** Nous invoquons la requête **ListDeadTIs()** (figure 4.12) afin de détecter toute transition morte dans le modèle.

Dans [BT06], les auteurs ont exprimé formellement les propriétés spécifiques et ce, par

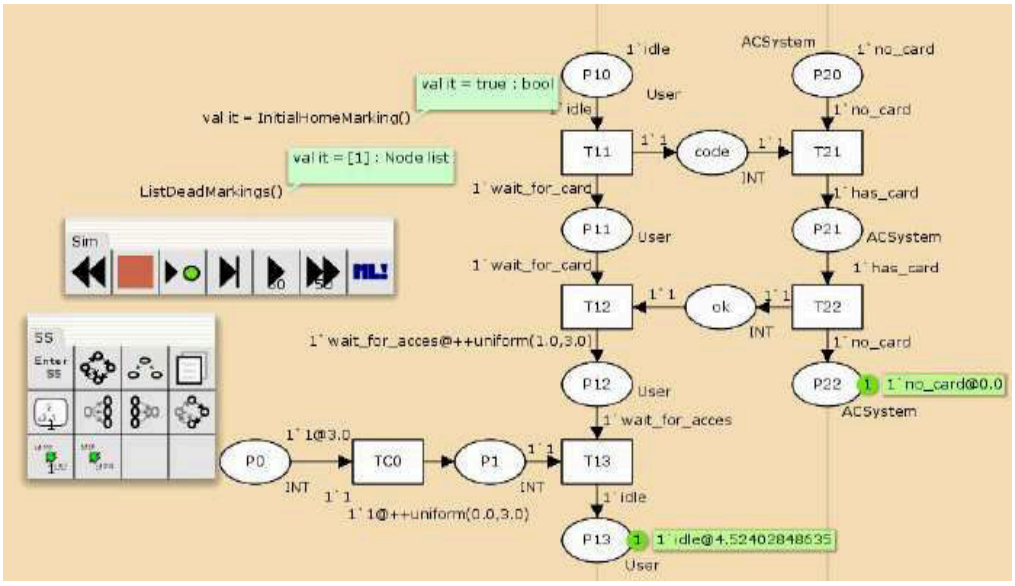



FIGURE 4.11 – Test de blocage du modèle

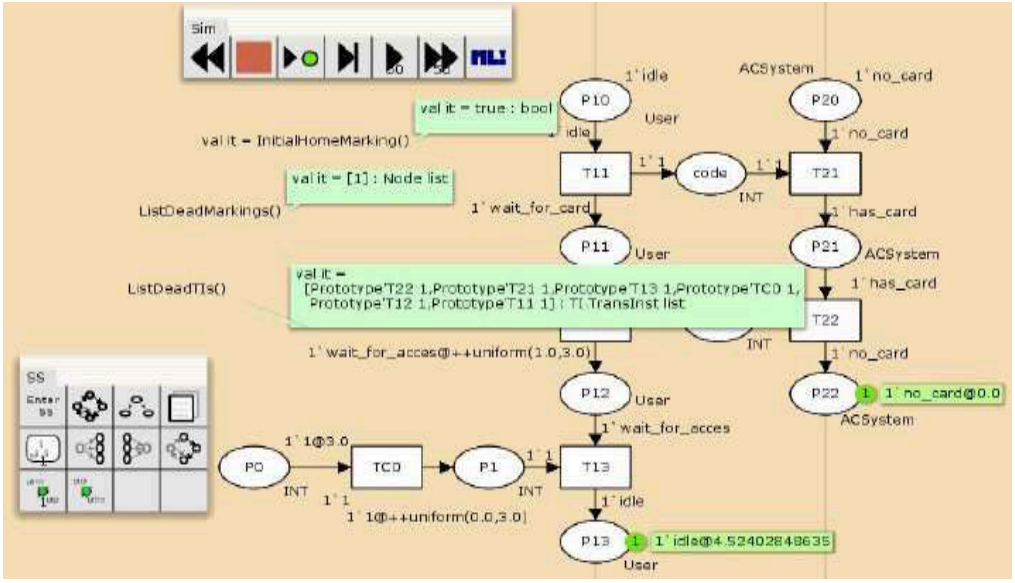



FIGURE 4.12 – Test de vivacité du modèle

la transformation des contraintes OCL [3] en termes de la logique temporelle CTL utilisant CPN Tools et le model checker PROD. Dans notre travail, nous cherchons à décrire une propriété sous forme d'une expression $OCL^{TR}$ afin de définir nos besoins. Dans ce cas, nous choisissons de reformuler les contraintes en terme d'une logique temporelle temporisée sur un système fini [Esh02]. CPN Tools n'offre pas la possibilité de cette classe de logique,

3. https ://www.omgmarte.org/

nous proposons alors une deuxième transformation des TD en termes de TPN que nous cherchons à définir sous la logique temporelle TPN-TCTL utilisant Roméo.

## 4.4 Formalisation des TD en TPN

Dans cette section, nous proposons la transformation des TD en termes d'éléments TPN ainsi que l'analyse et la vérification du modèle généré utilisant Roméo, par la suite, nous appliquons cette dernière à base d'un exemple d'illustration (l'exemple est détaillé dans le chapitre 5).

Nous définissons formellement le modèle noté $M_{TPN}$ pour la transformation d'un TD, ainsi qu'à la proposition d'un ensemble de règles de transformation de cette approche (tableau 4.1). Pour notre transformation, nous proposons une fonction F définie comme suit (Equation 4.2).

$$F : Lf \cup Pt \cup Msg \cup State \cup Dc \cup Tc \rightarrow T \cup P \cup A \cup W \cup m_0 \cup I \tag{4.2}$$

Ensuite et afin de décrire nos contraintes temporelles, nous considérons les annotations du profil UML MARTE (MARTE foundations), telles que $\ll RessourceUsage \gg$ et $\ll CommunicationMedia \gg$. Pour le cas de notre TD, nous faisons recours seulement à la première annotation qui occupe le temps d'exécution d'une tâche dans un processus d'un TD (tableau 4.2).

TABLE 4.2 – Annotations temporelles utilisées du profil UML MARTE

| **Profil UML MARTE** | **Spécification temporelle** |
|---|---|
| RessourceUsage | Temps d'exécution d'une tâche |

Au niveau de la transformation des éléments d'un TD, les messages et les lignes de vie sont translatés en termes de places TPN correspondants aux états initiaux des objets d'un TD. Les contraintes temporelles et de durée sont converties en termes de transitions munies d'intervalles temporels. Afin de différencier la sémantique de ses deux contraintes au moment de leur transformation, nous supposons deux transformations distinctes (figure 4.14). L'ensemble des règles de transformation est résumé dans le tableau 4.3.

Nous donnons quelques exemples de règles proposées, il s'agit respectivement des transformations d'une ligne de vie, d'un état, d'une contrainte temporelle et d'une contrainte de durée.

1. soit $lf_i \in$ Lf ; createplace$(p_i) \in$ P ; F : Lf$\rightarrow$P ; $F(lf_i) = p_i$ ;

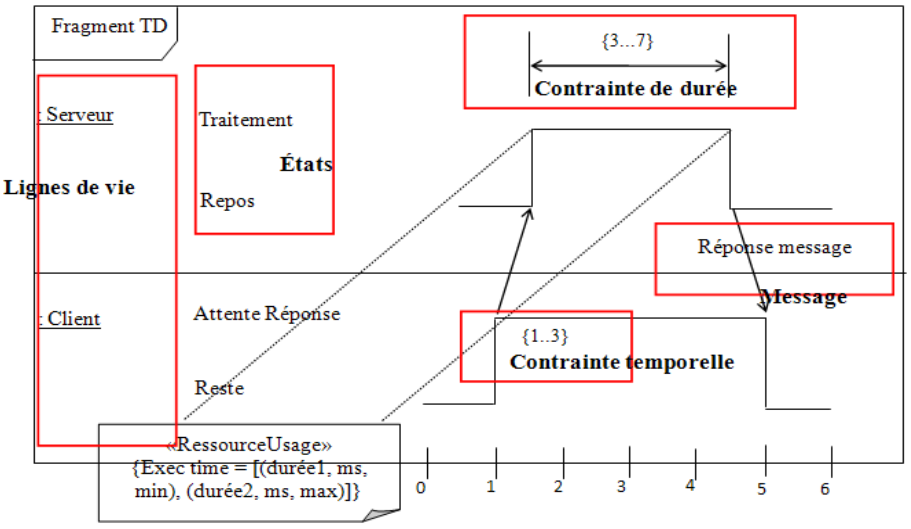



FIGURE 4.13 – Combinaison d'un TD avec les annotations MARTE

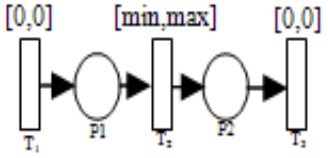


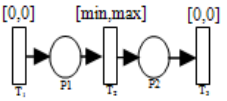

FIGURE 4.14 – Transformation d'une contrainte de durée

2. soit $st_i \in$ State, $p_i \in P$, $a_i, a_j \in A$ ; createsequence$(a_i, p_i, a_j)$ ; $F$ : State $\rightarrow P$ ; $F(st_i) = p_i$ ;
3. soit $tc_i \in$ Tc, $p_i \in P$, $a_i, a_j \in A$ ; createsequence$(a_i, t_i, a_j) + I[min, max]$ ;$F : Tc \rightarrow t + I[min, max]$ ; $F(tc_i) = t_i^{I[min,max]}$ ;
4. soit $dc_i \in$ Dc, $p_i, p_j \in P$, $t_i, t_j, t_k \in T, a_i, a_j, a_k, a_l \in A$ ; result = createsequence $(t_i^{I[0,0]}, a_i, p_i,$ $a_j, t_j^{I[min,max]}, a_k, p_j, a_l, t_k^{I[0,0]})$ ; $F$ : Dc $\rightarrow P$ ; $F(dc_i) = result$ ;

## 4.5 Conclusion

Tout le long de ce chapitre, nous avons proposé deux approches de formalisation pour la vérification des TD à base de classes temporelles de RdP. Dans une première partie, nous avons proposé une formalisation des TD en TCPN ainsi que son analyse et sa vérification utilisant CPN Tools. Un outil de transformation est développé, par la suite, nous l'avons appliqué sur un système de contrôle d'accès dans une salle serveur. Dans une deuxième partie, nous avons proposé une formalisation des TD en TPN. La simulation et la véri-

TABLE 4.3 – Transformation des éléments TD en éléments TPN

| Règles | Eléments TD | Construction TPN | Illustration Graphique |
|---|---|---|---|
| 1 | Ligne de vie | Place de marquage initial $M_0$ |  |
| 2 | Point d'interaction | Transition TPN + intervalle de temps [0,0] |   |
| 3 | Message | Place de communication |  |
| 4 | État | Place |  |
| 5 | Contrainte temporelle | Transition TPN + intervalle de temps [min, max] |   |
| 6 | Contrainte de durée | Transition TPN + intervalle de temps [min, max], 2 places et 2 transitions |   |

fication du modèle généré sont données dans le prochain chapitre où nous traiterons la logique TPN-TCTL. Pour la formulation des contraintes et du fait le manque préalable du concepteur UML au niveau de l'analyse formelle, nous proposerons dans le prochain chapitre, une nouvelle approche de formalisation des contraintes temporelles OCL $OCL^{TR}$ en termes des logiques temporelles TCTL et TPN-TCTL.

# Chapitre 5

# Formalisation des contraintes $OCL^{TR}$ en logiques temporelles temporisées

**Sommaire**



## 5.1 Introduction

Dans ce chapitre, nous présentons une contribution qui consiste en une transformation des contraintes $OCL^{TR}$ applicable sur les diagrammes UML (structure hiérarchique d'un IOD utilisant des TD) en termes des logiques temporelles TCTL et TPN-TCTL. Tout d'abord, nous décrivons la motivation de la formalisation OCL et de la logique TPN-TCTL utilisées ainsi que les contraintes $OCL^{TR}$. Par la suite, nous faisons part aux principaux travaux de formalisation OCL. Enfin, nous proposons notre contribution au niveau de la transformation des contraintes $OCL^{TR}$.


FIGURE 5.1 – Choix du langage OCL

## 5.2 Expression des contraintes sous OCL

### 5.2.1 Le langage OCL

Le langage OCL [omga] (Object constraint language), issu d'une grammaire précise, tend à décrire et exprimer des contraintes formelles sur la plupart des diagrammes UML définis. Prenons un exemple pour mieux éclaircir ces concepts. Soit un diagramme de classe ainsi que son diagramme d'objets associé illustrés comme suit (figure 5.2) :

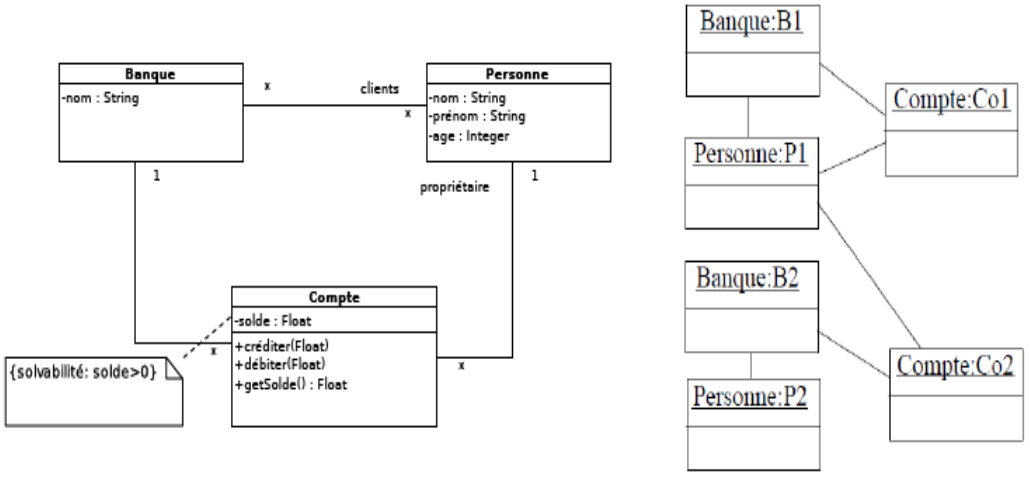



FIGURE 5.2 – Utilité d'une contrainte OCL

Pour lire ces diagrammes, nous pouvons commenter qu'une personne peut admettre un compte dans une banque où elle n'est pas cliente. Aussi, une personne est cliente d'une banque mais sans y avoir de compte. Ces deux affirmations sont fausses bien qu'elles se sont bien représentées dans le diagramme de classe. Dans ce cas, le langage OCL permettra de spécifier formellement cette ambiguïté qui n'a pas pu être modélisée, afin de donner plus d'expressivité au diagramme.

