



HAL
open science

Approches pour la modernisation et vérification des systèmes temporisés en utilisant les diagrammes états-transitions et les réseaux de Pétri colorés

Mohamed Benmoussa

► **To cite this version:**

Mohamed Benmoussa. Approches pour la modernisation et vérification des systèmes temporisés en utilisant les diagrammes états-transitions et les réseaux de Pétri colorés. Autre. Université Sorbonne Paris Cité, 2016. Français. NNT : 2016USPCD062 . tel-02130460

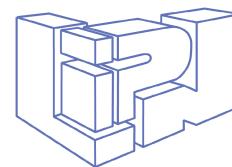
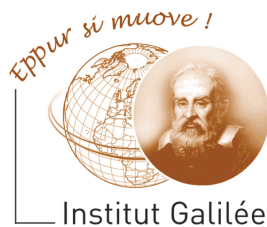
HAL Id: tel-02130460

<https://theses.hal.science/tel-02130460>

Submitted on 15 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS 13 - SORBONNE PARIS CITÉ (SPC)

présentée et soutenue publiquement par

Mohamed Mahdi BENMOUSSA

pour obtenir le titre de

Docteur de l'Université Paris 13-SPC

Approches pour la modélisation et vérification des systèmes temporisés en utilisant les diagrammes états-transitions et les réseaux Petri colorés

Soutenue le 06 Décembre 2016

Devant le jury composé de :

- Jean-Claude Royer Professeur, Rapporteur, École des Mines de Nantes
- Martin Wirsing Professeur, Rapporteur, Ludwig-Maximilians Universität
- Lom Messan Hillah Maître de conférences, Examineur, Université Paris Ouest
- Gianna Reggio Professeur, Examinatrice, Università di Genova
- Laure Petrucci Professeur, Présidente du jury, Université Paris 13-SPC
- Christine Choppy Professeur, Directrice, Université Paris 13-SPC
- Étienne André Maître de conférences, Co-Encadrant, Université Paris 13-SPC

Résumé

Nous présentons dans ce travail de thèse des approches pour la spécification et la vérification des systèmes temporisés. La première partie concerne une méthode de spécification en utilisant les diagrammes états-transitions pour modéliser un système donné en partant d'une description textuelle. Cette méthode guide l'utilisateur pour le développement de la modélisation. Elle comporte plusieurs étapes et utilise des observateurs d'états et des événements afin d'engendrer le diagramme états-transitions. Un outil qui implémente les différentes étapes de la méthode de spécification pour une application semi-automatique est présenté. La seconde partie concerne une traduction des diagrammes états-transitions vers les réseaux de Petri colorés, ce qui permet d'utiliser les méthodes de vérification. Nous prenons en considération dans cette traduction un ensemble important des éléments syntaxiques des diagrammes états-transitions, tels que la concurrence, la hiérarchie, etc. Un outil qui implémente la traduction pour un passage automatique des diagrammes états-transitions vers les réseaux de Petri colorés est en cours de développement. La dernière partie concerne l'intégration des contraintes temporelles dans les deux approches précédentes. Nous définissons des annotations pour les diagrammes états-transitions dont nous fournissons la syntaxe et la sémantique. Ces annotations seront ensuite utilisées dans la méthode de spécification et la traduction. Le but est de proposer des annotations faciles à comprendre et à utiliser avec une syntaxe qui prend en compte des contraintes parmi les plus utilisées.

Mots Clefs modélisation, diagrammes états-transitions UML, formalisation, sémantique formelle, réseaux de Petri colorés, contraintes temporelles

Abstract

In order to specify and verify timed systems, we present in this thesis approaches using UML state machines and coloured Petri nets. Our first approach is a specification method that takes into account a textual description of the system and generates the corresponding state machine diagram. This method helps a non-expert user to model a system in a structural way. We present a tool that implements the specification method. Our second approach is the translation of UML state machine diagrams to coloured Petri nets diagrams. In this approach we take into account an important set of UML state machine elements that allows the modelling of concurrent systems, etc. A tool that implements the approach and allows us to automate the translation is being developed. Finally, the last approach is the integration of time constraints in our specification method and in our translation. We propose a set of annotations to model time in state machine diagrams, and we define the corresponding syntax and semantics.

Keywords modelling, UML state machine diagrams, formalizing, formal semantics, coloured Petri nets, time constraints

Remerciements

Ce Rapport de thèse n'aurait pas été achevé sans l'aide d'un grand nombre de personnes. Je souhaite ici les remercier.

Je tiens tout d'abord à remercier chaleureusement mes encadrants Mme Christine Choppy et M. Étienne André qui par leur encouragement, leur soutien et leur disponibilité, ont été tout au long de mon travail d'un précieux apport dans la réalisation de ma thèse. Leur confiance et leurs conseils m'ont d'ailleurs permis d'acquérir et de développer un l'esprit scientifique, leur orientation m'a été également très utile pour le couronnement de mon travail dans ses différentes phases d'élaboration. Qu'ils trouvent ici un hommage vivant à leur haute personnalité.

Mes vifs remerciements vont également aux membres du jury pour l'intérêt qu'ils ont porté à mon travail en acceptant de l'examiner et de l'enrichir par leurs propositions.

M. Jean-Claude Royer pour ses remarques et ses corrections lors de ma soutenance mi-parcours ainsi que l'évaluation de mon rapport de thèse. La version annotée qu'il m'a transmise m'a été d'une grande aide.

M. Martin Wirsing pour ses remarques enrichissantes et sa disponibilité malgré ses nombreuses occupations. Ensuite Mme Gianna Reggio pour les différents travaux que nous avons effectués ensemble, l'accueil dont j'ai bénéficié dans son laboratoire.

Également M. Lom Messan Hillah pour son aide lors des différentes collaborations ou échange de mails et son évaluation lors de ma soutenance mi-parcours.

Enfin, Mme Laure Petrucci pour son accueil dans le laboratoire LIPN, son aide lors de mes différentes présentations, et sa disponibilité pour les différentes tâches administratives.

Mes remerciements s'étendent également aux différents membres de l'équipe LCR ainsi que les autres équipes du laboratoire, et ceux pour leur précieuse aide et leur accueil chaleureux durant ma thèse et durant mes différents enseignements.

Je n'oublie pas également les différents doctorants que ce soit du laboratoire LIPN ou des autres laboratoires (en particuliers de physique), pour leur aide, ainsi que l'ambiance conviviale avec laquelle ils m'ont comblé.

Mes remerciements vont également à toute l'équipe administrative du laboratoire, le BRED ainsi que l'école doctorale pour leur disponibilité et leurs efforts pour que mes différentes tâches administratives soient accomplies.

Je tiens à remercier Dieu le tout puissant qui m'a donné la chance, la force et la patience pour accomplir ce modeste travail. Que mes parents trouvent ici également une reconnaissance des efforts et la patience qu'ils m'ont offerte depuis ma naissance jusqu'à l'achèvement de ce travail. Quoique je dise, je suis incapable de les remercier assez, que dieu bénisse ma mère et que mon père repose en paix.

J'adresse mes plus sincères remerciements à tous mes proches que ce soit ma famille ou mes amis, qui m'ont toujours soutenu et encouragé au cours de la réalisation de ma thèse.

Merci à tous ceux que je n'ai pas cités, mais qui d'une façon ou d'une autre m'ont aidé pendant ce parcours.

Table des matières

1	Introduction	15
1.1	Contexte	15
1.2	Problématique	15
1.3	Objectif de la thèse	16
1.4	Contributions	16
1.5	Organisation du document	17
2	Concepts de base	18
2.1	Introduction	18
2.2	Diagrammes états-transitions UML (<i>SMD</i>)	19
2.2.1	Diagrammes états-transitions : Rappel	19
2.3	Réseaux de Petri colorés (CPN)	24
2.3.1	Définitions et notations	24
2.3.2	Définition formelle	26
2.3.3	CPNTools	28
2.4	Conclusion	30
3	État de l'art	31
3.1	Introduction	31
3.2	Formalisation d'UML	31
3.3	UML et temps	35
3.3.1	UML profile	35
3.3.2	MARTE pour UML	36
3.3.3	Transformation des diagrammes états-transitions vers les réseaux de Petri stochastiques	41
3.3.4	Validation des systèmes avec du temps en utilisant les statecharts et les automates temporisés	46
3.3.5	Model checking pour les diagrammes états-transitions et de collaborations avec du temps	50
3.3.6	Vérification formelle pour les systèmes temps réel	53
3.3.7	SCCharts pour la vérifications des systèmes réactifs	53
3.3.8	Patterns pour les systèmes temps réel	53
3.3.9	Patterns pour la modélisation des systèmes temporisés	55
3.3.10	Passage des contraintes de planning vers les automates temporisés	56
3.3.11	Le profile UML hybride pour UML 2.0	57
3.4	Méthodes de spécification	59
3.4.1	Concurrence et types de données : méthode de spécification	59
3.4.2	Conception des systèmes adaptatifs avec du temps	59

3.4.3	Modélisation des systèmes temps réel en utilisant UML et des annotations formelles	60
3.4.4	UML statecharts mobile	60
3.4.5	Une approche de modélisation en utilisant les réseaux de Petri colorés	61
3.4.6	Conception des systèmes basés agents en utilisant UML	62
3.5	Conclusion	63
4	Méthode de spécification	64
4.1	Introduction	64
4.2	Concepts utilisés dans la méthode de spécification	65
4.3	Méthode de spécification	66
4.4	Détails de la méthode	67
4.4.1	Vue générale de la méthode	67
4.4.2	Analyse du texte	68
4.4.3	Recherche des observateurs d'état [?]	70
4.4.4	Recherche des événements [?]	71
4.4.5	Construction du diagramme de séquence	72
4.4.6	Recherche des invariants des observateurs d'état [?]	72
4.4.7	Recherche des conditions / réactions des événements [?]	74
4.4.8	Construction de la table des états [?]	75
4.4.9	Recherche des transitions [?]	76
4.4.10	Génération du diagramme états-transitions [?]	77
4.4.11	Vérification de la cohérence	77
4.4.12	Améliorations apportées à la méthode de spécification	78
4.5	Étude de cas : exemple du paiement du parking	80
4.5.1	Analyse du texte et diagramme de classes	80
4.5.2	Observateurs d'état et événements	81
4.5.3	Invariants et réactions	84
4.5.4	Vérification en utilisant le diagramme de séquence	96
4.6	Hierarchie des états : états composites et états orthogonaux	97
4.7	Qualité du diagramme états-transitions	104
4.7.1	Métriques	105
4.7.2	Critères de qualité	106
4.7.3	Améliorations	107
4.8	Expérimentation	109
4.9	Outil support de la méthode : <i>EasySM</i>	113
4.9.1	EasySM : première version	113
4.9.2	EasySM : améliorations	118
4.10	Conclusion	120
5	Traduction	121
5.1	Introduction	121
5.2	Formalisation et hypothèses	122
5.3	Schéma général	125
5.4	Places et transitions : nommage	128
5.5	Algorithme de traduction	128
5.6	L'ajout des comportements au code des transitions	132
5.7	Application de l'algorithme à des cas simples	134
5.7.1	États simples et comportements "Do"	135

5.7.2	Transitions sans événements avec régions orthogonales	135
5.7.3	Transitions avec événement et régions orthogonales	135
5.7.4	Hiérarchie des régions orthogonales	137
5.7.5	Pseudo-état fork	137
5.7.6	Pseudo-état histoire	137
5.8	Au-delà de notre traduction	140
5.8.1	Supprimer les hypothèses	140
5.8.2	Extension de la syntaxe	141
5.9	Sélection des transitions	142
5.9.1	Gestion des événements	143
5.9.2	Gestion des priorités	144
5.10	Conclusion	154
6	Étude de cas	155
6.1	Introduction	155
6.2	Étude de cas : lecteur de CD	155
6.2.1	Description	155
6.2.2	Traduction du diagramme états-transitions du lecteur de CD	157
6.2.3	Limites des formalismes discrets pour des comportements continus	158
6.2.4	Modification du modèle	158
6.2.5	Vérification	159
6.2.6	Étude du passage à l'échelle	164
6.3	Conclusion	165
7	Implémentation de la traduction	166
7.1	Introduction	166
7.2	Implémentation	167
7.3	UML to CPN tool	168
7.4	Conclusion	174
8	Expression temporelle	175
8.1	Introduction	175
8.2	Choix des contraintes temporelles	176
8.3	Annotations temporelles	176
8.3.1	Classification	176
8.3.2	Syntaxe et sémantique des annotations temporelles	177
8.3.3	Exprimer le temps dans les états ou dans les transitions?	182
8.4	Unités de temps	183
8.5	Comparaisons	184
8.6	Exemples : prise en compte des contraintes temporelles	186
8.7	Le temps dans les réseaux de Petri colorés	191
8.8	Introduction des contraintes temporelles dans la traduction	194
8.9	Conclusion	200
9	Conclusion	201
9.1	Résultats	201
9.2	Perspectives à court terme	202
9.3	Perspectives à long terme	202

A		204
A	Rapport engendré par l'outil <i>EasySM</i>	204
B	Questionnaires de l'expérience des métriques	220
C	Exemples de l'expérience sur les métriques	227

List of Algorithms

1	Translating an SMD $(\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \text{SubStates}, \mathcal{T})$ into a CPN	131
2	Gestion des priorités de franchissement	147

Liste des tableaux

4.1	Observateurs d'état	70
4.2	Gestion de la bibliothèque[?] : observateurs d'état	71
4.3	Événements	71
4.4	Gestion de la bibliothèque[?] : événements	72
4.5	Invariants des observateurs d'état	73
4.6	Gestion de la bibliothèque[?] : invariants	74
4.7	Conditions/réactions des événements	74
4.8	Gestion de la bibliothèque [?] : conditions/réactions des événements	75
4.9	États	76
4.10	Gestion de la bibliothèque [?] : table des états	76
4.11	Événements et observateurs d'état	82
4.12	Observateurs d'état et invariants	88
4.13	Événements et Conditions/Réactions (Partie 1)	90
4.14	Événements et Conditions/Réactions (Partie 2)	91
4.15	Les états du diagramme états-transitions	92
4.16	Table des états composites	100
4.17	Table des états composites : mise à jour	102
4.18	Exemples utilisés dans l'expérimentation	109
5.1	Conventions du nommage pour les éléments <i>CPN</i>	128
5.2	Résumé des aspects syntaxiques considérés	143
6.1	Performances de la génération de l'espace d'état du lecteur de CD	165
8.1	Syntaxes des terminaux utilisés dans la grammaire de la Figure 8.4	178
8.2	Comparaison	185
A.1	Exemples utilisés dans l'expérimentation	227

Table des figures

2.1	Exemple d'un diagramme états-transitions	19
2.2	Notation des variables globales et la sémantique correspondante	24
2.3	Exemple de réseau de Petri coloré modélisé en utilisant CPN Tools	25
2.4	Évolution du marquage : Franchissement de la transition t1	28
2.5	Évolution du marquage : Franchissement de la transition t3	28
2.6	L'architecture de CPN Tools	29
2.7	Interface Standard de CPN tools	29
2.8	Exemple dans CPN tools	30
3.1	Le temps et les durées dans UML	36
3.2	Grammaire de la classification temporelle proposée dans Figure 8.1	36
3.3	MARTE et les standards de l'OMG	37
3.4	L'architecture de MARTE.	37
3.5	La structure du temps dans MARTE ([?]).	38
3.6	L'horloge dans MARTE ([?]).	39
3.7	TimeValue dans MARTE ([?]).	39
3.8	Liaison du temps avec l'événement et le comportement	40
3.9	Exemple d'utilisation du temps en MARTE dans les diagrammes états-transitions	41
3.10	Diagramme états transitions UML avec le temps	42
3.11	Réseaux de Petri stochastiques	43
3.12	Traduction d'un état	43
3.13	Traduction d'un pseudo-état initial	44
3.14	Traduction des pseudo-états Fork et Join	44
3.15	Traduction d'un pseudo-état choice	45
3.16	Traduction d'un point de jonction	45
3.17	Traduction d'une transition avec événement	46
3.18	Le processus général de l'utilisation des schémas [?]	47
3.19	Le processus de vérification [?]	48
3.20	Classification des contraintes de temps [?]	49
3.21	Pattern de la contrainte "Delay" [?]	49
3.22	Architecture de la transformation des SMD vers les TA [?]	50
3.23	Annotation temporelle de l'exemple du passage de train [?]	50
3.24	Diagramme de classes de l'exemple du train [?]	51
3.25	Diagramme états-transitions de la barrière [?]	51
3.26	Diagramme états-transitions du parcours du train [?]	51
3.27	Diagramme états-transitions du contrôleur [?]	51
3.28	Diagramme de séquence pour la propriété de sureté et d'utilité [?]	52
3.29	Exemple diagramme états-transitions [?]	52
3.30	Traduction de l'exemple de la Figure 3.29 [?]	52
3.31	Exemple de SCCharts [?]	54

3.32	Règles de transformations des SCCharts vers les SC graphs [?]	55
3.33	Observateur automate temporisé pour la propriété de précédence[?]	55
3.34	Observateur automate temporisé pour la propriété de vivacité [?]	55
3.35	Observateur automate temporisé pour la propriété de ponctualité [?]	55
3.36	Observateur automate temporisé pour la propriété de précédence avec un intervalle [?]	56
3.37	Observateur automate temporisé pour la propriété de latence [?]	56
3.38	Observateur automate temporisé pour la propriété de séquence [?]	56
3.39	Traduction des différents patterns vers les réseaux de Petri temporisés [?]	57
3.40	Relations temporelles basées sur les intervalles [?]	58
3.41	La méthode pour la spécification <i>LOTOS</i>	59
3.42	L'automate de l'exemple de l'hôpital	60
3.43	Diagrammes de classes du service de réseau	61
3.44	Étapes de la méthode	61
3.45	Exemple du train électrique : réseau de Petri coloré	62
4.1	Schéma des étapes de la méthode de spécification [?]	67
4.2	Diagramme d'activité des étapes de la méthode	69
4.3	Diagramme de classes de la gestion de bibliothèque	70
4.4	Diagramme de séquence	73
4.5	Modèle de recherche d'une transition	77
4.6	Diagramme de classes de l'exemple du paiement du parking	81
4.7	Diagramme de classe de l'exemple du paiement du parking : mise à jour des nouvelles classes	83
4.8	Diagramme de séquence du déroulement du système	84
4.9	Diagramme de classe de l'exemple du paiement du parking : mise à jour des classes avec les associations	89
4.10	La première étape pour trouver la transition de l'événement <code>ticketInserted</code>	93
4.11	La seconde étape pour trouver la transition de l'événement <code>ticketInserted</code>	93
4.12	La seconde étape pour trouver la transition de l'événement <code>card_Inserted</code>	93
4.13	La seconde étape pour trouver la transition de l'événement <code>card_Inserted</code>	93
4.14	Diagramme de classes final de l'exemple du paiement du parking	94
4.15	Diagramme états-transitions de l'exemple du paiement du parking	95
4.16	Diagramme de séquence et états-transitions : Comparaison	96
4.17	Diagramme états-transitions de la barrière de péage : nouvelle version	98
4.18	Exemple de transition vers état simple dans un état composite	101
4.19	Exemple de transition (avec ou sans événement) originaire d'un état simple dans un état composite	101
4.20	Exemple d'application de la règle 3 sur un diagramme états-transitions	102
4.21	Diagramme états-transitions de la barrière de péage avec hiérarchie	103
4.22	Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)	103
4.23	Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)	104
4.24	Exemple de diagramme états-transitions avec et sans amélioration de la duplication	108
4.25	Exemple d'introduction de la hiérarchie	108
4.26	Exemple d'introduction de transitions avec plusieurs événements	108
4.27	Exemple d'ajout de pseudo-état initial	109
4.28	Exemple d'ajout d'état final	109
4.29	Exemple de la barrière sans hiérarchie et avec hiérarchie	110
4.30	Exemple du thread avec des événements dans des transitions différentes	111