Généralement, une contrainte OCL agit sur une classe (un exemple est donné dans la

figure 5.3), cette dernière est dénotée **self** ou **invariant de classe**, elle signifie qu'une contrainte émise sur un diagramme doit être toujours vraie pour toutes les instances appartenant à cette classe (appelée aussi contexte). Une expression globale d'une contrainte OCL est définie comme suit :

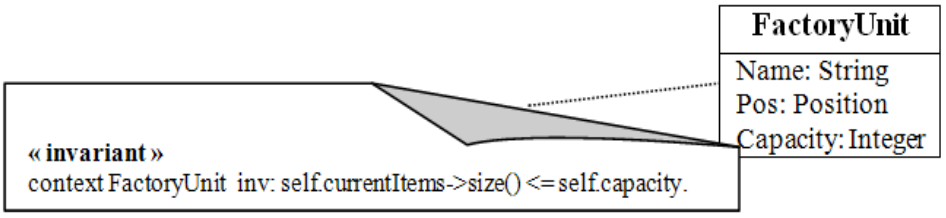



FIGURE 5.3 – Exemple d'une contrainte OCL

**Definition 5.1** Objet de contexte : *classe***invariant** : *expressionOCL*, sanchant que :

- Contexte : définit le classificateur sur laquelle la contrainte est évaluée.
- invariant : suite de la contrainte qu'elle doit être toujours vraie pour tous les états du système.
- expression OCL : spécifie la contrainte mise pour une éventuelle vérification.

Par conséquent, chaque contrainte établie doit admettre un type que ce soit défini par l'utilisateur (classe, interface, etc.) ou des types basiques sont prédéfinis (entier, réel, booléen, collection, etc.). Le type collection partage un ensemble d'éléments de même type. Entre autre, une bibliothèque standard est disponible fournissant des opérations prédéfinies telles que (size(), allInstances(), notEmpty(), asSet(), etc.) et manipulant des valeurs et des objets.

Mais actuellement, OCL ne peut pas fournir suffisamment de moyens pour bien spécifier les contraintes sur le comportement dynamique d'un modèle, ni pour la description d'information issues des STR. Nous marquons la naissance de l'extension temporelle d'OCL $OCL^{TR}$ permettant de spécifier les contraintes temporelles sur les diagrammes UML.

### 5.2.2 Vers $OCL^{TR}$

$OCL^{TR}$ marque une extension temporelle du langage OCL comprenant de nombreux nouveaux types et opérations qui sont mis en place. Dans [FM04], les auteurs ont abordé la définition de quelques concepts liant les diagrammes de machine d'état entre plusieurs objets. Pour cela, ils ont utilisé la fonction booléenne **OclInState** de type **OclAny** faisant partie du méta-modèle OCL d'une façon informelle et utile pour la définition et l'interprétation des expressions OCL, prenant en charge les états passés (pastOclExpression) @*pre* et les états futures (FutureOclExpression) @*post* dans les expressions temporelles.

Plusieurs autres travaux ont combiné l'utilisation des contraintes $OCL^{TR}$ sur les diagrammes UML. Dans [GCS08], les auteurs l'ont utilisé pour spécifier l'aspect TR sur les contraintes à base des diagrammes de séquence. Dans [FM02b], les auteurs ont proposé la traduction des contraintes $OCL^{TR}$ en termes de la logique temporelle CCTL (Clocked CTL)[RK97]. Dans [RKT+97], les auteurs ont exprimé les contraintes TR en termes de la logique TCTL sans pouvoir autant référencer le langage OCL. Dans [BM07], les auteurs ont défini formellement les diagrammes de machine d'état en termes d'automates temporisés dans le contexte d'une vérification décrite sous la logique TCTL. Compte tenu de la description des propriétés sous OCL, dans [BT06], les auteurs ont développé un outil automatique pour transformer les contraintes OCL en termes de la logique LTL.

Dans [FM02b], les auteurs ont formellement défini les opérations temporelles sous les types OclPath, OclConfiguration, OclState et OclAny. D'autres opérations telles que (excludes(), exists(), forall(), etc.) ont été réutilisées pour OclConfiguration dans la classe OclBasicKind, faisant partie du méta-modèle OCL. Un ensemble d'opérations sous le type OclState a été défini et appliqué pour les diagrammes de machine d'état afin de retourner l'ensemble d'état dans une séquence d'éxécution. Entre autre, les TD et les machines d'état partagent une certaine sémantique au niveau du séquencement d'état, dans ce sens, nous proposons l'extension de nouvelles capacités prédéfinies sur les TD sous le type OclAny afin de parcourir les états entre les objets. Dans ce sens, plusieurs opérations telles que $(config(), pre(), post()etnext(), etc.)$ peuvent être exploitées.

## 5.3 Model Checking TPN-TCTL

Le model checking TPN-TCTL [BGR09] est défini pour les TPN. Il est implémenté sous Roméo et propose un ensemble d'algorithmes de vérification utilisant ces modèles et la logique temporelle temporisée. Soit un RdP temporel T, nous notons $\|T\|$ l'ensemble des séquences de tir de transitions dans l'espace d'état :

Dans [BGR09], les auteurs ont formellement défini les TPN comme suit :

**Definition 5.1** $T = (P, T, \bullet(.), (.)\bullet, M_0, \alpha, \beta)$ tel que $P = \{p_1, ..., p_m\}$ et $T = \{t_1, ..., t_n\}$.

Aussi, les mêmes auteurs ont proposé une syntaxe d'une logique TPN-TCTL définie inductivement comme suit :

**Definition 5.2** $TPN - TCTL ::= M(p_k) \bowtie V \mid t_i - t_j \leq d \mid faux \mid \neg\varphi \mid \varphi \rightarrow \psi \mid \varphi E U_{\bowtie c} \psi \mid \varphi A U_{\bowtie c} \psi$.

Avec : M et faux sont des mots clés, $t_i$,$t_j \in T$, $M_i$,$M_j \in N_p$, $\varphi\psi \in TPN - TCTL$, $pi \in P$, $c, d, V \in N$ et $\bowtie \in \{<, \leq, =, >, \geq\}$.

Intuitivement, $M(p_i) \bowtie V$ indique que le marquage actuel de la place $p_i$ est en relation

$\bowtie$ avec V. $t_i - t_j \leq d$ indique qu'à partir de l'état courant, tout tir de $t_i$ est suivi d'un tir $t_j$ moins de d unités de temps plus tard. De même, $M_i - M_j \leq d$ indique qu'à partir de l'état courant, toute occurrence du marquage $M_i$ est suivie d'une occurrence du marquage $M_j$ moins de d unités de temps plus tard.

Dans ce sens, nous essayons d'utiliser la logique TPN-TCTL sous Roméo afin de vérifier certaines propriétés, nous nous sommes basés sur la syntaxe suivante :

$$GMEC = a*M(i)+,-b*M_j <,<=,>,>=,= k \mid deadlock \mid bounded(k) \mid pandq \mid porq \mid p \Rightarrow q \mid notp \tag{5.1}$$

Avec : M = marquage ; deadlock, bounded = mots clés ; i,j = indices place ; a,b,k = entiers ; $\{*, +, -, and, or, \Rightarrow, not\}$ : opérateurs habituels ; p,q : GMEC.

Une propriété TPN-TCTL peut être décrite sous la forme de l'une des propositions suivantes :

— $E(p)U[a,b](q)$,
— $A(p)U[a,b](q)$,
— $EF[a,b](p)$,
— $AF[a,b](p)$,
— $EG[a,b](p)$,
— $AG[a,b](p)$,
— $(p) \rightarrow [0,b](q)$,

Les quantificateurs et les opérateurs temporels sont ceux mêmes utilisés sous la logique arborescente CTL.

## 5.4 Formalisation en TCTL et TPN-TCTL à base des contraintes $OCL^{TR}$

La syntaxe et la sémantique des quantificateurs issus de la logique TCTL ont été définies dans le chapitre 2. Nous avons décrit le model checking de la logique TPN-TCTL, en vue de proposer une formalisation des propriétés temporelles dérivées des contraintes $OCL^{TR}$. Dans [FM02b], les auteurs ont proposé une nouvelle sémantique du méta-modèle du langage OCL applicables aux diagrammes de machine d'état, pour cela, ils ont enrichi les fonctions $OclConfig$ et $OclPath$.

Dans ce cadre, ils se sont intéressés à proposer une nouvelle sémantique de l'opération *post*, muni de l'opérateur @ permettant de se reporter à l'ensemble d'état d'une séquence d'exécution future, utilisée par un intervalle temporel $[min, max]$, décrivant une contrainte temporelle simple et exprimant que la séquence d'exécution sera applicable seulement entre

les bornes min et max.

Dans [omga] et au niveau des méta-modèles des duagrammes de machine et les TD, nous pensons la faisabilité de définir des fonctions applicables aux TD afin de vérifier le chemin des états d'un ensemble d'objets. Par conséquent, une contrainte $OCL^{TR}$ que nous proposons dans ce travail et retournant l'ensemble d'état d'une séquence d'exécution dans un intervalle $[min, max]$, applicable sur les TD peut être décrite comme suit :

$$inv : object@post[min, max] : set(OclPath) \tag{5.2}$$

Les bornes temporelles min et max doivent être du type entier où max peut être égale à $\infty$ dans le cas échéant. Par le tableau 5.1, nous proposons un ensemble de règles de transformation où nous définissons formellement les contraintes $OCL^{TR}$ en termes de formules TCTL et TPN-TCTL. Un invariant OCL inv indique qu'une formule TPN-TCTL doit commencer toujours par le prédicat AG (Always Globally). **'self'** dénote l'ensemble des instances prises dans la contrainte $OCL^{TR}$. Les états a et b dans une expression OCL dénotent respectivement les états source et destination propres à un même objet. Dans ce sens, notre objectif étant de tester la possibilité d'atteindre un état b à partir d'un état a dans un intervalle $[min, max]$.

Sous les logiques temporelles TCTL et TPN-TCTL, a et b représentent les états source et puits du modèle TPN généré. D'une façon intuitive, l'opérateur Exists sous OCL est transformé en un quantificateur E (Exists) sous TCTL, de même l'opérateur temporel ForAll est transformé en un quantificateur G (Globally). Les imbrications Exists(Exists) et ForAll(ForAll) sont retranscrites respectivement aux prédicats EF et AG.

Seules les règles 4 et 5 peuvent être générées sous la syntaxe TPN-TCTL, la compatibilité des trois premières règles sera proposée dans des travaux ultérieurs.

Les contraintes $OCL^{TR}$ peuvent être exprimées directement sur les diagrammes UML et générées directement sur les modèles TPN utilisant les logiques temporelles TCTL et TPN-TCTL. Par conséquent, l'analyste pourra par exemple vérifier l'accessibilité, la sûreté du modèle TPN (règle 4) comme donnée dans [Yov98], la vivacité ou détecter un certain blocage (règle 5).

TABLE 5.1 – Contraintes $OCL^{TR}$ et formules TCTL équivalentes

| **Numéro** | **Contrainte $OCL^{TR}$** | **Formule TCTL et TPN-TCTL** | **Logique et syntaxe** |
|---|---|---|---|
| 1 | inv :self@*post*[min,max]→ Exists(a,b :OclPath\|b→ ForAll(a)) | $AG(b \rightarrow EG_{[min,max]}(a))$ | TCTL |
| 2 | inv :self@*post*[min,max]→ Exists(a,b :OclPath\|b→ Exists(a)) | $AG(b \rightarrow EF_{[min,max]}(a))$ | TCTL |
| 3 | inv :self@*post*[min,max]→ ForAll(a,b :OclPath\|b→ ForAll(a)) | $AG(b \rightarrow AG_{[min,max]}(a))$ | TCTL |
| 4 | inv :self@*post*[min,max]→ ForAll(a,b :OclPath\|b→ Exists(a)) | $AG(b \rightarrow AF_{[min,max]}(a))$ | TCTL et TPN-TCTL |
| 5 | inv :self@*post*[min,max]→ Exists(a :OclPath\|a→ (Next(a) → isActive())) | $AG_{[min,max]}$(not(a) → not deadlock) | TPN-TCTL |

## 5.5 Exemple d'illustration 'Chargement d'une page web' et vérification du modèle

### 5.5.1 Description et modélisation UML

Maintenant, nous décrivons la procédure de chargement d'une page web via le navigateur, afin de pouvoir montrer les interactions possibles du processus de chargement. Un internaute déclenche le chargement de la page, il saisit son adresse URL via le navigateur et attends le résultat d'une éventuelle prestation. Ce dernier exécute immédiatement cette requête via le serveur DNS de test et tente de résoudre l'adresse émise. Si la réponse est positive, le système présente la requête d'exécution au Servlet et au Serveur Http, qui fournissent les données nécessaires pour l'accès. Ensuite, le Servlet envoie ces informations à la page du serveur (Java Server Page) qui à son tour les envoie directement au navigateur.

Nous donnons initialement un diagramme de classe qui représente l'architecture de notre système (figure 5.4). En outre et afin de décrire l'évolution des objets d'une manière temporelle, nous modélisons un diagramme TD (figure 5.5).