4.31	Exemple du thread avec des événements dans une seule transition	111
4.32	Exemple de la playlist avec des états dupliqués	112
4.33	Exemple de la playlist sans états dupliqués	112
4.34	Interface de base de <i>EasySM</i>	113
4.35	Édition du diagramme de classes et de la classe de contexte dans <i>EasySM</i>	114
4.36	Édition des observateurs d'états dans <i>EasySM</i>	115
4.37	Édition des événements dans <i>EasySM</i>	115
4.38	Édition de la table des états dans <i>EasySM</i>	116
4.39	Diagramme états-transitions dans <i>EasySM</i>	117
4.40	Mise à jour du diagramme de classes dans <i>EasySM</i>	118
4.41	Manuel utilisateur dans <i>EasySM</i>	119
4.42	Le bouton <i>Export Rapport</i> dans <i>EasySM</i>	119
4.43	Menu déroulant des propositions de types dans <i>EasySM</i>	120
5.1	Exemple d'un diagramme états-transitions sans concurrence et avec des comportements	132
5.2	Exemple d'un diagramme états-transitions avec de la concurrence et des comportements	133
5.3	Exemple avec modification concurrente des variables	134
5.4	Exemple d'états simples et de comportement "Do"	135
5.5	Exemple : transitions sans événements avec des régions orthogonales	136
5.6	Exemple : transitions avec événements et régions orthogonales	136
5.7	Exemple : hiérarchie des régions orthogonales	137
5.8	Exemple : pseudo-état fork	138
5.9	Exemple : pseudo-état histoire	138
5.10	Exemple : pseudo-état histoire avec la concurrence	139
5.11	Modélisation de la place des événements et de l'arrivée aléatoire des événements	144
5.12	Exemple avec priorités	145
5.13	Un exemple d'une transition fusionnée pour l'événement a	145
5.14	Exemple avec des priorités et de la concurrence	148
5.15	Exemple avec des priorités et de la concurrence : traduction	148
5.16	Modélisation d'une transition de terminaison	149
5.17	Exemple avec les priorités : traduction complète sans code	149
5.18	Exemple avec les priorités : traduction complète avec code	149
6.1	Exemple du lecteur de CD	156
6.2	Exemple du lecteur de CD (version modifiée)	159
6.3	Traduction en <i>CPN</i> de la version révisée de l'exemple du lecteur de CD : version sans code	160
6.4	Traduction en <i>CPN</i> de la version révisée de l'exemple du lecteur de CD : version simplifiée	161
6.5	Traduction en <i>CPN</i> de la version révisée de l'exemple du lecteur de CD : version complète	162
6.6	Évaluation des propriétés dans <i>CPN Tools</i>	164
7.1	Exemple du code <i>Acceleo</i>	167
7.2	Fichiers <i>JDOM</i>	169
7.3	Packages et classes utilisés dans <i>UML2CPN</i>	170
7.4	L'interface de l'outil <i>UML2CPN</i>	171
7.5	La récupération du fichier <i>XMI</i> dans <i>UML2CPN</i>	172

7.6	La récupération des données du SMD à partir du fichier <i>XMI</i> dans UML2CPN .	172
7.7	La traduction du diagramme états-transitions vers le réseaux de Petri coloré correspondant dans UML2CPN (version non graphiques)	173
7.8	La génération du fichier CPN Tools dans UML2CPN (module en cours de développement)	173
8.1	Nouvelle classification des besoins temporels	177
8.2	Exemple d'explication pour l'élément syntaxique after	180
8.3	Exemple d'explication pour l'élément syntaxique before	180
8.4	Grammaire de la classification temporelle proposée dans Figure 8.1	181
8.5	Représentation des contraintes temporelles dans les diagrammes états-transitions	181
8.6	Gestion de l'échec des contraintes temporelles	182
8.7	Le temps dans les transitions	183
8.8	Le temps dans les états	183
8.9	Exemple de la barrière de parking temporisé	187
8.10	L'exemple de la bibliothèque temporisé	188
8.11	L'exemple de location de scooter temporisé (version 1)	189
8.12	L'exemple de location de scooter temporisé (version 2)	190
8.13	Exemple d'un réseau de Petri coloré avec du temps ([?])	192
8.14	Exemple d'un réseau de Petri coloré avec du temps avec marquage ([?])	192
8.15	Franchissement de la transition SendPacket	193
8.16	Valeur de l'horloge globale avant et après le franchissement de la transition SendPacket	193
8.17	Gestion des estampillages de date dans les transitions	194
8.18	Traduction d' at_least	195
8.19	Traduction d' at	196
8.20	Traduction d' at_most	197
8.21	Traduction d' after	198
8.22	Traduction de before	199
A.1	Exemple de l'article avec des états dupliqués	227
A.2	Exemple de l'article sans états dupliqués	227
A.3	Exemple de l'inscription dans un séminaire sans hiérarchie	228
A.4	Exemple de l'inscription dans un séminaire avec hiérarchie	228
A.5	Exemple de la vie d'un objet avec des événements dans des transitions différentes	229
A.6	Exemple de la vie d'un objet avec des événements dans une seul transition	229

Chapitre 1

Introduction

Contents

1.1	Contexte	15
1.2	Problématique	15
1.3	Objectif de la thèse	16
1.4	Contributions	16
1.5	Organisation du document	17

Contexte

Avec l'augmentation de la complexité des systèmes concurrents, une phase de validation des systèmes est nécessaire. Plusieurs solutions existent afin de vérifier ces systèmes. La phase de modélisation est devenue l'une des solutions les plus utilisées afin de réduire les coûts induits par les problèmes de développement logiciel et afin de détecter les erreurs avant la phase d'implémentation.

Le langage UML (Unified Modeling Language) est devenu le standard de la modélisation des systèmes et cela dans le domaine de la recherche ou de l'industrie. Proposé par l'OMG (Object Management Group) [?], UML comporte une syntaxe riche et un ensemble de diagrammes permettant ainsi la modélisation des différents aspects des systèmes. Parmi ces diagrammes, les diagrammes états-transitions qui sont des systèmes à transitions pour exprimer les comportements dynamiques. Les diagrammes états-transitions permettent une représentation graphique précise à même de spécifier le comportement des systèmes concurrents. Leur syntaxe offre des éléments puissants tels que la concurrence, la hiérarchie, les comportements.

Afin de réduire les coûts induits par les problèmes de développement logiciel, il est nécessaire de détecter les erreurs de modélisation avant la phase d'implémentation. Ces erreurs peuvent être très coûteuses en terme de développement : ce n'est parfois qu'une fois l'implémentation réalisée et testée que l'erreur est détectée ; donc le processus entier doit alors être repris de zéro.

Problématique

Malgré son utilisation large dans l'industrie ou dans la recherche, la sémantique d'UML dans la spécification de l'OMG est décrite en langage naturel et donc elle n'est que semi-formelle. Il

1 existe dans la littérature des travaux visant à donner une sémantique formelle aux diagrammes
2 états-transitions. Cependant, ces travaux ne sont pas intégrés dans le standard d'UML (pro-
3 posé par l'OMG) d'une part. D'autre part, il y a beaucoup d'ambiguïtés dans la sémantique
4 semi-formelle proposée par l'OMG, et par conséquent il est très difficile de savoir quelle sémantique
5 formelle (proposées dans la littérature) choisir. L'absence de sémantique formelle standard
6 rend les diagrammes états-transitions non adaptés directement à des techniques de vérification
7 formelle.

8 Objectif de la thèse

9 Dans le cadre de ce travail de thèse, nous proposons une solution pour la modélisation des
10 systèmes en utilisant les diagrammes états-transitions et une solution pour vérifier ces systèmes en
11 traduisant les diagrammes états-transitions vers les réseaux de Petri colorés. Le premier objectif
12 est de proposer une méthode de spécification qui fournit des lignes directrices aux non-experts
13 souhaitant concevoir un diagramme états-transitions UML modélisation un système. Le second
14 objectif est de proposer une formalisation des diagrammes états-transitions afin de pouvoir les
15 vérifier. Cette formalisation est effectuée avec une traduction des diagrammes états-transitions
16 vers les réseaux de Petri colorés (un formalisme qui permet de faire la vérification formelle).
17 D'autres objectifs entrent dans le cadre de cette thèse qui sont l'implémentation des différents
18 travaux proposés mais également faire de la vérification sur des études de cas.

19 Contributions

20 Contribution 1

21 Nous proposons dans cette thèse comme première contribution une méthode de spécification
22 afin de guider les utilisateurs pendant la phase de modélisation. Cette méthode est structurée
23 et comporte différentes étapes traitant point par point les différentes parties du systèmes.
24 L'entrée est la description textuelle du système (en langage naturel) et la sortie le diagramme
25 états-transitions correspondants. Dans la méthode de spécification nous considérons des élé-
26 ments syntaxiques des diagrammes états-transitions tels que la hiérarchie, les états simples et
27 composites, les transitions etc. Nous proposons aussi un outil semi-automatique (*EasySM*) qui
28 implémente la méthode et permet l'application des différentes étapes.

29 Contribution 2

30 Nous proposons comme seconde contribution une traduction des diagrammes états-transitions
31 vers les réseaux de Petri colorés. Ce passage vers les réseaux de Petri colorés est basé sur une
32 sémantique implémentée en utilisant deux algorithmes. Nous considérons un ensemble d'éléments
33 syntaxiques, par exemple : les états simples, les états finaux, les pseudo-états initiaux, les pseudo-
34 états histoires (uniquement plats), les comportements (entrée, sortie et do) et les variables, les
35 transitions (simples, composites, avec événement, garde, effet, locaux, externes et inter-niveau).
36 Nous considérons également des éléments plus complexes et plus importants pour modéliser les
37 systèmes tels que : la concurrence, les états orthogonaux, les pseudo-états fork, la hiérarchie des
38 états et des comportements ainsi que l'algorithme de sélection de transitions. Plus il y a des
39 éléments syntaxiques considérés, plus la difficulté de faire la traduction augmente. En effet, la
40 représentation graphiques des diagrammes états-transitions est simple et facile à utiliser, cepen-
41 dant la sémantique correspondante est très compliqué et parfois ambiguë. Afin d'automatiser
42 notre traduction et par conséquent faciliter l'application de nos algorithmes nous proposons un

1 nouvel outil. Nous avons implémenté une première version de notre traduction dans un nouveau
2 prototype UML2CPN. Nous utilisons le langage *Java* pour l'implémentation. Notre outil prend
3 en entrée un diagramme états-transitions édité en utilisant *Papyrus* et engendre le réseau de Petri
4 coloré correspondant (le fichier sera compatible avec l'outil CPN Tools qui est un outil principal
5 pour la simulation et la vérification des réseaux de Petri colorés).

6 **Contribution 3**

7 Nous proposons comme troisième contribution de cette thèse une approche préliminaire afin
8 d'étendre la syntaxe d'UML pour autoriser certaines expressions temporelles dans les diagrammes
9 états-transitions ; nous considérons ensuite l'intégration de nos contraintes temporelles à la fois
10 dans la méthode de spécification, et dans la traduction vers les réseaux de Petri colorés tempo-
11 risés.

12 **Organisation du document**

13 Dans ce document nous présentons dans le [Chapitre 2](#) les concepts de base utilisés dans
14 nos contributions à savoir les diagrammes états-transitions et les réseaux de Petri colorés. Le
15 [Chapitre 3](#) comportera les différents travaux de l'état de l'art concernant aspects que notre
16 thèse traite, tels que la vérification des diagrammes états-transitions, les méthode de spécifica-
17 tions et l'intégration des contraintes temporelles dans les diagrammes états-transitions. Dans le
18 [Chapitre 4](#), nous présentons la méthode de spécification que nous utilisons et les différentes amé-
19 liorations que nous apportons à cette dernière. Nous présentons également l'outil *EasySM* (qui
20 implémente la méthode de spécification) avec les différentes améliorations apportées à ce dernier.
21 Dans le chapitre suivant ([Chapitre 5](#)), nous présentons notre principale contribution concernant
22 la traduction des diagrammes états-transitions vers les réseaux de Petri colorés. Ensuite nous pré-
23 sentons dans le [Chapitre 6](#) une étude de cas avec son diagramme états-transitions correspondant
24 et le réseau de Petri coloré résultat de l'application de notre traduction. La suite du document
25 comportera le [Chapitre 7](#) sur l'implémentation de notre traduction et le [Chapitre 8](#) sur l'inté-
26 gration des contraintes temporelles dans nos approches. Enfin, nous présentons une conclusion
27 et des perspectives dans le [Chapitre 9](#).

Chapitre 2

Concepts de base

Contents

2.1	Introduction	18
2.2	Diagrammes états-transitions UML (<i>SMD</i>)	19
2.3	Réseaux de Petri colorés (CPN)	24
2.4	Conclusion	30

Introduction

Nous présentons dans ce chapitre les différents concepts de base que nous utilisons dans nos contributions. Nous abordons dans la [Section 2.2](#) les diagrammes états-transitions UML (une partie concernera un rappel et l'autre la formalisation), et les réseaux de Petri colorés dans la [Section 2.3](#).

1 Diagrammes états-transitions UML (SMD)

2 Nous présentons d'abord dans cette section un rappel des diagrammes états-transitions pro-
 3 posés par l'OMG (Section 2.2.1); ensuite nous formalisons la syntaxe des diagrammes états-
 4 transitions en introduisant des hypothèses (Section 5.2). Nous nous concentrons sur les éléments
 5 qui nécessitent de la précision et nous présentons uniquement les éléments que nous prenons en
 6 considération dans la traduction.

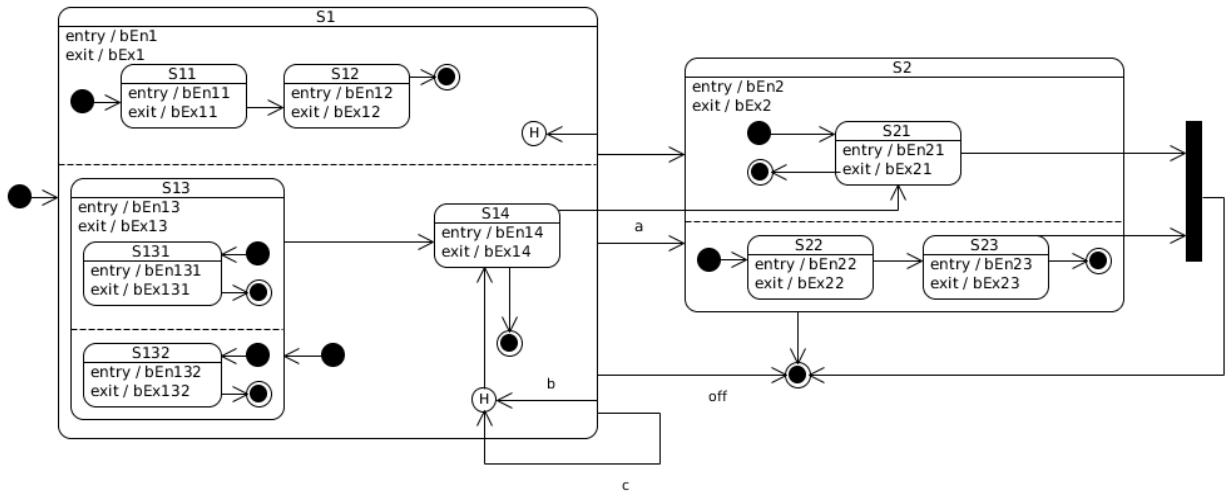


FIGURE 2.1 – Exemple d'un diagramme états-transitions

7 Diagrammes états-transitions : Rappel

8 Les diagrammes états-transitions [?] sont des automates finis, tels que chaque entité (ou sous
 9 entité) se trouve dans un seul état (situation) et peut passer dans un autre état à n'importe quel
 10 moment par une transition. UML fournit un ensemble de constructions pour les diagrammes
 11 états-transitions. Dans ce qui suit, nous rappelons ces constructions. Nous utilisons l'exemple de
 12 la Figure 2.1 afin de donner plus détails pour chaque construction (ou élément). Cet exemple
 13 comporte les éléments syntaxiques les plus importants que nous prenons en considération dans
 14 notre traduction.

15 États

16 UML considère trois types d'états (qui peuvent contenir ou non des régions) : états *simples*
 17 (par exemple, S11 dans la figure Figure 2.1), états *composites* (par exemple, S1 et S13 dans la
 18 Figure 2.1) et les états *submachines*.

19 Un état *simple* ne contient pas de régions, d'états internes ou de transitions. Un état *composite*
 20 est un état qui contient au moins une région et peut être ou bien un état composite *simple* ou
 21 un état *orthogonal*. Un état composite simple contient exactement une seule région contenant
 22 d'autres états; cela permet d'avoir une hiérarchie dans les diagrammes états-transitions. Un
 23 état orthogonal (par exemple, les états S1 et S13 dans la Figure 2.1) contient plusieurs régions
 24 (chaque région peut contenir un ensemble d'états). La présence de plusieurs régions permet
 25 d'avoir la notion de concurrence (car les états dans chaque régions sont exécutés en parallèle). Les
 26 régions sont délimitées par des lignes pointillées. Chaque état composite ne doit pas être vide ("A

1 composite State contains at least one region” [?, Section State, p.306]), et chaque région contient
2 au moins un état. Soit une région d’un état composite :

- 3 • Ses *sous-états directs* sont l’ensemble des états immédiatement contenus dans cette région (par
4 exemple, dans la [Figure 2.1](#) les sous-états directs de la région inférieure de l’état **S1** sont **S13**,
5 **S14** et l’état final de la région).
- 6 • Ses *sous-états indirects* sont des états obtenus par la fermeture transitive de la relation des
7 sous-états (par exemple, dans la [Figure 2.1](#) les sous-états indirects de la région inférieure de
8 l’état **S1** sont **S13**, **S131**, **S132**, **S14** et les trois états finaux).

9 Un état *submachine* est une machine à état qui peut être contenue dans un état (composite
10 simple ou composite orthogonal).

11 Nous appelons un état *racine* un état contenu dans la région supérieure du diagramme états-
12 transition (par exemple, les états **S1** et **S2** dans la [Figure 2.1](#)). La région supérieure du diagramme
13 états-transitions est la région directe contenue dans la machine à état (“topmost Region (i.e., a
14 Region owned by the StateMachine)” [?, Compound transitions, p.313]).

15 États finaux

16 Un état final (par exemple, l’état le plus à droite dans l’état **S2** de la [Figure 2.1](#)) est un état
17 particulier dans lequel on entre lorsque l’exécution dans la région le contenant est terminée.

18 Comportements

19 Les comportements sont définis soit à l’entrée dans un état, soit à la sortie d’un état, soit
20 pendant la présence d’un état (quand l’état a un comportement *do*) ou pendant le franchissement
21 d’une transition. Un comportement d’entrée est exécuté quand on entre dans l’état par une
22 transition externe (c’est-à-dire quand la flèche traverse la bordure de l’état ou bien la touche de
23 l’extérieur, voir ci-dessous), ou bien si l’état est la cible d’une transition à partir d’un pseudo-état
24 initial appartenant à un état composite qui dans lequel on est entré par une transition externe.
25 Le comportement de sortie est exécuté soit quand l’état sort à travers une transition externe, soit
26 dans le cas où il appartient à un état composite qui est quitté. Le comportement *do* est exécuté
27 après que l’état ait exécuté son comportement d’entrée, et il continue de s’exécuter (en parallèle
28 avec d’autres comportements, s’ils existent) jusqu’à ce qu’il ait terminé ou bien qu’on sorte de
29 l’état.

30 Pseudo-états

31 La différence entre un état et un pseudo-état est que l’état actif du système peut être un état
32 (ou un ensemble d’états), mais pas un pseudo-état (“In general, Pseudostates and Connection-
33 PointReferences are transitive, in the sense that a compound transition execution simply passes
34 through them [...] State and FinalState, however, represent stable Vertices, such that, when a
35 StateMachine execution enters them it remains in them until [...]” [?, Section Vertices, p.305]).
36 UML définit différents pseudos-états qui sont décrits comme suit.

37 **Initial** Un pseudo-état initial est le point de départ d’une région d’un état composite ou d’une
38 machine à états. Chaque état composite (simple ou orthogonal) et chaque machine à états
39 peut avoir un pseudo-état initial. Dans le cas où il n’y a pas de pseudo-état initial alors une
40 possibilité est de considérer que le modèle (le diagramme états-transitions) est mal défini
41 (“However, no specific approach is defined if there is no initial Pseudostate that exists

within the Region. One possible approach is to deem the model ill defined”[?, Section Regions, p.305]). Une autre alternative est que la région reste inactive même si l’état qui la contient est actif, c’est-à-dire l’état composite qui contient cette région est considéré comme un état simple (“An alternative is that the Region remains inactive, although the State that contains it is active. In other words, the containing composite State is treated as a simple (leaf) State.”[?, Section Regions, p.305]). La transition allant du pseudo-état initial peut avoir un comportement mais ne peut pas avoir d’événement ni de garde (une expression booléenne qui définit la condition selon laquelle la transition peut être franchie).

Histoire Seulement les états composites peuvent avoir au maximum un pseudo-état histoire dans chaque région (“A deepHistory Pseudostate can only be defined for composite States and, at most one such Pseudostate can be contained in a Region of a composite State”[?, Section Pseudostate and PseudostateKind, p.310]).UML propose deux types de pseudo-états histoire : pseudo-état histoire plat (“*shallow history*”) et pseudo-état profond (“*deep history*”). Un pseudo-état histoire plat est une variable qui représente le dernier état visité (dans l’état qui le contient) et non pas les sous-états de cet état “is a kind of variable that represents the most recent active state configuration of its containing State, but not the substates of that substate” [?, Section Pseudostate and PseudostateKind, p.310], ce qui signifie que le pseudo-état sauvegarde uniquement le dernier état visité dans l’état composite qui le contient.Un pseudo-état histoire plat est représenté par un “H”. Dans la [Figure 2.1](#) il y a un pseudo-état histoire dans chaque région de l’état S1.