### 5.5.2 Vérification de l'exemple

L'analyse et la vérification du modèle TPN généré ont été assurées utilisant le model checker de Roméo et la logique temporelle TPN-TCTL pour exprimer les propriétés. Par

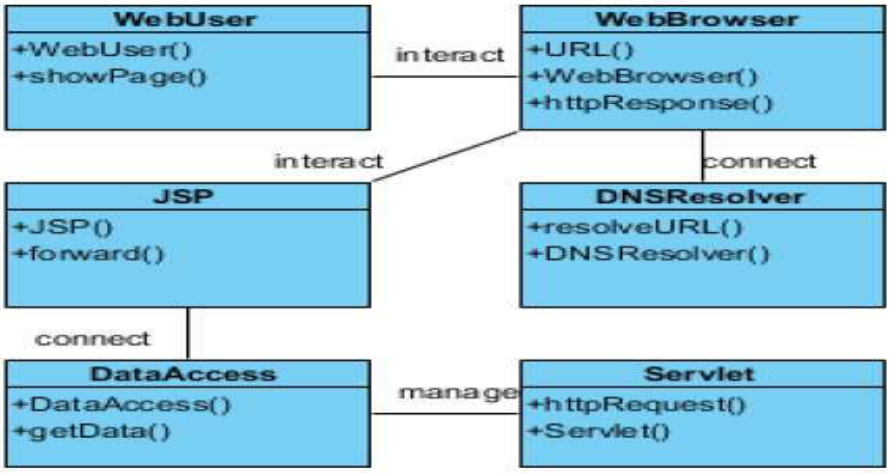



FIGURE 5.4 – Diagramme de classe du processus de chargement

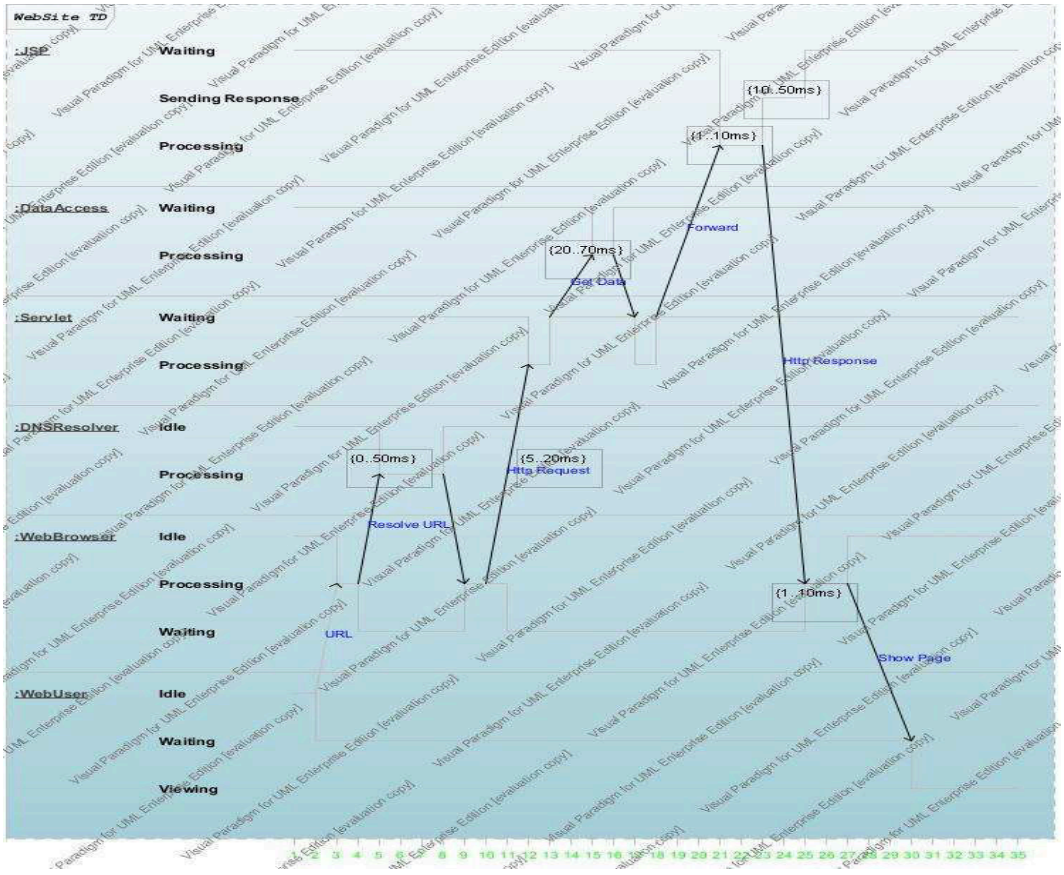



FIGURE 5.5 – Diagramme de timing du processus de chargement

les figures 5.6 et 5.7, nous illustrons respectivement le modèle TPN et les jeux de tests effectués. Notons que si la contrainte de durée d'un TD n'est pas affichée, un intervalle de

temps valant $[0, 1]$ est attribué par défaut sur la transition.

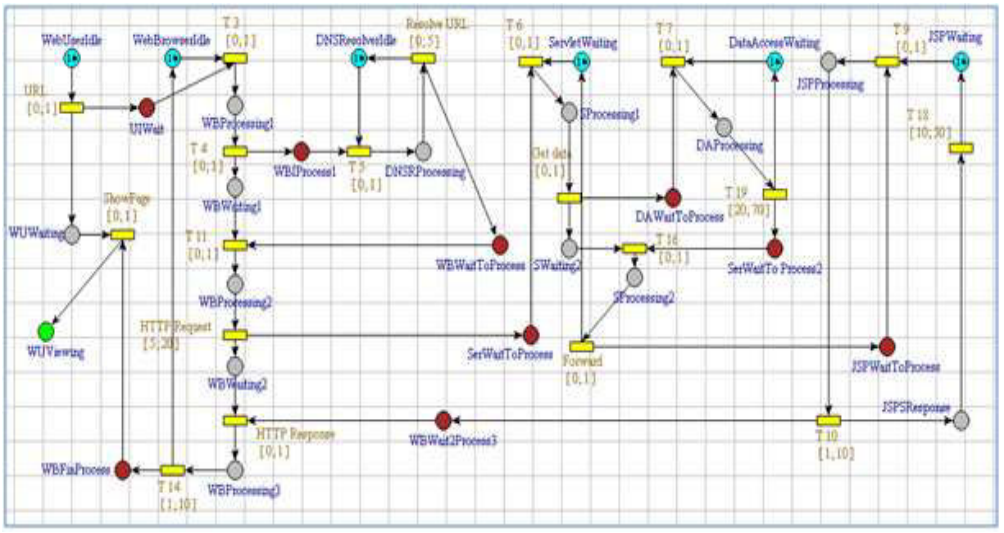



FIGURE 5.6 – Modèle TPN généré

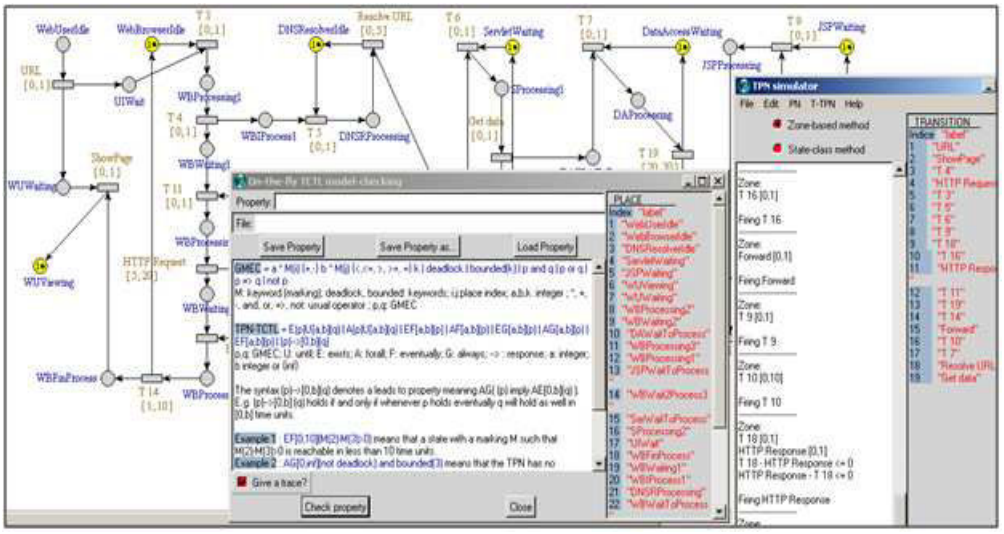



FIGURE 5.7 – Analyse du modèle TPN

Au niveau de la vérification du modèle [BGR09], nous devons garantir qu'il est toujours possible d'atteindre le marquage final en tenant compte de toutes les contraintes temporelles. Par conséquent, prouver que la composition du modèle a été bien définie sur un volet temporel. Comme suit, nous donnons les résultats de quelques propriétés retournés par Roméo et décrites informellement suivies par leurs traductions sous la logique TPN-TCTL.

1. *Propriété 1* : Nous voulons vérifier l'exécution du modèle. La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrite et

illustré comme suit (voir figure 5.8) :

— **Description informelle** : Nous préférons recevoir le résultat du processus de chargement dans les $27^{ms}(27 * 10^{-3} seconds)$, après une saisie d'une adresse URL par un internaute.

— **Description sous la logique TPN-TCTL** : EF[0,27](M(6)>=1) où 6 représente l'index du marquage final associé par Roméo et correspondant à la place WUViewing.

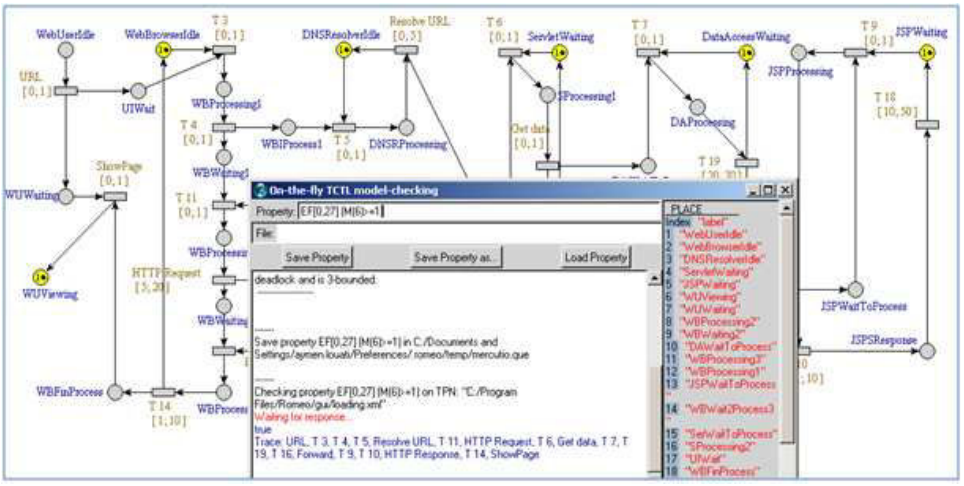



FIGURE 5.8 – Test de l'exécution du modèle

2. *Propriété 2* : Nous voulons vérifier la durée d'échange des messages entre les objets WebBrowser et JSP. La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrite et illustré comme suit (voir figure 5.9) :
   — **Description informelle** : Nous désirons vérifier l'échange entre les messages. L'envoi de la décision finale doit être au moins de 0 $25^{ms}$, la même pour les demandes d'envoi.
   — **Description sous la logique TPN-TCTL** : EF[0,25](M(27)-M(1)>0) où 27 et 1 représentent respectivement les indexes des places JSPProcessing et WebBrowserIdle associés par Roméo.
3. *Propriété 3* : Nous voulons vérifier la terminaison de l'exécution du processus respectant toute les contraintes de durée. La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrite et illustré comme suit (voir figure 5.10) :
   — **Description informelle** : Cette propriété commence avec le début de l'exécution du processus, jusqu'à un marquage de tous les états finaux. Dans notre modèle, nous voulons atteindre l'état final dans les $37^{ms}$, après le déclenchement du processus.

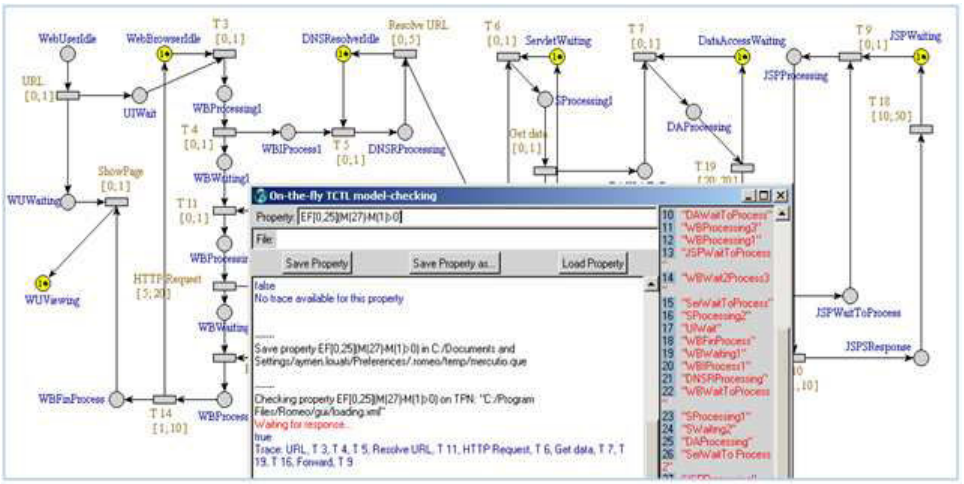



FIGURE 5.9 – Test de la durée d'échange des messages

— **Description sous la logique TPN-TCTL** : EF[0,37](M(6)>=1) and (M(2)>=1) and (M(3)>=1) and (M(4)>=1) and (M(5)>=1) and (M(30)>=1) où : 6, 2, 3, 4, 5 et 30 représentent respectivement les indexes des places WebUserViewing, WebBrowserIdle, DNSResloverIdle, ServletWaiting, JSPWaiting et DataAccess-Waiting associés par Roméo.

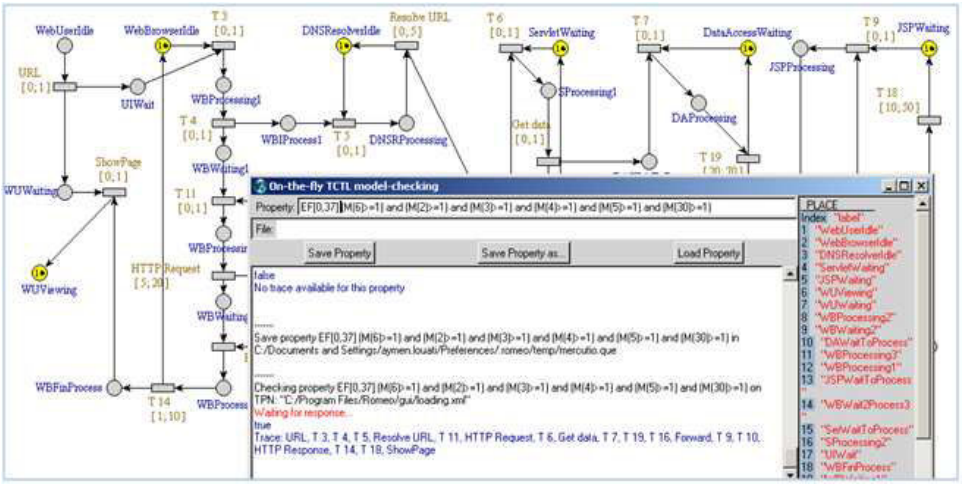



FIGURE 5.10 – Test de la terminaison du modèle

4. *Propriété 4* : Nous voulons vérifier la vivacité du modèle. La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrite et illustré comme suit (voir figure 5.11) :
   — **Description informelle** : Une transition est vivante, si elle est franchie au moins une fois à partir du marquage initial. Un modèle TPN est dit vivant, si toute

transition qui le compose est vivante. Par conséquent, il est nécessaire de vérifier au moins toute transition destinée à être utilisée dans le futur.