Un pseudo-état histoire profond permet de sauvegarder la plus récente configuration d’état de tous les états visités dans l’état composite qui le contient (la configuration d’état est restaurée quand une transition arrive vers le pseudo-état).

Fork et Join UML propose deux types de pseudo-états qui permettent la fusion ou la jointure des transitions. Le pseudo-état join permet à plusieurs transitions (qui sont originaires de différentes régions orthogonales) d’être fusionnées en une seule transition. La transition allant du pseudo-état join est exécutée seulement après l’exécution des transitions allant vers le pseudo-état. Une transition qui arrive à un pseudo-état join, n’a ni garde ni événement. Un pseudo-état fork permet de diviser une transition en plusieurs transitions allant vers des états dans différentes régions. La transition allant d’un pseudo-état fork n’a ni garde ni événement. Par exemple, dans la [Figure 2.1](#), quand le système est en même temps en S21 et S23, le pseudo-état join peut directement mener à un état final. Nous ne prenons pas en compte les pseudos-états fork et join implicites, c’est-à-dire qui n’entrent pas (ou ne sortent pas) explicitement dans toutes les régions d’un état composite. Dans un état composite avec trois régions, un pseudo-état fork implicite peut avoir deux transitions entrantes, une sortante de la première région, la seconde de seconde région, mais pas de transitions partant de la troisième région.

Transitions

Type des transitions

Dans UML il y a trois types de transitions : externes, locales et internes. Il y a une ambiguïté concernant les types de transitions dans la spécification UML, et cela avec une contradiction (Section 14.2.3.8.1 p.312 et Section 14.2.4.10 p.332 dans [?]).Nous avons choisi de suivre la sémantique (informelle) définie dans [?, Section 14.2.4.10 p.332], qui fait référence a la représentation graphique de la transition et qui est plus claire que la description textuelle.

- Une transition *locale* peut être une transition allant de la bordure de l’état composite qui la contient ou une transition entre deux sous-états d’un état composite “can originate from the

border of the containing composite State, or one of its entry points, or from a Vertex within the composite State” ([?, p.332]). Elle n’exécute ni le comportement de sortie ni le comportement d’entrée de l’état qui la contient. Graphiquement, la flèche touche l’intérieur de l’état et ne le traverse pas sa bordure. Par exemple, la transition vers le pseudo-état histoire dans la seconde région de l’état S1 (la région du bas) avec l’événement **b** dans la Figure 2.1 est une transition locale.

- Une transition *interne* est un cas particulier d’une transition locale. C’est une transition qui a le même état comme source et comme destination. Elle n’est pas représentée explicitement dans un diagramme états-transitions ([?, p.332]), mais elle peut être définie dans le compartiment interne d’un état ([?, Section 14.2.4.5, p.317]). Afin de garder notre traduction claire, nous avons choisi de ne pas considérer les transitions internes et nous présentons dans la section Section 5.8.2 comment les prendre en compte.
- Une transition *externe* est une transition qui n’est ni une transition locale ni une transition interne car elle exécute le comportement de sortie de son état source et le comportement d’entrée de son état destination. Par exemple, la transition vers le pseudo-état histoire de la seconde région (celle du bas) de l’état S1 avec l’événement **c** dans la figure Figure 2.1 est une transition externe. Toutes les transitions de la Figure 2.1 sont externes, avec l’exception des deux transitions internes (la première dans la première région allant vers le pseudo-état histoire et la seconde région avec l’événement **b**).

Une transition de *haut niveau* est une transition qui a comme source un état composite. Nous soulignons aussi la transition « inter-niveaux », un type de transition qui traverse les bordures des états composites. Par exemple, la transition allant de S14 vers S121 dans la figure Figure 2.1 est une transition inter-niveau. Beaucoup de travaux dans la littérature ne prennent pas en compte les transitions inter-niveaux (par exemple, [?, ?, ?, ?, ?]), mais nous considérons cette transition dans notre traduction.

Syntaxe des transitions

Chaque transition peut avoir une garde : une expression booléenne qui définit la condition sous laquelle la transition peut être franchie (par exemple, la garde `track ≠ trackCount` de la transition allant de BUSY vers BUSY dans la Figure 6.1), un événement (par exemple, l’événement `stop` de la transition entre BUSY et NONPLAYING dans la Figure 6.1), un comportement (par exemple, le comportement `track ++` de la transition entre BUSY et BUSY dans la Figure 6.1), et peut être une transition de terminaison (une transition sans événement) ou une transition avec événement.

Ordre d’exécution des comportements au franchissement d’une transition

L’ordre d’exécution des comportements de sortie pendant la sortie d’état à travers une transition est le suivant : si la source de la transition (une transition de terminaison ou une transition avec événement) est un état simple alors nous avons besoin d’exécuter uniquement le comportement de sortie de cet état. Si l’état source de la transition est un état composite, alors dans le cas d’une transition de terminaison on exécute uniquement le comportement de sortie de l’état composite. Dans le cas d’une transition avec événement on exécute les comportements de sortie en allant de l’état le plus intérieur jusqu’à l’état composite source de la transition (“When exiting from a composite State, exit commences with the innermost State in the active state configuration. This means that exit Behaviors are executed in sequence starting with the innermost active State.” [?, Section State, p.308]). Cette règle s’applique pour chaque région de l’état composite, dans le cas où c’est un état orthogonal.

1 L'ordre d'exécution des comportements d'entrée pendant l'entrée dans un état à travers une
2 transition est le suivant : si la destination de la transition est un état simple alors on exécute
3 le comportement d'entrée de cet état. Si l'état destination est un état composite avec une seule
4 région alors on exécute en premier le comportement d'entrée de cet état ensuite on exécute en
5 séquence les comportements d'entrée de tous ses sous-états (directs ou indirects) qui sont des
6 états initiaux (états liés avec les pseudos-états initiaux).

7 **Run-to-completion step**

8 Le *run-to-completion step* dans les diagrammes états-transitions est l'exécution complète des
9 comportements associés à une transition avec événement, ou une transition de terminaison. La
10 garde est vérifiée, et le comportement de sortie de l'état source (ou les états) est exécuté (ou rien
11 dans le cas d'absence de comportement), le comportement associé à la transition est exécuté, et
12 enfin le comportement d'entrée de l'état (ou les comportements d'entrée des états) destination
13 est exécuté (ou rien dans le cas d'absence de comportement). Cela concerne également la manière
14 de gérer les transitions et leurs exécutions dans le cas d'un conflit ou dans le cas de plusieurs
15 transitions avec événements. Dans la spécification d'UML il y a une définition de la sélection
16 des transitions : à chaque fois qu'un événement est distribué, zéro, une ou plusieurs transitions
17 peuvent être exécutées. Les événements de terminaison (*completion event*) sont prioritaires dans
18 la distribution, c'est-à-dire qu'ils sont distribués devant toutes les occurrences d'événements en
19 attente dans la file d'attente des événements. Cependant l'ordre entre les différentes transitions
20 de terminaison n'est pas spécifié. Pour un événement donné, les transitions actives sont celles
21 déclenchées par cet événement, tel que toutes les états sources sont actifs, et les gardes asso-
22 ciées à ces transitions sont vérifiées. Rappelons qu'un état est actif lorsqu'il se trouve dans une
23 configuration d'états. Une configuration d'états représente la situation dans laquelle se trouve la
24 machine à états à un moment donné ("Consequently, a particular "state" of an executing State-
25 Machine instance is represented by one or more hierarchies of States [...] This complex hierarchy
26 of States is referred to as a state configuration" [?, Section State configurations, p.306]). Ensuite,
27 une transition t peut être franchie à condition qu'il n'y ait pas de transitions activées dans les
28 sous-états de l'état source (ou les états) de t . Cela implique : premièrement, que plusieurs transi-
29 tions peuvent être franchies en même temps durant le même *run-to-completion step*, du moment
30 qu'elles sont dans des régions différentes; Deuxièmement, qu'il y a un non-déterminisme, par
31 exemple, dans le cas d'un état simple avec deux transitions sortantes activées avec le même évé-
32 nement, et ayant des gardes vérifiées (dans ce cas la une seule transition peut être franchie). À
33 noter également, si un événement est distribué et qu'il n'y a aucune transition active avec cet
34 événement alors l'événement est simplement jeté.

Réseaux de Petri colorés (CPN)

Les réseaux de Petri colorés (CPNs) [?] sont un type d'automates représentés par un graphe bipartite avec deux types de nœuds, d'une part les places (dessinées sous forme de cercles ou d'ovales avec le nom à l'intérieur, par exemple, p_1 dans Figure 2.2a), d'autre part les transitions (dessinées sous forme de rectangles avec le nom à l'intérieur, par exemple, t dans Figure 2.2a). Il est également possible d'avoir des inscriptions textuelles à côté des places, des arcs et des transitions. Dans l'outil CPN Tools ([?]) ces inscriptions sont écrites dans le langage de programmation *CPN ML* [?] qui est une extension du langage *standard ML* [?]. Les places et les transitions représentent les nœuds et, avec les arcs, ils forment la structure du réseau. Les places peuvent être connectées uniquement avec les transitions en utilisant les arcs et vice versa (la connexion entre deux nœuds de même type n'est pas autorisée).

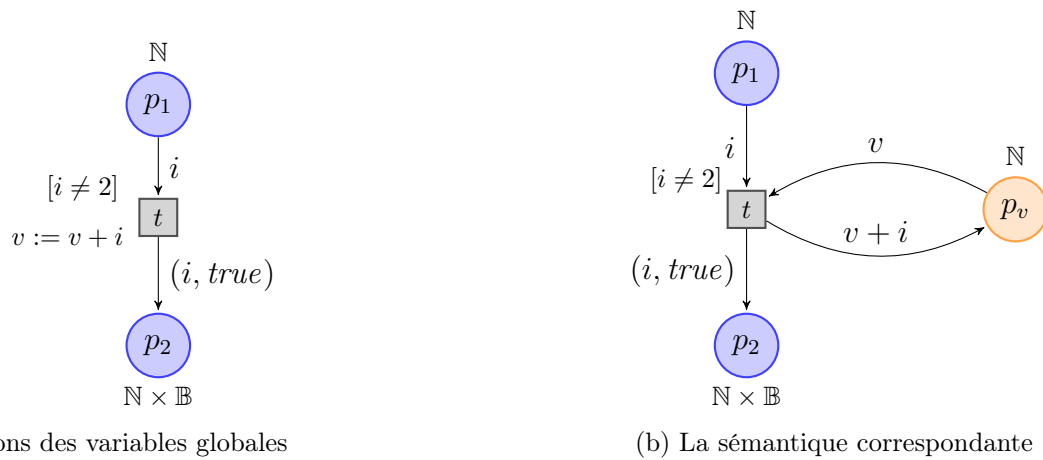


FIGURE 2.2 – Notation des variables globales et la sémantique correspondante

Définitions et notations

\mathbb{N} dénote l'ensemble des entiers naturels et \mathbb{B} , l'ensemble des booléens.

Les places peuvent être marquées avec des jetons et une valeur est attachée à chaque jeton. Le type de la valeur attachée au jeton doit être le même que celui associé à la place qui contient ce jeton (par exemple, le type $\mathbb{N} \times \mathbb{B}$ dans la Figure 2.2a). Un type spécial, nul (noté \bullet), est utilisé pour les jetons qui ne portent pas de valeur.

Les arcs sont étiquetés par des expressions où certaines variables correspondant aux type de la place peuvent apparaître (par exemple, $v + i$ dans Figure 2.2b). Les expressions sur les arcs doivent être correctement typées en fonction du type des variables, des constantes, des opérateurs et des fonctions. Dans le cas où les variables dans les expressions sont instanciées par des valeurs (du bon type) l'expression peut être évaluée.

Les transitions peuvent avoir une *garde* (par exemple, $[i \neq 2]$), qui sera la condition pour que la transition soit franchissable (s'il n'y a pas de garde alors cela veut dire que sa valeur est à vrai). Il est possible également d'avoir dans les transitions un compartiment pour les *segments de code*. Ce compartiment permet d'insérer du code écrit dans le langage *CPN ML* afin de modifier des variables globales, d'utiliser des fonctions (pré-définies ou définies par l'utilisateur) ou de modifier la valeur des jetons.

1 Le nombre de jetons et leur couleur dans chaque place représentent l'état du système et c'est
 2 le marquage du modèle CPN. La Figure 2.3 montre un autre exemple de réseau de Petri coloré
 3 avec divers éléments syntaxiques tels que : des variables (x , j , $k2$, v), des types (INT , $STRING$),
 4 des gardes sur les transitions, et des segments de code, etc.

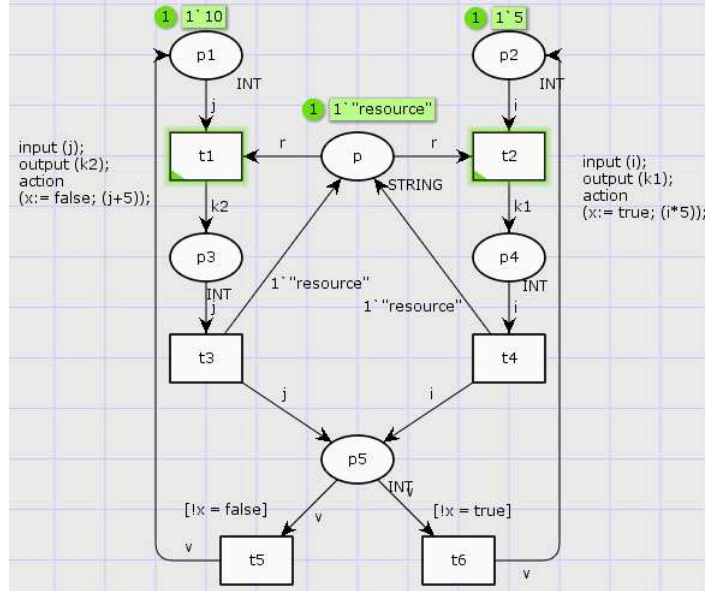


FIGURE 2.3 – Exemple de réseau de Petri coloré modélisé en utilisant CPN Tools

5 Nous définissons dans ce qui suit la représentation formelle des réseaux de Petri colorés avec
 6 les variables globales.

7 **Definition 1** (réseaux de Petri colorés). *Un réseau de Petri coloré étendu avec des variables*
 8 *globales (CPN) est un tuple*

$$9 \quad CPN = (P, T, A, B, V, C, G, E, MI, VI), \text{ où}$$

- 10 1. P un ensemble fini de places,
- 11 2. T un ensemble fini de transitions tel que $P \cap T = \emptyset$,
- 12 3. $A \subseteq P \times T \cup T \times P$ est un ensemble dirigé d'arcs,
- 13 4. B est un ensemble fini non vide de types (et un type est un ensemble de couleurs),
- 14 5. V est un ensemble fini de variables typées tel que $\forall v \in V, Type(v) \in B$,
- 15 6. $C : P \rightarrow B$ est une fonction de couleur assignant pour chaque place un ensemble de
- 16 couleurs,
- 17 7. $G : T \rightarrow Expr(V)$ est une fonction de gardes qui permet d'assigner à chaque transition
- 18 une garde telle que $Type(G(t)) = \mathbb{B}$, et $Var[G(t)] \subseteq V$,
- 19 8. $E : A \rightarrow Expr(V)$ est une fonction d'expression d'arcs qui permet d'assigner une expression
- 20 d'arcs pour chaque arc tel que $Type(E(a)) = C(p)_{MS}$, où p est la place connectée à l'arc a ,
- 21 et MS dénoté le multi-ensemble,
- 22 9. $MI : P \rightarrow Expr(V)$ est la fonction du marquage initial qui permet d'assigner le marquage
- 23 initial pour chaque place tel que $Type(MI(p)) = C(p)_{MS}$, et

10. VI est la fonction d'initialisation de variables qui permet d'assigner pour chaque variables la valeur initiale telle que, pour tout $v \in V$, $Type(VI(v)) = Type(v)$.

Variables globales et segment de code Nous utilisons le concept de *variables globales*, une notation qui n'ajoute pas d'expressivité aux CPNs, mais qui permet de les rendre plus compacts. Les variables globales peuvent être lues dans les gardes et mises à jour dans le segment de code de la transition. Afin de représenter ces variables globales dans CPN Tools nous utilisons le concept de variables de références. Ces variables sont comme des pointeurs en langage C et elles peuvent être utilisées dans les gardes ou les segments de code. Un exemple est donné dans [Figure 2.3](#) : x est une variable globale (une variable de référence dans CPN Tools) de type \mathbb{B} . La variable est mise à jour dans le segment de code associé avec la transition $t1$ ou la transition $t2$, avec l'expression `false` ou `true` (notons pour accéder à la valeur de la variable globale il suffit d'utiliser la notation suivante : `!x`).

Ce genre de mise à jour au niveau du segment de code est supporté par CPN Tools mais avec quelques limitations comme par exemple la possibilité d'utiliser les variables globales dans le segment de code mais la non prise en compte de ces variables lors de la vérification. Sinon, il est possible de simuler les variables globales en utilisant des places "globales" avec un seul jeton (du type de la variable) qui encode la valeur courante de la variable. Cette construction est équivalente à celle dans [Figure 2.2b](#). Quand une variable globale est lue dans une garde, le jeton avec la valeur $v + i$ est de retour dans la place p_v .

Définition formelle

Afin de mieux comprendre les éléments syntaxiques présentés ci-dessus, nous présentons dans cette sous-section une description de la sémantique des réseaux Petri colorés.

Multi-ensemble

Soit \mathbb{Y} un ensemble fini et non vide. Un multi-ensemble a sur \mathbb{Y} est une fonction de \mathbb{Y} vers \mathbb{N} . Notons :

$$a = \sum_{x \in \mathbb{Y}} a(x) x$$

où $a(x)$ désigne le nombre d'occurrences de x dans a .

$Bag(\mathbb{Y})$ désigne l'ensemble des multi-ensembles sur \mathbb{Y} .

Structure

Un réseau de Petri coloré est un 6-uplet $\langle P, T, C, Pre, Post, M_0 \rangle$ où :

- P est l'ensemble des places, T est l'ensemble des transitions ($P \cap T = \emptyset, P \cup T \neq \emptyset$)
- C définit pour chaque place et chaque transition son domaine de couleur
- Pre indexée sur $P \times T$, est la matrice d'incidence arrière du réseau
- $Post$ indexée sur $P \times T$, est la matrice d'incidence avant du réseau
- $Pre(p, t)$ et $Post(p, t)$ sont des fonctions linéaires de couleurs définies de $Bag(C(t))$ dans $Bag(C(p))$.
- M_0 est le marquage initial du réseau, c'est un vecteur indexé par P et $M_0(p)$ est un élément de $Bag(C(p))$

1 Marquage et marquage initial d'un CPN

2 Pour chaque réseau de Petri coloré, nous avons la notion de marquage et de marquage initial.
 3 Un marquage est la distribution des jetons avec leurs valeurs dans les places du réseaux (dans
 4 le cas où la place ne comporte pas de jetons alors ça sera l'ensemble vide). Un marquage M
 5 d'un réseau de Petri coloré est un vecteur indexé par P , avec $M(p) \in Bag(C(p))$. Nous notons
 6 (Mark P1, Mark P2, ..., Mark P3) le marquage M d'un réseau de Petri coloré, tels que :

- 7 • $Mark = (nb_1'value_1 + +nb_2'value_2 + +... nb_k'value_k)$.
- 8 • Chaque place peut contenir un ensemble de couleurs possibles de jetons. Pour un ensemble
 9 de jetons d'une couleur donnée on a la notation suivante : $(nb_i'value_i)$, où nb_i est le
 10 nombre de jetons ayant la valeur $value_i$. En d'autres termes, une place peut contenir un
 11 nombre nb_1 de jetons ayant la valeur $value_1$, un nombre nb_2 de jetons ayant la valeur
 12 $value_2$, etc.

13 Le marquage initial est un marquage que le réseau de Petri coloré a initialement, c'est-à-dire
 14 la distribution initiale des jetons avec leurs valeurs dans le réseau. Dans la [Figure 2.3](#) le marquage
 15 initial du réseau de Petri coloré présent dans l'exemple est $(1'10, 1'5, 1''resource'', \emptyset, \emptyset, \emptyset)$. Ce
 16 marquage signifie qu'initialement la place $p1$ comporte un jeton de valeur 10, la place $p2$ comporte
 17 un jeton de valeur 5, la place p comporte un jeton de valeur *resource* et le reste des places (c'est-
 18 à-dire $p3, p4, p5$) comportent 0 jetons.

19 Franchissement des transitions et consommation/production des jetons

20 Une transition t est dite franchissable pour une instance $c_t \in C(t)$ et un marquage M dans le
 21 cas où :

- 22 • Soit t est non gardée, soit la garde vaut *true* pour c_t
- 23 • $\forall p \in P, M(p) \geq Pre(p, t)(c_t)$

24 Autrement dit, t est franchissable si (i) toutes les places d'entrée de la transition comportent
 25 le nombre de jetons nécessaire. Dans le cas où dans l'arc (allant d'une place à une transition)
 26 comporte une inscription avec un nombre n de jetons alors la place doit contenir ce nombre de
 27 jetons pour que la transition soit franchissable. (ii) La garde de la transition doit être vérifiée.