— **Description sous la logique TPN-TCTL** :EF[0,∞](M(1)=0) and (M(7)=1) and (M(12)=1) où : 1, 7 et 12 représentent respectivement les indexes associés par Roméo des places WebUserIdle, WebUserWaiting et WBProcessing1.

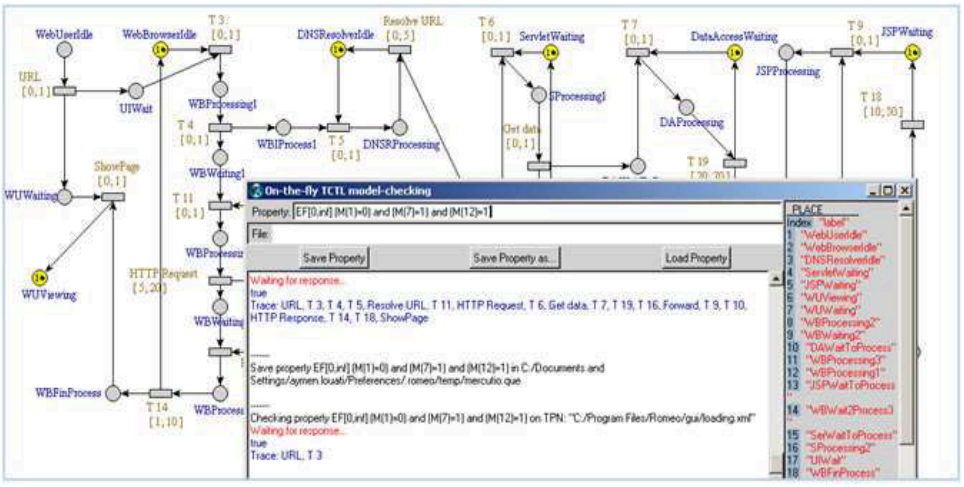



FIGURE 5.11 – Test de la vivacité du modèle

## 5.6 Conclusion

Tout le long de ce chapitre, nous avons cherché à définir formellement les contraintes $OCL^{TR}$ en termes des logiques temporelles TCTL et TPN-TCTL, afin de pouvoir faciliter et augmenter l'expressivité et la vérification des STR modélisés sous UML et ses profils TR en termes de TPN. L'exemple traité dans le chapitre précédent a été vérifié utilisant la logique TPN-TCTL. Dans le prochain chapitre, nous viserons l'application de l'ensemble des contributions proposées sur une étude de cas significative. Dans ce sens, nous chercherons à mettre en place une approche hiérarchique de formalisation pour la vérification des STR, utilisant un IOD et un ensemble d'interaction modélisant des TD.

# Chapitre 6

# Etude de cas et Applications

**Sommaire**



## 6.1 Introduction

Dans ce chapitre, nous présentons notre approche qui consiste à combiner des annotations temporelles du profil UML MARTE que nous appliquons sur l'ensemble de nos diagrammes UML dans une structure hiérarchique, représentée par un IOD et un ensemble de TD. Les propriétés désirées à vérifier sont originaires de l'expression des contraintes temporelles décrites sous forme de formules TPN-TCTL, compatible avec le model checker de Roméo et dérivées des contraintes $OCL^{TR}$ [FM02c].

Notre contribution est appliquée sur une étude de cas originale (Integrated Modular Avionics/Airborne Systems).

## 6.2 Description et modélisation UML du système

Dans cette partie, nous décrivons notre étude de cas ainsi que notre modélisation UML des TD dans la structure hiérarchique proposée utilisant les annotations du profil MARTE.

### 6.2.1 Description du système

Le système Integrated Modular Avionics **IMA-Based Airborne Systems** [GP12] est composé d'un émetteur jouant le rôle d'un capteur de collection de données et de deux récepteurs intégrés jouant le rôle de deux calculateurs A et B. Le routeur représente un lien virtuel de l'avionique full duplex (AFDX). Quant aux jeux de données, celles d'entrées pour le calcul sont émises par l'intermédiaire du routeur. Le comportement global du système est donné par un IOD (figure 6.2). Un protocole naïf est implémenté et mis en œuvre entre les deux récepteurs afin de bien distinguer les deux modes actif et passif et ce, en vue de transmettre une notification du message fourni par le AFDX. Dans ce contexte, l'émetteur lance les données vers les calculateurs A et B, utilisant le réseau de communication, contraignant d'un délai de retard de transmission entre 20 et $30^{ms}(10^{-3}seconds)$.

La conception du dispositif de commande redondante nécessite que la sortie des deux calculs des récepteurs sera disponible au même temps dans chaque cycle de travail (procédure d'échange de données à partir d'un émetteur jusqu'à sa réception des résultats de calcul par les calculateurs A et B).

A cet effet, nous vérifions le temps de calcul entre A et B. Comme il est impossible de bien respecter un strict calendrier pour la synchronisation des deux temps, une tolérance temporelle est définie. Nous considérons que les deux instants se coïncident où leurs calculs s'achèvent dans une même fenêtre temporelle égale à une tolérance proposée $\theta$ et définie par l'équation 6.1 comme suit :

$$\mid T(ActiveReceiver) - T(PassiveReceiver) \mid \leq \theta \tag{6.1}$$

Après l'obtention des données, le récepteur actif envoie automatiquement une notification asynchrone à son homologue passif et attend sa durée fixe pour lancer son calcul redondant. Le temps d'attente du récepteur actif et la modification de la gigue de réseau devraient être validés. Les contraintes temporelles du réseau au niveau du délai de retard de communication et les exécutions des tâches de chaque calculateur seront spécifiées utilisant le profil UML MARTE.

### 6.2.2 Modélisation UML

Dans cette partie, nous présentons les différents diagrammes dynamiques UML du système. Nous supposons la présence d'une structure statique illustrée par la figure 6.1. Le comportement global du système est décrit par un IOD et un ensemble de TD et est illustré comme suit (figure 6.2) :

La durée d'une tâche dans le processus d'exécution est modélisée par une annotation

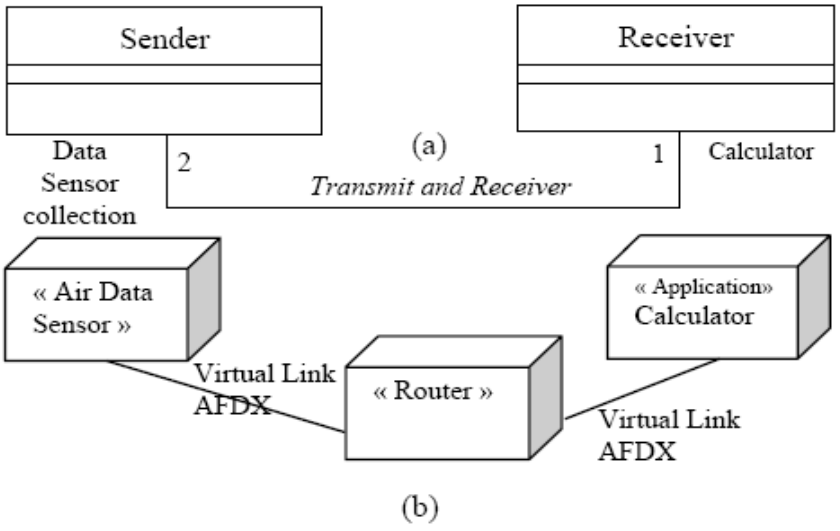



FIGURE 6.1 – Architecture du système : Diagramme de classe et de déploiement

temporelle $\ll$ *RessourceUsage* $\gg$. Le retard de communication enregistré entre deux processus est modélisée par une annotation temporelle $\ll$ *CommunicationMedia* $\gg$. Les deux dernières annotations du profil MARTE sont définies comme suit par le tableau 6.1.

TABLE 6.1 – Annotations utilisées du profil UML MARTE

| **Profil UML MARTE** | **Spécification temporelle** |
|---|---|
| RessourceUsage | Temps d'exécution d'une tâche |
| CommunicationMedia | Retard de communication |

### 6.2.3 Méthodologie adoptée

Dans la littérature, l'émergence du profil MARTE n'a pas encore été explorée avec les TD. Cependant, les TPN ont été bien traités dans de nombreuses publications comme un modèle de vérification. Peu de travaux ont traité la transformation des contraintes $OCL^{TR}$ en termes de plusieurs logiques temporelles [FM02a, FM02c, FM04].

L'idée de base que nous portons dans ce travail est de pouvoir exprimer les contraintes $OCL^{TR}$ sur les TD dans une structure hiérarchique. Par la suite, leurs transformations dans la logique temporelle TCTL et TPN-TCTL sur le modèle TPN généré.

L'architecture du système est décrite à base des diagrammes de classe et de déploiement. Nous modélisons un diagramme IOD où ses différents nœuds d'interaction référencent un ensemble de TD, à ce niveau, nous exploitons les annotations temporelles du profil MARTE. Enfin, nous exprimons toutes les propriétés désirées à vérifier en termes de contraintes

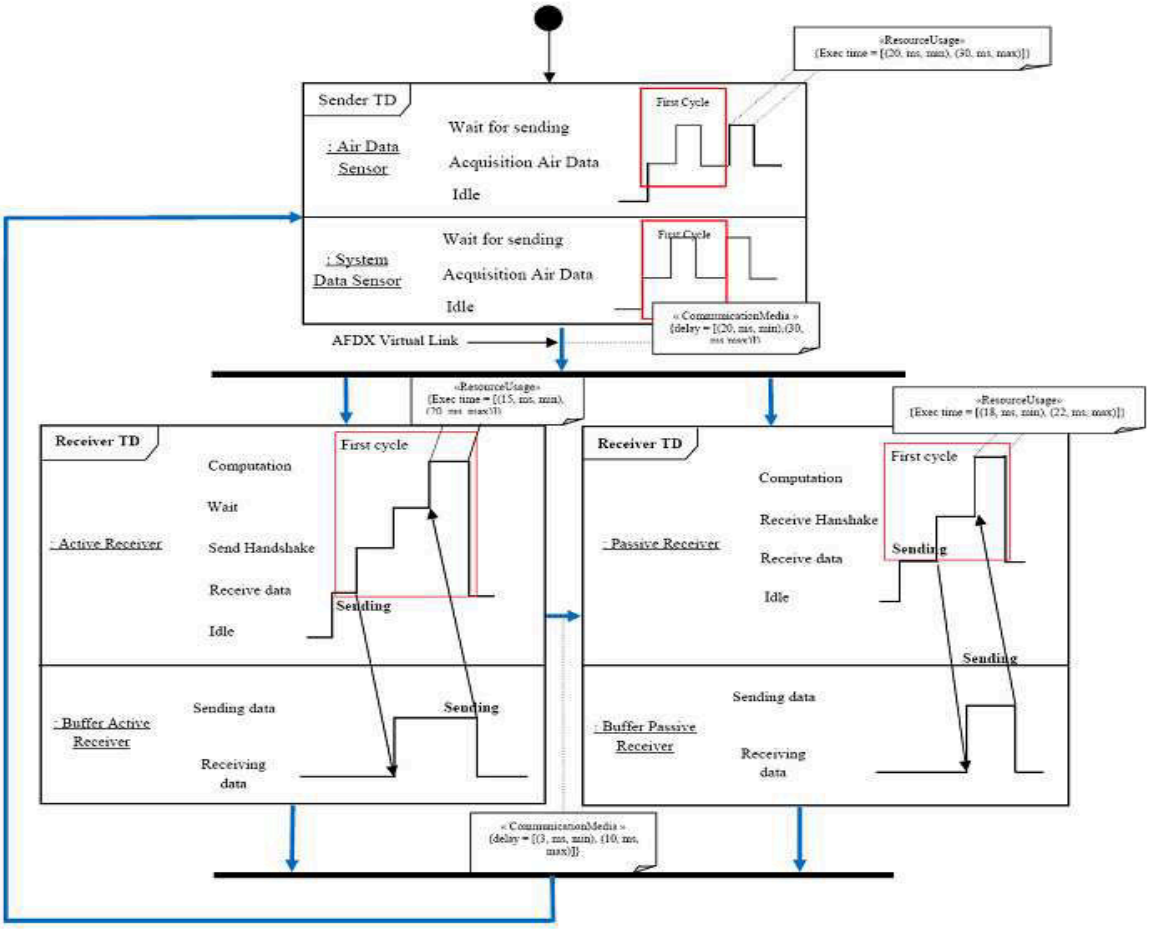



FIGURE 6.2 – Diagramme IOD et ses nœuds d'interaction TD

$OCL^{TR}$ où elles sont transformées en termes des logiques temporelles TCTL et TPN-TCTL sur les modèles TPN générés, dérivés eux même des diagrammes UML proposés. Cette contribution est illustrée comme suit (figure 6.3) :

Pour la description des contraintes $OCL^{TR}$, la fonction $OclPath()$ dans un TD sera exploitée afin de donner l'ensemble de tous les états d'une classe. Par un point initial d'un IOD, les nœuds d'interaction seront connectés afin de prendre en charge la structure hiérarchique du diagramme IOD.

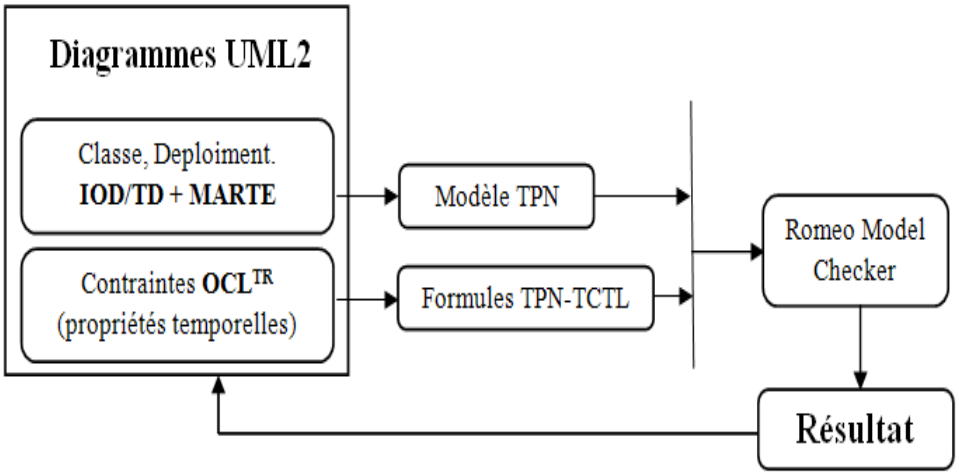



FIGURE 6.3 – Méthodologie adoptée

## 6.3 Transformation des éléments de modélisation UML en TPN

La transformation des éléments des TD a été traitée dans le chapitre 4 (tableau 4.3), dans ce sens, nous avons traduit les contraintes temporelles et de durée, les lignes de vie, les états et les messages en termes d'éléments TPN. Nous réutilisons cette transformation dans une approche hiérarchique composant un IOD et nous cherchons à transformer ses éléments de modélisation utilisés en éléments TPN. Comme suit, nous illustrons graphiquement les différentes règles de transformation des éléments utilisés.

### 6.3.1 Nœuds initial et final

Un nœud initial d'un IOD est appelé une place de départ $Pi$ dans un modèle TPN. Un nœud final du même modèle est appelé une place finale $Pf$. Pour assurer la liaison à une première composante TPN, la place $Pi$ doit être reliée à une transition de départ $T_i$ muni d'un intervalle de temps $[0, 0]$. La place $Pf$ est reliée de même à une transition finale $T_f$. Comme suit, nous démontrons par la figure 6.4 la transformation de ces deux nœuds. Notons que $Pi$ est représentée par $P_0$ dans ce même modèle.

### 6.3.2 Lien de communication entre deux nœuds d'interaction TD

Dans le but d'établir une communication entre les différents nœuds d'interaction d'un IOD (figure 6.5), nous transformons chaque nœud d'interaction en un sous-composant TPN. Pour les relier, nous décrivons le retard de communication par une contrainte de durée, transformée en trois transitions et deux places (figure 6.5). La transition $T_2$ est

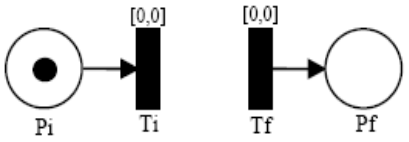



FIGURE 6.4 – Transformation des nœuds initial et final

munie d'un intervalle de temps $[min, max]$. Pour synchroniser deux nœuds d'interaction, nous employons des transitions $T_1$ et $T_3$ avec des intervalles de temps $[0, 0]$ et les places $P_1, P_2$.

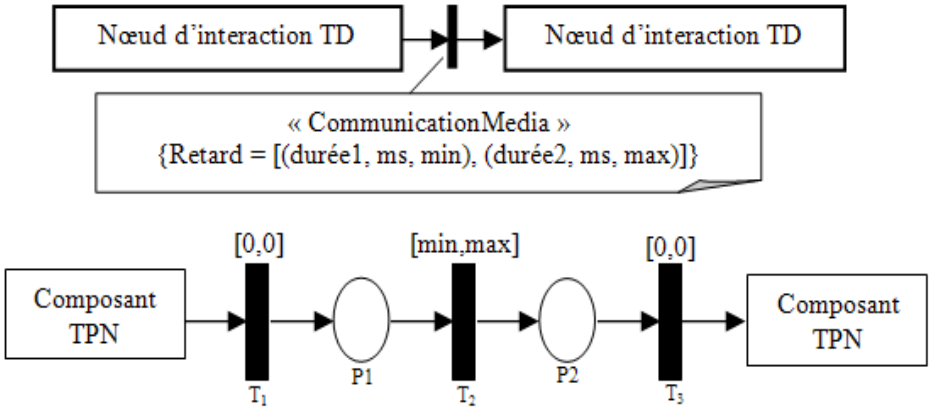



FIGURE 6.5 – Transformation d'une communication entre deux nœuds TD

Une fois la transition $T_1$ est exécutée, le jeton généré est lancé dans la place $P1$ jusqu'à l'opération du tir de la transition $T_2$. Par la suite, la transition $T_3$ est exécutée.

### 6.3.3 Transformation des nœuds de fourche et de jointure

Les nœuds de fourche et de jointure sont utilisés pour modéliser le principe de la concurrence dans un IOD. Ils définissent par conséquent une relation **ET** logique. Cela veut dire que leurs exécutions sont éventuellement concurrentes. Nous donnons comme suit par la figure 6.6 la transformation de ces deux éléments.

Dans cette description, nous nous appuyons sur une définition formelle équivalente des TPN donnée comme suit :

**Definition 6.1** $M_{TPN} = (P, T, A, W, m_0 \text{ et } I)$, où :

- $P = p_0, p_1, p_i, .., p_n$ représente un ensemble non vide de places,
- $T = t_0, t_1, t_i, .., t_n$ représente un ensemble non vide de transitions $\backslash (P \cap T = \emptyset)$,
- $A$ représente un ensemble fini d'arcs ou de liaisons $\backslash A \subseteq (P \times T) \cup (T \times P)$,

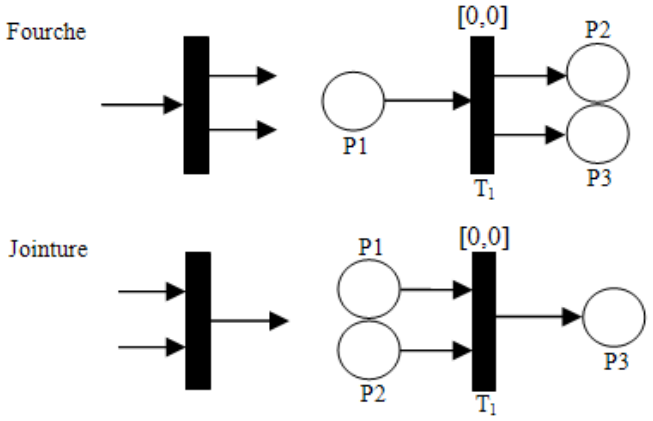



FIGURE 6.6 – Transformation des nœuds de fourche et de jointure

— $W : A \rightarrow N^*$,
— $m_0$ représente le marquage initial du modèle,
— $I : T \rightarrow [min, max]$ définit une contrainte temporelle où min représente le temps de franchissement minimum de la transition et max le temps au plus tard possible de son déclenchement.

Dans cette définition, un modèle TPN est représenté par un réseau de workflow temporel [BB12], où la place $p_0$ est appelée une place initiale $p_i$ ($p_0 = p_i$) et la place $p_n$ est appelée la place finale $p_f$ ($p_n = p_f$).

## 6.4 Vérification du modèle

Pour la vérification de nos modèles, nous avons eu recours à une vérification automatique par transformation de modèles qui consiste à l'utilisation de la logique TPN-TCTL [BGR09] utilisant le model checker de Roméo. Tout de même, il est possible de tester les propriétés de borgnitude, de la sûreté, de la vivacité ou du blocage sur le modèle TPN généré. Pour ce dernier, nous devons vérifier que le marquage final peut être toujours atteint tenant compte de toutes les contraintes temporelles. Nous illustrons respectivement par les figures 6.7 et 6.8, le formalisme TPN généré ainsi que la trace de simulation effectuée.

Dans une première étape, nous lançons simplement quelques propriétés spécifiques décrites sous la logique TPN-TCTL, nous donnons comme suit, les différents résultats de vérification retournés par le model checker de Roméo.

1. *Propriété 1* : Nous voulons vérifier l'absence du blocage pour tout le processus d'exécution dans les $42^{ms}$. La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrites comme suit (figure 6.9) :

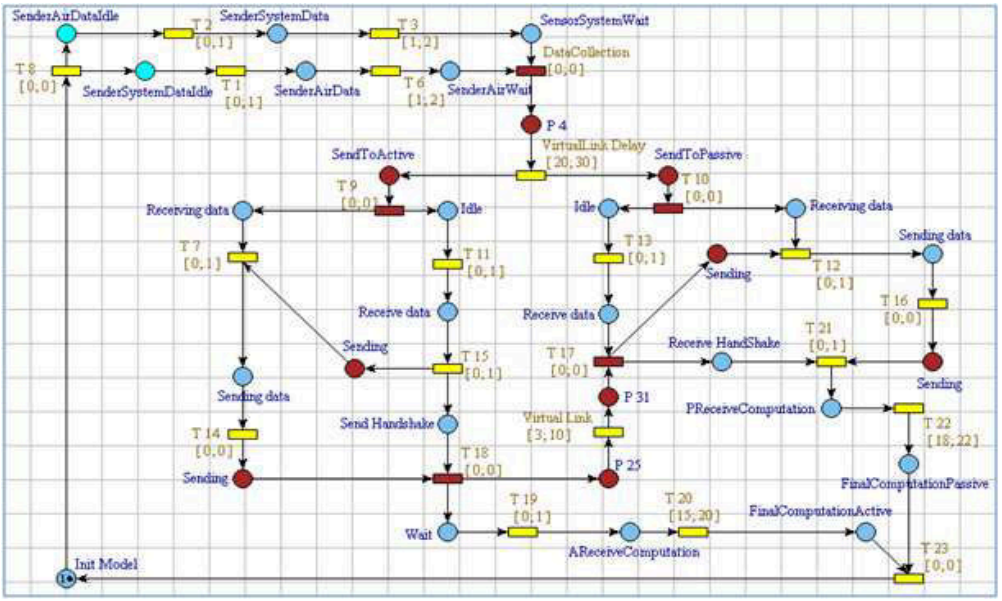



FIGURE 6.7 – Modèle TPN généré

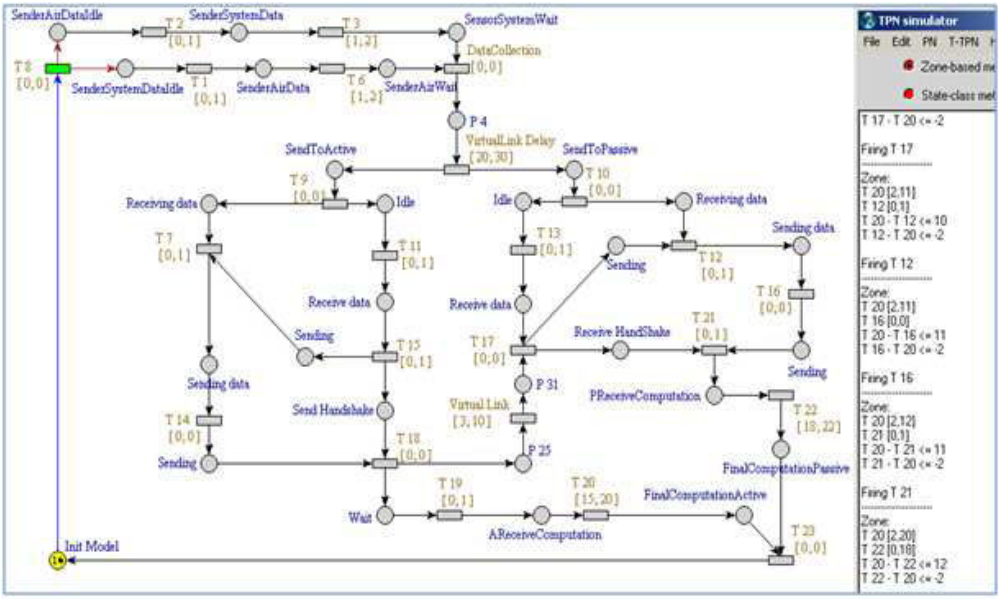



FIGURE 6.8 – Modèle TPN en simulation

Sous la logique TPN-TCTL, nous pouvons écrire : $AG[0, 42]$ (not deadlock)

2. *Propriété2* : Nous voulons chercher le temps de coïncidence de calcul entre les récepteurs actif et passif. Si le temps de calcul d'un récepteur actif se termine dans un instant t, le passif doit terminer son exécution dans (t+$\theta$). La formulation de cette propriété ainsi que le résultat retourné par Roméo sont respectivement décrites

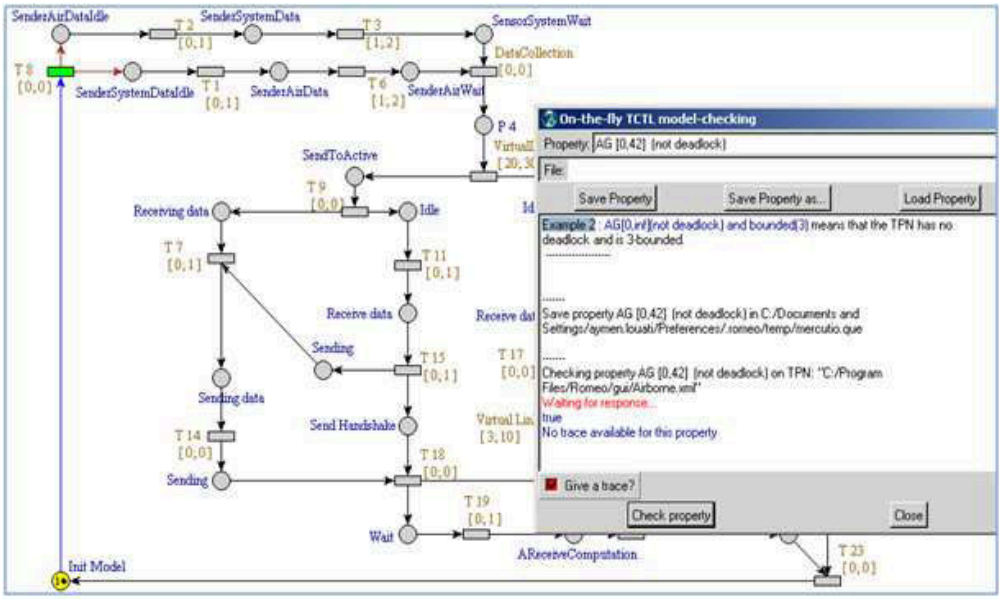



FIGURE 6.9 – Test de la propriété du blocage

comme suit (figure 6.10) :

Sous la logique TPN-TCTL, nous pouvons écrire : $EF[36, 36 + \theta]$ (M(22)>0) and (M(31)>0)

Notons de même que 36 représente le temps de calcul du récepteur actif. Le retard de communication est représenté par une durée $\in [20, 30]$ et le retard émis par le Handshake $\in [3, 10]$.