28 Le franchissement d'une transition permet de produire ou de consommer des jetons, cepen-
 29 dant cela n'indique pas combien de jetons il faut produire ou consommer. Nous présentons dans
 30 la [Section 2.3.1](#) que les arcs entre les places et transitions peuvent avoir des inscriptions. Ces
 31 inscriptions permettent de définir le nombre de jetons à produire (dans le cas où la source de
 32 l'arc c'est une transition) et le nombre de jetons à consommer (dans le cas où la source de l'arc
 33 est une place). Par exemple, dans la [Figure 2.3](#) le nombre de jetons à consommer en franchissant
 34 la transition $t1$ est j .

35 Exemple d'évolution du marquage d'un CPN

36 Le réseau de Petri coloré peut évoluer et son évolution correspond à l'évolution de son mar-
 37 quage dans le temps. Cela est représenté par la consommation et production des jetons dans les
 38 différentes places du réseau. Le passage d'un marquage à un autre (ou la consommation/produc-
 39 tion de jetons) se fait grâce au franchissement des transitions. En faisant la simulation (avec des
 40 outils dédiés), cela nous permet de voir l'évolution du réseau de Petri d'un marquage à un autre.
 41 Nous présentons dans la [Figure 2.3](#) un exemple de réseau de Petri coloré avec son marquage initial
 42 qui est $(1'10, 1'5, 1''resource'', \emptyset, \emptyset, \emptyset)$ correspondant à la suite de places ($p1, p2, p, p3, p4, p5$).

43 Afin de faire évoluer le marquage du réseau, la transition $t1$ sera franchie ; le résultat du fran-
 44 chissement de cette transition est montré dans la [Figure 2.4](#). Nous remarquons que le marquage

- 1 est passé de $(1'10, 1'5, 1''resource'', \emptyset, \emptyset, \emptyset)$ à $(\emptyset, 1'5, \emptyset, 1'15, \emptyset, \emptyset)$ après le franchissement de la transition t1.

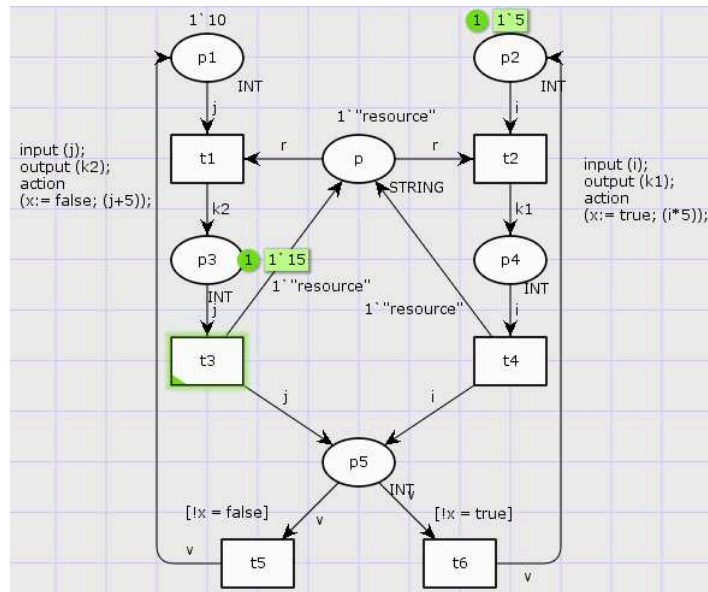


FIGURE 2.4 – Évolution du marquage : Franchissement de la transition t1

2

3

4

- Le marquage évolue encore une fois en franchissant la transition t3, le résultat est présenté dans la Figure 2.5. Le marquage du réseau est passé de $(\emptyset, 1'5, \emptyset, 1'15, \emptyset, \emptyset)$ à $(\emptyset, 1'5, 1''resource'', \emptyset, \emptyset, 15)$ après le franchissement de la transition t3.

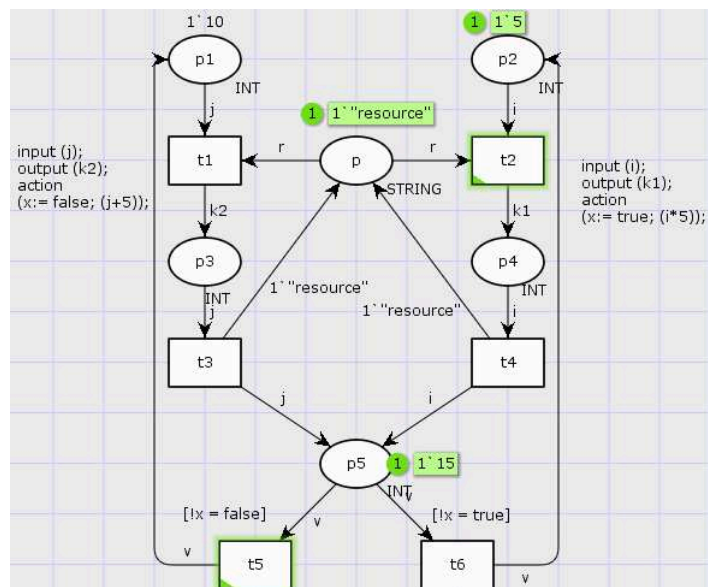


FIGURE 2.5 – Évolution du marquage : Franchissement de la transition t3

5

6 CPNTools

7

8

- CPNTools est un outil de modélisation, de simulation et d'analyse des réseaux de Petri. Il supporte les réseaux de Petri places/transitions, les réseaux de Petri colorés et les réseaux de

1 Petri colorés temporisés. En plus de l'interface de modélisation, CPN Tools possède un simulateur
2 et un générateur de l'espace d'états.

3 CPN Tools est composé de deux parties principales. La première concerne la modélisation
4 graphique et l'interface qui permet de dessiner les réseaux de Petri colorés. La seconde partie
5 concerne la partie analyse, simulation et vérification qui permet d'effectuer différentes opérations
6 sur les réseaux de Petri colorés. Le langage utilisé pour les différentes inscriptions et pour l'écriture
7 des propriétés est le langage *CPN ML* (une extension du standard ML). La Figure 2.6 montre
8 l'architecture de CPN Tools avec ses différents composants.

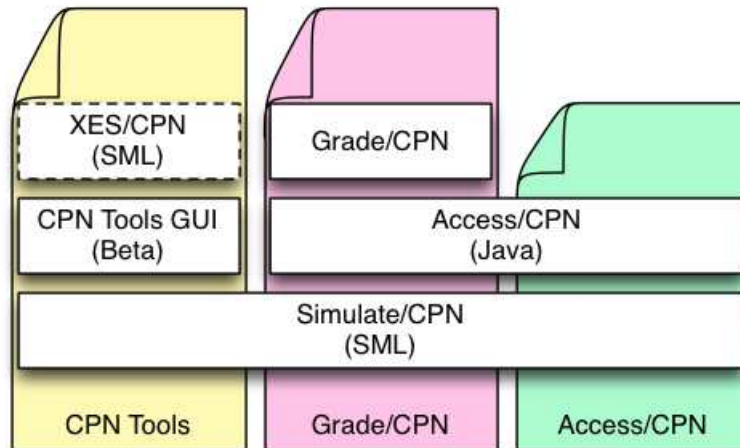


FIGURE 2.6 – L'architecture de CPN Tools

9 La Figure 2.7 montre l'interface standard de CPN tools. Cette dernière est composée de deux
10 parties, la partie à gauche de la fenêtre représente les différents composants qu'on peut insérer
11 ainsi que la manipulation des programmes (déclarations, code, etc). La partie qui est à droite de
12 la fenêtre est celle qui concerne la modélisation graphique et qui permet de dessiner les réseaux
13 de Petri.

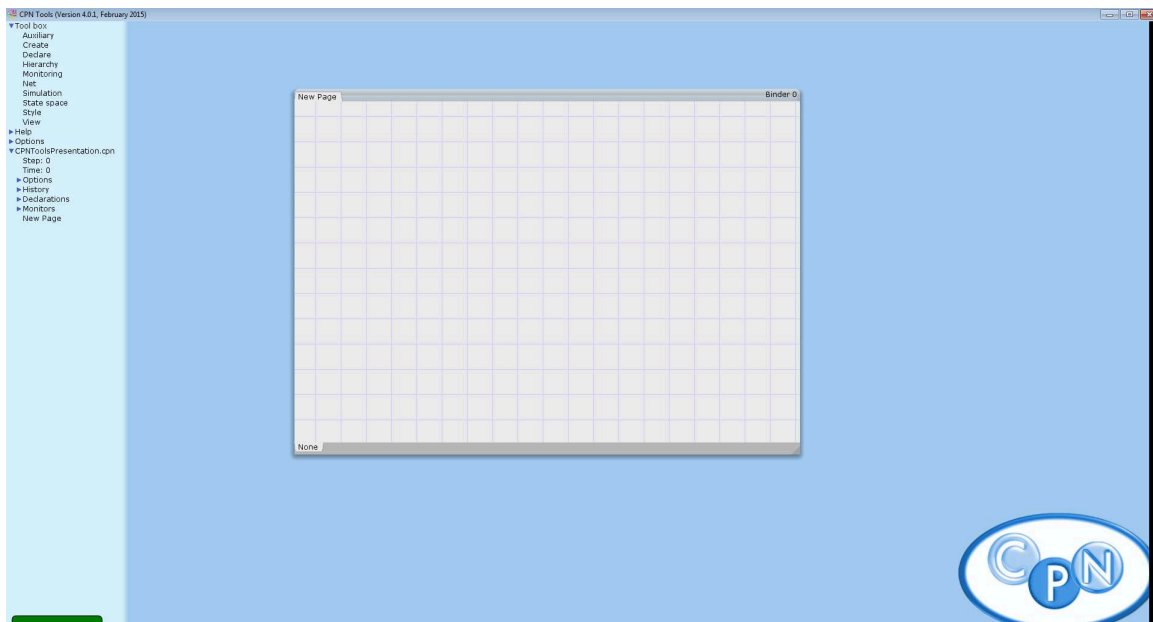


FIGURE 2.7 – Interface Standard de CPN tools

1 La Figure 2.8 montre un exemple de réseau de Petri coloré avec le simulateur ainsi que
 2 l'analyseur de l'espace d'état. On remarque que le premier composant représenté par *Binder*
 3 *0* représente la modélisation graphique d'un exemple des réseaux de Petri colorés. Le second
 4 composant représenté par *Sim* représente le simulateur qui permet de lancer le déroulement du
 5 réseau de Petri (pas à pas ou continue), l'arrêt de l'exécution ainsi que les retours en arrière.
 6 Le troisième composant représenté par *SS* représente l'analyseur de l'espace d'état, il permet de
 7 générer l'espace d'état, la génération d'un rapport comportant les différents détails ainsi que de
 8 différentes manipulations sur l'espace d'état.