— $\theta = 1$ et Retard Handshake $\in [3, 10]$, **Faux**
— $\theta = 2$ et Retard Handshake $\in [3, 10]$, **Faux**
— $\theta = 3$ et Retard Handshake $\in [3, 10]$, **Faux**
— $\theta = 4$ et Retard Handshake $\in [3, 10]$, **Faux**
— $\theta = 5$ et Retard Handshake $\in [3, 10]$, **Faux**
— $\theta = 6$ et Retard Handshake $\in [3, 10]$, **Vrai**

Conclusion : Les deux temps de calculs se coiïncident à partir d'une valeur de $\theta = 6$

Dans une deuxième étape, nous nous offrons la possibilité de reformuler les contraintes $OCL^{TR}$ et les transformer en termes d'une propriété TPN-TCTL [SBB14]. Comme suit, nous donnons quelques exemples sur cette dernière formulation.

1. *Propriété 1* : Absence de blocage pour tout le processus d'exécution dans les 42ms, le résultat approprié est illustré par la figure 6.11 comme suit :

   Sous l'extension $OCL^{TR}$, nous exprimons cette contrainte ci-dessous :

   Context Sender inv :

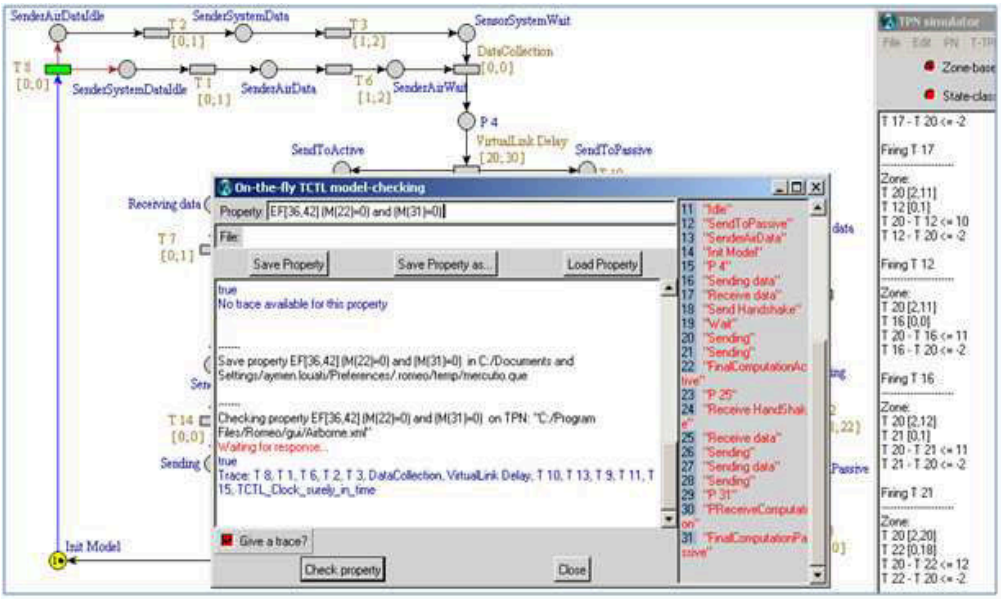



FIGURE 6.10 – Test de la tolérance temporelle

Self $@post[min, max] \rightarrow Exists(p : Self :: Idle|Next(p) \rightarrow isActive())$
Sous la logique TPN-TCTL, nous générons comme suit : $AG[0, inf](not(M(14) >= 1) \Rightarrow$ not deadlock)

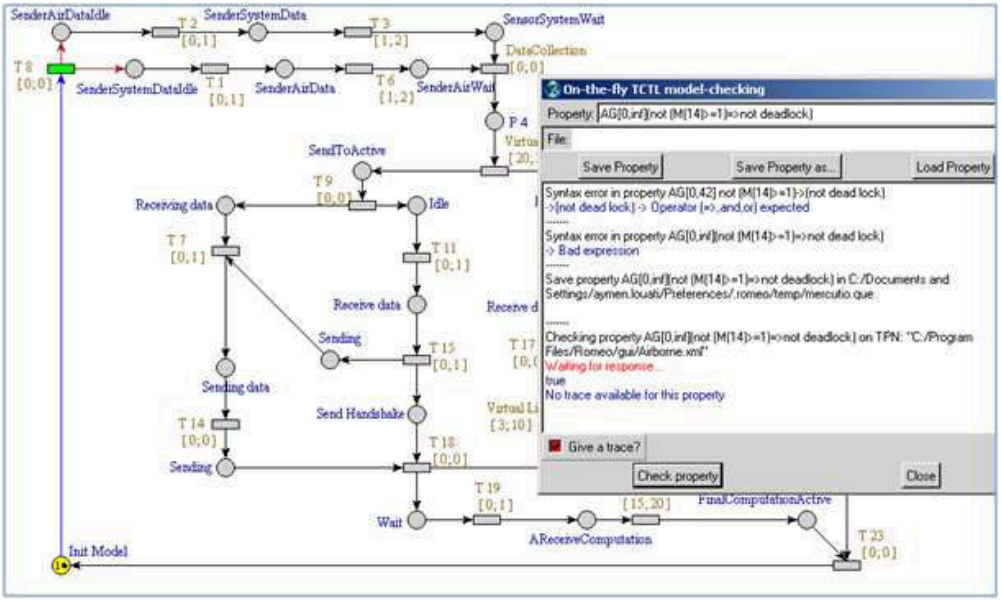



FIGURE 6.11 – Test de la propriété du blocage par transformation OCL

2. *Propriété 2* : Nous désirons vérifier le processus final d'exécution du modèle TPN généré et la terminaison proprement dite du calcul final dans les 42ms, après avoir

lancé le processus, le résultat approprié est illustré par la figure 6.12 comme suit :

Sous l'extension $OCL^{TR}$, nous exprimons cette propriété ci-dessous :

Context Sender inv :

inv :Self@$post[0,70]$ →ForAll(let $p1$ =ActiveReceiver : :FinalComputationActive, let $p2$ = PassiveReceiver : :FinalComputationPassive, let q = Self : :Idle|($p1$ and $p2$) → Exists(q)).

Sous la logique TPN-TCTL, la formule est générée comme suit :

$AG(M(22) = 0 and M(31) = 0) \Rightarrow AF[0,70](M(14) >= 1)$, Roméo nous offre syntaxiquement la possibilité de remplacer cette formule par : $(M(22) = 0 and M(31) = 0) \Rightarrow [0,42](M(14) >= 1)$

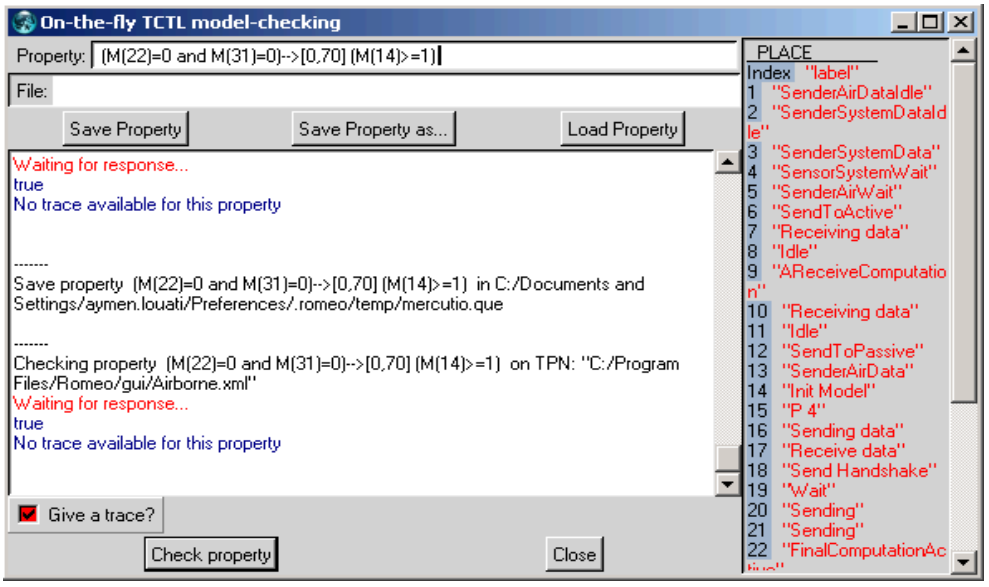



FIGURE 6.12 – Test de la sûreté du modèle

La place 'FinalComputationActive' illustre un état final du récepteur actif en relation avec la classe Receiver. La place $FinalComputationPassive$ représente l'état final du récepteur passif en relation avec la classe Receiver. De même, la place $Idle$ représente l'état initial de la classe Sender.

## 6.5 Conclusion

Tout le long de ce chapitre, nous avons proposé une nouvelle approche de formalisation pour la vérification des STR. Celle-ci a consisté en une transformation d'un ensemble de TD dans une structure hiérarchique en termes de modèles TPN. Les annotations temporelles du profil UML MARTE nous ont assurées la description des contraintes temporelles. Aussi,

nous avons exploité les contraintes $OCL^{TR}$ pour la formulation des propriétés désirées, que nous avons transformé par la suite en termes de formules décrites sous la logique TPN-TCTL et évaluées utilisant le model checker de Roméo. Pour la mise en valeur de nos expérimentations, nous avons appliqué notre approche sur une étude de cas, modélisant un Système Temps Réel asynchrone afin de garantir son efficacité logique et temporelle.

# Conclusion générale et perspectives

**Sommaire**



## Résumé des contributions et publications

La transformation des modèles définit une passerelle entre leur modélisation et leur vérification. Les utilisateurs qui tendent à modéliser formellement leurs besoins spécifiés, s'accrochent à une tâche qui n'est et ne sera jamais évidente [Tib09], celle de la vérification sur les modèles d'exécution générés. Cette étape qui sert à assurer la validité du modèle vis-à-vis d'une spécification générée, laisse encore une part à l'interprétation des techniques adéquates adoptées pour la spécification du comportement global d'un système.

Dans nos travaux et dans le but de réduire la complexité de la vérification des modèles, nous avons constaté l'intérêt de l'utilisation des RdP qui nous ont paru les modèles les plus appropriés pour notre modélisation, ils nous ont facilité la tâche de la spécification de nos diagrammes IOD et TD et de plus, leur robustesse, leur bonne application sur les STR nous ont servi à bien maîtriser toutes les démarches d'analyse et de vérification.

Le travail présenté dans ce mémoire de thèse, s'est intéressé essentiellement à la formalisation et à la vérification des diagrammes dynamiques UML2 à base des RdP. Pour cela, nous avons proposé un ensemble de contributions sur plusieurs niveaux.

Une première contribution a concerné la proposition d'une approche hiérarchique de formalisation pour la vérification des IOD utilisant les SD et les TD à base des HCPN. Pour ce faire, nous avons défini un ensemble de règles de transformation que nous les avons appliqué sur une partie du système DAB (Distributeur Automatique de Billets). CPN Tools nous a servi pour la génération, l'analyse et la vérification du modèle d'exécution. A ce niveau, nos travaux ont fait l'objet de trois publications ([LJB12], [LJB13a] et [LJB13b]).

Dans une deuxième étape, nous avons proposé la formalisation des TD en termes des

RdP temporisés et temporels. Pour le faire, nous avons défini et développé deux approches pour les vérifier que nous les avons appliqué par la suite, sur deux études de cas modélisées, analysées et vérifiées utilisant les outils CPN Tools et Roméo. Pour faire face aux exigences temporelles et contrôler l'exécution des différents objets, nous avons modélisé sur ces diagrammes à base du profil UML MARTE, l'ensemble des contraintes temporelles.

Le langage OCL a été utilisé dans le but de structurer la modélisation des systèmes sous UML et de décrire les propriétés temporelles désirées à vérifier. Pour ce faire, des règles de transformation ont été proposées en termes de la logique temporelle TCTL compatible avec le model checker de Roméo et la syntaxe TPN-TCTL.

Enfin et dans le but de mettre en valeur toutes les approches proposées, nous avons cherché à appliquer nos contributions sur une étude de cas (IMA- Based AirBorne Systems), donnée dans une structure hiérarchique sous forme d'un IOD et un ensemble de TD et utilisant le profil UML MARTE en termes de TPN. Le langage OCL et la logique TCTL ont été exploités au niveau de la vérification des modèles utilisés. A ce niveau, nos travaux ont fait l'objet de trois autres publications ([LBJ14b], [LBJ14a] et [LBS15]).

## Perspectives de recherche

Nos travaux ont touché plusieurs voies afin de mieux valoriser et prendre en considération d'autres travaux :

Premièrement, un défi s'impose afin de pouvoir expérimenter nos contributions à grande échelle. Nous tenterons l'intégration des opérations proposées dans le chapitre 5 dans le méta-modèle d'OCL qui seront capable de prendre en charge l'application des contraintes temporelles sur les TD.

Ensuite, nous proposerons l'exploration des cycles dans les TPN. Nous avons testé jusqu'à maintenant le temps de calcul sur un seul cycle, nous aurons à boucler les tests afin de réduire la taille de la tolérance permise issue de nos travaux. Nous essayerons de rapprocher au maximum les durées allouées par les calculateurs actif et passif.

Au niveau de la vérification et la formulation des propriétés sous la logique TPN-TCTL, nous avons remarqué que le model checker de Roméo n'exploite pas réellement les capacités offertes par la logique TCTL au niveau de la manipulation des quantificateurs et les opérateurs logiques. A titre d'exemple, la formule $AG(p- > EF[min, max](q))$ est syntaxiquement correcte au niveau TCTL, mais ne l'est pas sous la syntaxe de TPN-TCTL. Nous avons de même pensé à remplacer la propriété de sûreté $AG(p- > AF[min, max](q))$ en $p- > [min, max](q)$, celle-ci est vraie sous la dite syntaxe. Dans ce sens, nous proposerons cette implémentation dans le model checker de Roméo.

Finalement, nous proposerons la migration des modèles TPN conçus en termes des Time Open Workflow Nets, afin de bien spécifier les interactions des différents objets composant le modèle.

# Annexe A

# Annexe

## A.1 Outils utilisés pour la modélisation, l'analyse et la vérification des RdP

### A.1.1 CPN Tools

L'aspect formel des RdP a encouragé les développeurs à mettre au point une multitude d'outils pour la modélisation et la vérification automatique des RdP de haut niveau. L'outil CPN Tools [1], successeur des outils Design/CPN [2] et CPNAMI [3] est largement adopté au niveau industriel. CPN Tools est muni de plusieurs caractéristiques et est démontré très efficace sur le plan technique, par le format d'échange XML qu'il propose, par son exportation des traces de simulation après l'analyse de l'espace d'état et sa compatibilité avec les applications externes développées. CPN Tools convient avec les approches de vérification hiérarchique exploitant le concept des transitions de substitution, un choix qui nous a motivés à l'utiliser. Au point de nos travaux, nous avons utilisé la version 3.2.2 (02-2012).

Les composants de CPN Tools sont :

— CPN editor pour l'édition et la vérification de la syntaxe des CPN.

— CPN simulator pour la simulation du comportement du modèle.

— CPN State Space Tool pour la vérification de l'espace d'état.

L'outil CPN Tools combine la capacité des CPNs et la capacité d'un langage fonctionnel, le CPN-ML [4]. Ce dernier implémente et intègre d'une façon spécifique, le Standard ML (SML) qui est conçu à base du lambda-calcul. Cette spécification rajoute des constructeurs pour la définition de l'ensemble des couleurs et la déclaration des variables. SML est une des limites à la spécification des définitions, difficile à maîtriser et nécessitant un apprentissage

1. http ://cpntools.org/
2. http ://www.daimi.au.dk/designCPN/
3. http ://www-src.lip6.fr/logiciels/mars/CPNAMI/
4. http ://cpntools.org/trash/$an \cdot extension \cdot of \cdot standard \cdot ml$

assez long, mais efficace et performant pour la concision des programmes afin d'éviter l'explosion combinatoire et la réduction de la taille du graphe d'états. CPN-ML est destiné aux CPN pour son pouvoir expressif et demeure un moyen pour définir l'ensemble des couleurs et des fonctions et l'écriture des gardes sur les transitions et les arcs à travers des expressions booléennes sous forme de segments de code.

### A.1.2 Roméo

L'outil Roméo [5] (Olivier, H. Roux) est un logiciel performant d'analyse des RdP temporels (TPN). Roméo est développé sous la direction de l'équipe des STR et se compose d'une interface graphique d'utilisateur écrite sous (Tcl/Tk) assurant la modélisation et la simulation des Transition-TPN ainsi que les modules de calculs (AGPM et Mercutio, développées sous $C++$). Roméo pourra de même importer des modèles issues de l'outil TINA [6]. Sur les modèles TPN, Roméo assure :

— Le calcul de l'espace d'état basé sur les zones ou les classes d'état.
— La vérification des propriétés à base des marquages accessibles.
— Les jeux de tests pour le comportement dynamique du modèle.

Sur une extension des TPN à la programmation (TPN avec StopWatches PN), Roméo assure :

— Le calcul de l'espace d'état.
— La vérification des propriétés à base des marquages accessibles.
— Les jeux de tests pour le comportement dynamique du modèle à base d'intervalles de temps.

Sur les extensions paramétriques des CPN et SWPN, Roméo assure :

— Le calcul de l'espace d'état paramétrique.
— La vérification des propriétés paramétriques à base des marquages accessibles.
— Les jeux de tests pour le comportement dynamique du modèle à base d'intervalles de temps.

## A.2 Exemples de règles de transformation (TD-TCPN) implémentées sous Java

public class Regle

{

//**Transformation d'une ligne de vie en termes d'une place**

5. https ://romeo.rts-software.org/
6. https ://projects.laas.fr/

```
public void regleLifeline(List<Element> lstlifelines)
{
Iterator<Element> i = lstlifelines.iterator() ;
Iterator<State> st = lstState.iterator() ;
int x =0 ; while(i.hasNext() and st.hasNext())
{
State curr = st.next() ;
Element current = (Element) i.next() ;
Element ModelChildren = current.getChild("ModelChildren") ;
Iterator<Element> temp =ModelChildren.getChildren("TimeInstance").iterator() ;
String color =curr.getState(temp.next().getAttributeValue("StateCondition")) ;
number++ ;
Element place = createPlace(x,0,"1`"+color,"P"+number+"0",curr.getName()) ;
Element vguideline = new Element("vguideline") ;
vguideline.setAttribute(new Attribute("id",id + String.valueOf(compteur))) ;
vguideline.setAttribute(new Attribute("x",String.valueOf(x))) ;
Element workspaceElements = doc.getRootElement() ;
Element cpnet = workspaceElements.getChild("cpnet") ;
Element page = cpnet.getChild("page") ;
page.addContent(place) ; lstplace.add(place) ;
page.addContent(vguideline) ;
x+=200 ;
compteur++ ;
}
//Transformation d'une contrainte de durée en termes d'une garde
public void regleDC(Element current, Element arc)
{
List<Element> line = modelchildren.getChildren("LifeLine") ;
Iterator<Element> i = line.iterator() ;
while(i.hasNext())
```

```
{Element current1 = i.next();
Element modelChildren = current1.getChild("ModelChildren");
List<Element> listedc = modelChildren.getChildren("DurationConstraint");
Iterator<Element> i1 = listedc.iterator();
while(i1.hasNext())
{Element current2 = i1.next();
if(current2.getAttributeValue("EndTime").equals(current.getAttributeValue("Id")))
{ modifyArrow(arc, parse(current2.getAttributeValue("Name")));}
}}}
```

**//Transformation d'une contrainte temporelle en termes de deux places et deux transitions**

```
private static int n1=0;
public void regleTC(Element current, Element transition)
{
Element workspaceElements = doc.getRootElement();
Element cpnet = workspaceElements.getChild("cpnet");
Element page = cpnet.getChild("page");
if(current.getAttributeValue("TimeConstraint") != null)
{ String delims = "[{}]";
String [] tokens = current.getAttributeValue("TimeConstraint").split(delims);
String [] tokens1 = tokens[1].split(" [
.]+ ");
Element p1 = createPlace(50, 50, "1`1@"+Float.parseFloat(tokens1[1]), "P"+n1, "INT");
Element p2 = createPlace(20, 20, "", "P"+n1, "INT");
Element t1 =createTransition(100,100,"TC"+n1);
Element a1 = createArrow(t1.getAttributeValue("id"), p1.getAttributeValue("id"), "PtoT",
"1`1", 30, 30);
Element a2 = createArrow(t1.getAttributeValue("id"), p2.getAttributeValue("id"),
"TtoP", "1`1"+parse1(current.getAttributeValue("TimeConstraint")), 30, 30);
Element a3 = createArrow(transition.getAttributeValue("id"), p2.getAttributeValue("id"),
"PtoT", "1`1", 30, 30);
```

```
n1++ ;
page.addContent(a1) ; page.addContent(a2) ; page.addContent(a3) ;
page.addContent(p2) ; page.addContent(p1) ; page.addContent(t1) ;
}}}
```

# Glossaire

— $BDD$ : Binary Decision Diagrams
— $CADP$ : Construction and Analysis of Distributed Processes
— $CPN$ : Colored Petri Nets
— $CSP$ : Communicating Sequential Process
— $CTL$ : Computation Tree Logic
— $DAMARTS$ : Dependability Analysis Model for Real-Time Systems
— $HCPN$ : Hierarchical Colored Petri Nets
— $IMA$ : Integrated Modular Avionics
— $IPN$ : Instanciable Petri Net
— $LTL$ : Linear Temporal Logic
— $LTS$ : Labeled Transition Systems
— $MARTE$ : Modeling and Analysis of Real-Time and Embedded Systems
— $MDA$ : Model Driven Architecture
— $MDE$ : Model Driven Engineering
— $MITL$ : Metric Interval Temporal Logic
— $MTL$ : Metric Temporal Logic
— $OCL$ : Object Constraint Language
— $OCL^{TR}$ : Object Constraint Language temps réel
— $OMG$ : Object Management Group
— $RdP$ : Réseaux de Petri
— $RTLOTOS$ : Real-Time-LOTOS
— $SD$ : Sequence Diagrams
— $SMV$ : Symbolic Model Verifier
— $SOA$ : Service-Oriented Architecture
— $STR$ : Systèmes temps réel
— $TCPN$ : Timed Colored Petri Nets
— $TCTL$ : Timed Computation Tree Logic
— $TD$ : Timing Diagrams
— $THCPN$ : Temporal HCPN

— $TLTL$ : Timed Linear Temporal Logic
— $TPN$ : Time Petri Nets
— $TURTLE$ : Timed UML and RT-LOTOS Environment
— $UML$ : Unified Modeling Language
— $UMLTR$ : UML temps réel
— $XML$ : eXtreme Markups Language

# Bibliographie


[AAM05] N. Addouche, C. Antoine, and J. Montmain. Combining extended uml models and formal methods to analyze real time systems. *Proceedings of The 24th International Conference of Computer Safety, Reliability and Security (SAFECOMP)*, 3688 :24–36, 2005. 9, 25

[AFH96] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43 :116–146, 1996. 22

[Amb04] Ambysoft. *Agile Modeling Driven Development avec UML2.* Cambridge University Press, 2004. 7

[AMCN09] E. Andrade, P. Maciel, G. Callou, and B. Nogueiras. A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. *IEEE Digital society*, pages 266–271, 2009. 24, 25, 26

[APS03] C. Attiogbé, P. Poizat, and G. Salaum. Intégration de données formelles dans les diagrammes d'états d'uml. *Proceedings of The International Congres sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 3–17, 2003. 23

[BACTHTM07] K. Benghazi Akhlaki, M. Capel Tunon, J. Holagado Teriza, and L. Mendoza. *A methodological approach to the formal specification of real-time systems by transformation of UML RT design models.* Increasing Adequacy and Reliability of EISE, Elsevier, Science of Computer programming, 2007. 25

[Bal07] G. Balbo. *Introduction to Generalized Stochastic Petri Nets.* The 7th International School on Formal Methods for the Design of Computer, Communication and Software Systems : Performance Evaluation, 2007. 13

[Bas00] R. Bastide. *Spécification Comportementale par Réseaux de Petri : Applications aux Systèmes distribués à objets et aux Systèmes Interactifs.* PhD thesis, Centre de Recherche en Informatique, Université Toulouse1, France, 2000. 13

[BB12] H. Boucheneb and K. Barkaoui. Parametric verification of time workflow nets. *Proceedings of International Conference on Software Engineering*, pages 375–380, 2012. 75

[BBB14] S. Boufenara, Belala F. Barkaoui, K., and H. Boucheneb. Transactional petri nets : A semantic framework for uml2 activities. *Journal of Critical Computing Based Systems*, 5 :5–23, 2014. 24, 25, 47

[BBHP08] N. Belloir, J. Bruel, N. Hoang, and C. Pham. *Utilisation de SysML pour la modélisation des réseaux de capteurs.* Université de Pau et des pays de Adour LIUPPA, Version 1, Canada, 2008. 8

[BCD05] D. Berardi, D. Calvanese, and D. DeGiacomo. Reasoning on uml class diagrams. *Journal of Artificial Intelligence*, 168 :70–118, 2005. 25

[BD09] B. Bruegge and A.H. Dutoit. *Introduction to UML.* Object Oriented Software Engineering : Using UML, Patterns, and Java, Pearson, 2009. 1, 6, 8

[BDM02] S. Bernardi, S. Donatelli, and J. Merseguer. From uml sequence diagrams and state charts to analyzable petri net model. *3rd International workshop on Software and performance (WOSP)*, pages 35–45, 2002. 24

[BGR09] H. Boucheneb, G. Gardey, and O. Roux. Tctl model-checking of time petri nets. *Journal of Logic and Computation*, 196 :1509–1540, 2009. 22, 47, 60, 65, 75

[BM06] H. Bel Mokadem. *Vérification des propriétés temporisées des automates programmables industriels.* PhD thesis, Laboratoire Spécification et Vérification, cachan, France, 2006. 12

[BM07] A. Bouamari and M. Mostefai. Vérification formelle des modèles uml temps réel. *The 4th International Conference on Computer Integrated Manufacturing*, 2007. 60

[BMMR10] L. Baresi, A. Morzenti, A. Motta, and M. Rossi. From interaction overview diagrams to temporal logic. *Proceedings of the International Workshops and Symposia at MODELS*, 6627 :90–104, 2010. 25

[Boz12] A. Bozek. *Timed Coloured Petri Nets for Modelling, Simulation and Scheduling of Production Systems. Production Scheduling.* Industrial Engineering and Management, 2012. 46

[BT06] T. Bouabana-Tebibel. Formal validation with ocl. *IEEE Proceedings on Systems Man and Cybernetics*, pages 2731–2736, 2006. 24, 52, 60

[BT09] T. Bouabana-Tebibel. Semantics of the interaction overview diagram. *IEEE Proceedings on Information Reuse and Integration*, pages 283–287, 2009. 25, 26, 30, 31, 36

[BTB07] T. Bouabana-Tebibel and M. Belmesk. An object oriented approach to formally analyze the uml2.0 activity partitions. *Journal of Information and Software Technology*, pages 999–1016, 2007. 25

[CE81] E. Clarke and E. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic.* Dexter Kozen, Logics of Programs Workshop, Springer, 1981. 20

[CKZ11] C. Choppy, K. Klai, and H. Zidani. Formal verification of uml state diagrams : a petri net based approach. *ACM SIGSOFT Software Engineering Notes*, 36 :1–8, 2011. 14

[CM06] J. Campos and J. Merseguer. On the integration of uml and petri nets in software development. *Proceedings of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*, 4024 :19–36, 2006. 24, 25

[DBLM02] V. Del Bianco, L. Lavazza, and M. Mauri. Model checking uml specifications of real time software. *IEEE Proceedings on Engineering of Complex Computer Systems (ICECCS)*, pages 203–212, 2002. 9, 24

[DL03] J. Delatour and F. Lamotte. Argopn a case tool merging uml and petri nets. *Proceedings of the 1st International Workshop on Validation and Verification of Software for Entreprise Information Systems*, 2003. 24

[EBC03] M. Encarnaci, M. Barrio, and C. Cuesta. Uml automatic verification tool (tabu). *Proceedings of the 4th International Workshop on Specification and Verification of Component Based Systems*, 2003. 24

[EBSC04] M. Encarnacion, M. Barrio-Solorzano, and Carlos. E. Cuesta. Uml automatic verification tool (tabu). *Proceedings of the Workshop on Specification and Verification of Component-Based Systems*, pages 106–109, 2004. 23

[EJW02] R. Eshuis, D.N. Jansen, and R. Wieringa. Requirement level semantics and model checking of object oriented state charts. *Springer Verlag London Limited Requirements Engineering*, 7 :243–263, 2002. 11, 12

[ES09] S. Emadi and F. Shams. A new executable model for software architecture based on petri net. *Journal of Science and Technology*, 2 :15–25, 2009. 24

[Esh02] R. Eshuis. *Validation des diagrammes d'activités pour la modélisation du Workflow*. PhD thesis, Université Twente, Hollande, 2002. 53

[FM02a] S. Flake and W. Mueller. *An OCL Extension for Real Time Constraints*. Lecture Notes in Computer Science, Object Modeling with the OCL, 2002. 71

[FM02b] S. Flake and W. Mueller. Specification of real time properties for uml models. *International Conference on System Sciences*, 2 :3977–3986, 2002. 60, 61

[FM02c] S. Flake and W. Mueller. A uml profile for real time constraints with the ocl. *The 5th International Conference on Model Engineering, Concepts, and Tools*, pages 179–195, 2002. 69, 71

[FM04] S. Flake and W. Mueller. Past- and future-oriented time-bounded temporal properties with ocl. *The 2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 154–163, 2004. 59, 71

[FTJR07] J. M. Fernandes, S. Tjell, J. B. Jorgensen, and O. Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a colored petri net. *Proceedings of the 6th International Workshop on Scenarios and State Machines*, page 2, 2007. 24

[GCS08] D. Garcia, M. Cengarle, and N. Szasz. Uml 2.0 interactions with ocl rt constraints. *IEEE Specification Verification and Design Languages*, pages 167–172, 2008. 60

[GMC04] J. Grao, J. Merseguer, and J. Campos. From uml activity diagrams to stochastic petri nets : application to soft-ware performance engineering. *Proceedings of The 4th International Workshop on Softaware and performance*, pages 25–36, 2004. 24

[GP12] N. Ge and M. Pantel. Time properties verification framework for uml-marte safety critical real time systems. *Proceedings of the 8th European*

*conference on Modeling Foundations and Applications*, pages 352–367, 2012. 9, 70

[Gra08] Ch. Grandpierre. *Stratégies de génération automatique de tests à partir de modèles comportementaux UML/OCL*. PhD thesis, Université de Franche Comté, U.F.R Sciences et Techniques, France, 2008. 7

[HKL+10] H. Hansen, J. Ketema, B. Luttik, M. Mousavi, and J. Van de Pol. Towards model checking executable uml specifications in mcrl2. *Proceedings of International Conference on Innovations in Systems and Software Engineering (ICISSE)*, 6 :83–90, 2010. 23, 25

[Jen97] K. Jensen. *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science, Springer Verlag, 1997. 32

[Jen07] K. Jensen. Special section on colored petri nets. *Journal on Software Tools for Technology Transfer*, 9 :209–212, 2007. 14, 15

[Joo10] T. Joochim. *Bringing Requirements Engineering to Formal Methods Timing diagrams for Event B and KAOS*. PhD thesis, School of Electronics and Computer Science, University of Southampton, 2010. 45

[KM10] F. Kordon and Y. Mieg. Experiences in model driven verification of behavior with uml. *Proceedings of The 15th Monterey Workshop Foundations of Computer Software,Future Trends and Techniques for Development*, 6028 :181–200, 2010. 24

[KZJK10] C. Kangle, Y. Zongyuan, X. Jinkui, and W. Kaiyu. Unifying modeling and simulation based on uml timing diagram and uppaal. *The 2nd IEEE International Conference on Computer Modeling and Simulation*, pages 615–620, 2010. 24, 25

[LBJ14a] A. Louati, K. Barkaoui, and C. Jerad. Temporal properties verification of real-time systems using uml/marte/ocl-rt. *Formalisms for Reuse and Systems Integration, Advances in Intelligence Systems and Computing*, pages 133–147, 2014. 82

[LBJ14b] A. Louati, K. Barkaoui, and C. Jerad. Time properties verification of uml/marte real-time systems. *IEEE Information Reuse and Integration, International Workshop on Formal Methods Integration*, pages 386–393, 2014. 82

[LBS15] A. Louati, K. Barkaoui, and Z. Sbaï. Formal verification of uml2 timing diagrams based on time petri nets. *International Conference on Information Systems and Technologies*, 2015. 82

[LJB12] A. Louati, C. Jerad, and K. Barkaoui. Formalization and verification of hierarchical use of interaction overview diagrams. *The 24th International Conference on Software and Systems Engineering and their Applications*, 2012. 81

[LJB13a] A. Louati, C. Jerad, and K. Barkaoui. Formalization and verification of hierarchical use of interaction overview diagrams using timing diagrams. *Journal of Soft Computing and Software Engineering*, 3 :205–211, 2013. 81

[LJB13b] A. Louati, C. Jerad, and K. Barkaoui. On cpn based verification of hierarchical formalization of uml2 interaction overview diagrams. *IEEE of Modeling Simulation and Applied Optimization*, pages 1–6, 2013. 26, 81

[Mar06] A. Marchand. *UML pour les systèmes temps réel et embarqué, Module ME1 : Ingénierie du logiciel.* Université Polytechnique de Nantes, 2006. 6, 8

[MDM03] Y. Mieg, C. Dutheillet, and I. Mounier. Automatic symetry detection in well formed nets. *Proceedings of The 24th International Conference on Applications and Theory of Petri Nets*, 2679 :82–101, 2003. 13

[MH08] Y. Mieg and L. Hillah. Uml behavioral consistency checking using instantiable petri nets. *Journal of Innovations in systems and software engineering*, 4 :293–300, 2008. 24, 25

[MLG01] G. Martin, L. Lavagno, and J. Guerin. Embedded uml, a merger of real time uml and codesign. *The 9th International Symposium on Hardware and Software Codesign(CODES)*, pages 23–28, 2001. 25

[Mos07] Y. V. Moshe. *Linear Time Model Checking : Automata Theory in Practice.* Comuting Departement, Rice University, Houston, 2007. 23

[Nas07] O. Nasr. *Spécification et vérification des ordonnanceurs Temps Réel en B.* PhD thesis, Ecole Doctorale Mathématique, Informatique et Télécommunications, Université de Toulouse III, 2007. 19, 20, 22

[omga] Object management group, inc : Omg uml, super structure (10-2012). `http://www.omg.org/spec/UML2.4.1/`. 3, 8, 9, 29, 30, 44, 58, 62

[omgb] Object management group, inc : Omg uml, superstructure (06-2011). http://www.omg.org/spec/MARTE/1.1. 9, 10

[omgc] Object management group, inc : Omg uml2.0, superstructure (07-2005). http://www.omg.org/spec/UML/2.0/. 5, 6, 24

[omgd] Object management group, inc : Omg uml2.1.2, superstructure (11-2007). http://www.omg.org/spec/UML/2.1/. 8, 9, 24, 34

[RK97] J. Ruf and T. Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. *Proceedings of the International Conference on Correct Hardware Design and Verification Methods*, pages 146–163, 1997. 60

[RKT+97] E. Roubtsova, J. Katwijk, W. Toetenel, C. Pronk, and R. Rooij. *Specification of Real-Time Systems in UML.* Elsevier Science, 1997. 60

[SA09] P. Sannes and L. Apvrille. Making formal verification amenable to real-time uml practitioners. *Proceedings of The 12th European Workshop on Dependable Computing*, 2009. 25

[SB06a] S. Sengupta and S. Bhattacharya. Formal specification of uml use case diagram a z notation based approach. *Proceedings of the International Conference on Computing and Informatics (ICOCI)*, pages 1–6, 2006. 23

[SB06b] C. Snook and M. Butler. Uml b : Formal modeling and design aided by uml. *Proceedings of the ACM Transaction on Software Engineering and Methodology*, 15 :92–122, 2006. 23

[SBB14] Z. Sbaï, K. Barkaoui, and H. Boucheneb. Compatibility analysis of time open workflow nets. *Proceedings of the International Workshop on Petri Nets and Software Engineering*, pages 249–268, 2014. 77

[SH05] H. Störrle and J. Haussmann. Towards a formal semantics of uml2 activities. *Proceedings Software Engineering*, pages 117–128, 2005. 5

[SKM01] T. Schafer, A. Knapp, and S. Merz. *Model Checking UML State Machines and Collaborations.* Electronic notes in Therotical Computer Science, 2001. 23

[SMCC13] H. Seok Min, S. M. Chung, and J. Y. Choi. Deriving system behavior from uml state machine diagram. *Journal of Universal Computer Science Applied to Missile Project*, 19 :53–77, 2013. 23

[Stö04a] H. Störrle. Semantics and verification of data flow in uml2 activities. *IEEE Proceedings of Press Visual Languages and Formal Methods*, pages 38–52, 2004. 2, 24

[Stö04b] H. Störrle. Semantics of control flow in uml2 activities. *IEEE Proceedings of Symposium on Visual Languages, Human Centric Computing*, pages 235–242, 2004. 2, 24

[Stö04c] H. Störrle. Semantics of exceptions in uml2 activities. *Journal of Software and Systems Modeling*, 2004. 24, 25

[Stö04d] H. Störrle. Semantics of expansion nodes in uml2.0 activities. *Proceedings 2nd Nordic Workshop on UML, Modeling, Methods and Tools*, 2004. 24

[Sta07] T. S. Staines. Supporting uml sequence diagrams using a processor net model. *IEEE of Engineering of Computer-Based Systems (ECBS)*, pages 279–286, 2007. 24

[Sys07] OMG SysML. *OMG Systems Modeling Language Final public version planned Specification.* OMG, 2007. 9

[Tib09] O. Tibermacine. Uml et model checking. Master's thesis, Département Informatique, Faculté des Sciences de l'ingénieur, 2009. 81

[Vau87] J. Vautherin. Parallel systems specifications with coloured petri nets and algebraic specifications. *Proceedings of The 7th European Workshop on Applications and Theory of Petri Nets*, pages 293–308, 1987. 13

[VV13] G. Vinai and K. Vinod. Verification of interaction overview diagrams using colored petri nets. *International Journal of Advanced Research in Computer and Communication Engineering*, 2 :2138–2149, 2013. 30

[Yov98] S. Yovine. *Model checking timed automata.* Springer Verlag, 1998. 22, 62

[YS06] S. Yao and S. Shatz. Consistency checking of uml dynamic models based on petri net techniques. *Proceedings 10th IEEE International Conference on Computing (CIC)*, pages 289–297, 2006. 24

[Zha94] Y. Zhang. *A Foundation for the Design and Analysis of Robotic Systems and Behaviors.* PhD thesis, Department of Computer Science, University of British Columbia, Canada, 1994. 22

[ZS04] H. Zhaoxia and S. Shatz. Mapping uml diagrams to a petri net notation for system simulation. *Proceedings of the 16th International Conference on*

*Software Engineering and Knowledge Engineering(SEKE)*, pages 213–219, 2004. 24

le cnam

# Aymen LOUATI

# Contribution à la Formalisation et à la Vérification des Diagrammes Dynamiques UML2 à base des Réseaux de Petri

**Résumé :**
Les systèmes informatiques envahissent de plus en plus notre quotidien, en allant de la plus simple application de lecture des fichiers audio, à la plus critique comme les voitures et les avions. Dans les systèmes critiques, la validation par vérification formelle s'impose. Cette thèse s'inscrit dans ce cadre et tend à doter le langage UML, langage de modélisation standard de facto, d'une sémantique formelle pour des finalités de vérification. En premier lieu, nous avons analysé et révisé le fondement théorique des principales approches de vérification formelle issues de la littérature et se focalisant sur le langage UML, ses profils et les concepts des réseaux de Petri (RdP). En deuxième lieu, nous avons proposé une nouvelle approche hiérarchique de formalisation pour la vérification des diagrammes globaux d'interaction (IOD). Ensuite, nous avons développé des formalismes temporels et temporisés des diagrammes de Timing UML2 (TD), appliqués par des exemples d'illustration. Sur ceux, nous avons conçu une nouvelle approche hiérarchique, s'intéressant aux Systèmes Temps Réel (STR), utilisant l'extension temporelle du langage des contraintes objets OCL/Temps Réel ($OCL^{TR}$), le profil UML MARTE et la logique temporelle temporisée (TCTL), exploitée d'une technique de vérification automatique après la transformation du modèle (Model Checking). Enfin, nous avons appliqué les formalismes proposés sur une étude de cas, afin de garantir leur efficacité logique et temporelle.

**Mots clés :**
UML, MARTE, RdP, STR, IOD, TD, $OCL^{TR}$, TCTL, Model Checking.

**Abstract :**
The computer systems are increasingly invading our daily lives from the simplest application as audio files reading to the most critical one as cars and airplanes. For the critical systems, the validation by the formal verification is required. This Thesis concerns this area of research and aims to ensure the betterment of UML language, which is the de facto standard, with formal semantics for verification finality. For the first part, we have analyzed and revised the theoretical foundations the existing formal verification approachs used UML, their profiles and the basic concepts of the Petri Nets (PN). For the second part, we have created a novel hierarchical approach of formalization in order to verify the Interaction Overview Diagrams (IOD). Then, we have developed temporal formalisms based on the UML2 Timing Diagrams (TD), applied by illustration examples. Based on these, we have proposed a novel hierarchical approach which is interested in Real Time Systems (RTS) and employed the temporal extension of the Object Constraints language (OCL/Real Time) ($OCL^{TR}$), the UML MARTE profile and the timed computation Tree logic (TCTL), given by the Model Checking technique after the model's transformation. Finally, we have applied all the proposed formalisms using a case study, in order to ensure its logical and temporal efficiency.

**Keywords :**
UML, MARTE, PN, RTS, IOD, TD, $OCL^{TR}$, TCTL, Model Checking.