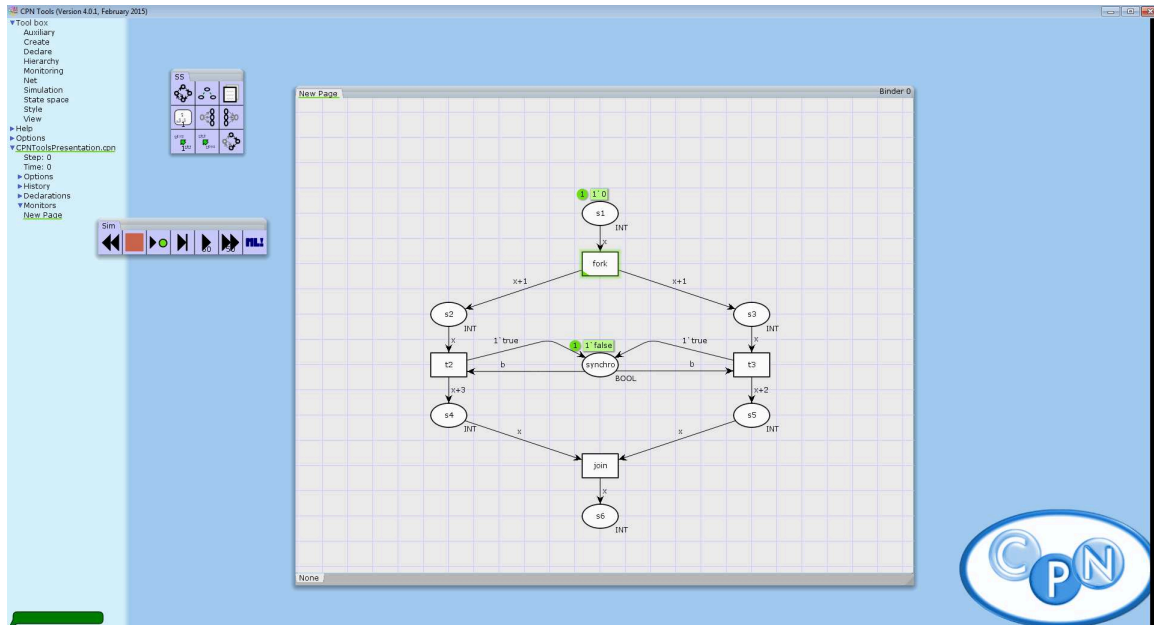


FIGURE 2.8 – Exemple dans CPN tools

9 Conclusion

10 Nous avons présenté dans ce chapitre les diagrammes états-transitions ainsi que les réseaux de
 11 Petri colorés. Ces concepts sont utilisés dans notre traduction des diagrammes états-transitions
 12 vers les réseaux de Petri colorés (voir le Chapitre 5), dans notre méthode de spécification avec les
 13 diagrammes états-transitions (voir le Chapitre 4) et dans la définition des contraintes temporelles
 14 (voir le Chapitre 8). Nous présentons dans ce qui suit les différents travaux de l'état de l'art que
 15 nous utilisons pour comparer nos travaux (voir le Chapitre 3).

Chapitre 3

État de l'art

Contents

3.1	Introduction	31
3.2	Formalisation d'UML	31
3.3	UML et temps	35
3.4	Méthodes de spécification	59
3.5	Conclusion	63

Introduction

Nous présentons dans ce chapitre les différents travaux qui existent dans l'état de l'art. La [Section 3.2](#) est consacrée à l'aspect traduction des diagrammes états-transitions vers d'autres formalismes (tels que les réseaux de Petri, les automates, etc). La [Section 3.3](#) est consacrée à l'aspect temporel dans les diagrammes états-transitions et le traitement des systèmes temps-réel. Enfin la [Section 3.4](#) est consacrée à l'aspect modélisation et spécification des systèmes en utilisant les diagrammes états-transitions et UML en général.

Formalisation d'UML

Il existe plusieurs travaux visant à donner une sémantique formelle aux diagrammes états-transitions (SMD). Cependant, la plupart de ces approches considèrent un ensemble restreint des éléments syntaxiques proposés par l'OMG [?]. Ces approches peuvent être classifiées en deux catégories : les approches définissant une sémantique opérationnelle, et les approches qui traduisent les diagrammes états-transitions vers un langage équipé d'une sémantique formelle.

Les réseaux de petri et (une variante de) CSP sont les formalismes destinations les plus utilisés dans l'état de l'art de la traduction des diagrammes états-transitions, en plus les traductions vers les réseaux de Petri et CSP sont, en général, des travaux complets en terme de prise en compte de la syntaxe des diagrammes états-transitions. CSP est un langage hiérarchique qui capture la hiérarchie des diagrammes états-transitions, tandis que les réseaux de Petri ont un système de jetons qui capture l'état actif des diagrammes états-transitions. Nous discutons également des approches traduisant les diagrammes états-transitions vers d'autres formalismes, tels

1 que les diagrammes états-transitions abstraits ou Promela (le langage d'entrée de SPIN modèle
2 checking).

3 **Sémantique opérationnelle** Plusieurs approches sont proposées afin de fournir une sé-
4 mantique opérationnelle pour les diagrammes états-transitions UML. Dans [?], une sémantique
5 opérationnelle est proposée en prenant en compte un ensemble des éléments syntaxiques d'UML.
6 L'approche utilise des terminaux pour représenter les états, et les transitions sont emboîtées dans
7 des Or-termes, cela rend l'extension avec d'autres éléments difficile (surtout dans le cas d'une
8 transition composée en présence des pseudo-états fork et join).

9 Dans [?], une comparaison entre les différentes interprétations de la sémantique des state-
10 charts dans un le cadre d'une sémantique opérationnelle structurée est présentée. Cependant,
11 au-delà du fait que les statecharts sont différents des diagrammes états-transitions UML, seule-
12 ment certains éléments syntaxiques sont pris en compte par la sémantique.

13 Dans [?] un travail sur la sémantique et le raffinement des objets mobiles est présenté. Les au-
14 teurs modélisent ces objets en utilisant les diagrammes états-transitions UML. Ces diagrammes
15 seront ensuite formalisés en utilisant MTLA (une extension de la logique temporelles des actions
16 de Lamport). Enfin, une étude sur le raffinement de ces diagrammes (dont la sémantique est
17 justifiée dans MTLA) est effectuée. Ce travail ne prend pas en compte les éléments syntaxiques
18 suivants : la hiérarchie, les pseudo-états (à l'exception des pseudo-états initiaux), les événements
19 différés. Il n'est pas précisé si d'autres aspects tels que les comportements des états sont consi-
20 dérés. Les auteurs précisent également que la gestion des priorités entre transitions est différente
21 de celle proposée par la spécification d'UML.

22 Dans [?], le non déterminisme est considéré dans les états composites orthogonaux, seulement
23 une partie des éléments syntaxiques est pris en compte, et ni la gestion des événements ni le run-
24 to-completion (un aspect important dans les diagrammes états-transitions) ne sont discutés.

25 Dans [?], une autre sémantique formelle est définie pour un ensemble des éléments syntaxiques
26 des diagrammes états-transitions. Cependant, la procédure de transformation informelle, ainsi
27 que les coûts supplémentaires que la transformation introduit sont tels que ce n'est pas adapté
28 pour le développement d'outil et pour la vérification formelle.

29 Nous pensons que le travail le plus complet en terme d'éléments syntaxiques considérés est
30 la sémantique opérationnelle proposée dans [?], où la syntaxe complète des diagrammes états-
31 transitions est considérée avec l'exception de la notion temporelle.

32 Les inconvénients de fournir une sémantique opérationnelle sont dus à deux points. La premier
33 point est que ces approches (basées sur une sémantique opérationnelle) nécessitent le développe-
34 ment d'un outil *ad-hoc* ce qui implique, dans le cas général, énormément de temps. En revanche,
35 les approches basées sur la traduction ont le bénéfice des outils existants dans l'état de l'art
36 (l'unique outil à implémenter est l'outil qui permet de faire la traduction, ce qui, en général, per-
37 met de consommer moins de temps comparant à l'implémentation d'un outil de modèle checking).
38 L'état de l'art du modèle checking souvent ont le bénéfice de plusieurs année de développement
39 et de maturation, et intègre plusieurs optimisation afin d'effectuer une vérification efficace. Le
40 second point est que les approches basées sur une sémantique opérationnelle sont plus difficile
41 à mettre à jour : dans le cas d'un nouvel élément syntaxique introduit par l'OMG, ou dans le
42 cas du changement de la syntaxe et/ou la sémantique (semi-formelle), alors non seulement la
43 sémantique opérationnelle doit être changée mais aussi l'outil de vérification associé.

44 **Traduction à CSP** Ng et Butler [?, ?] proposent une traduction des diagrammes états-
45 transitions vers CSP. Cependant, plusieurs éléments syntaxiques des diagrammes états-transitions,
46 tels que le mécanisme des priorités des transitions, ne sont pas considérés.

1 Zhang et Liu [?] proposent une traduction des diagrammes états-transitions vers CSP \sharp , une
2 extension du langage CSP, qui sert comme entrée dans le langage de modélisation PAT [?]. La
3 traduction prend en compte un ensemble d'éléments syntaxique tels que : structures de don-
4 nées, join/fork, pseudo-état histoires, comportements d'entrée et de sortie (sans utilisation de
5 variables) ainsi que la vérification des propriétés de sûreté et de vivacité. Cependant, la sélection
6 des transitions (ainsi que les priorités de franchissement) semble ne pas être considéré dans la
7 traduction.

8 Jacobs et Simpson [?] présentent une traduction des diagrammes SysML d'activités et états-
9 transitions vers CSP, en prenant en compte un ensemble limité d'éléments syntaxiques des dia-
10 grammes états-transitions.

11 Notez que CSP ne permet pas une représentation graphique en comparaison aux réseaux de
12 Petri.

13 **Traduction vers réseaux de Petri** Un formalisme destination pour la traduction des dia-
14 grammes états-transitions est le réseau de Petri (et ses extensions) pour sa similitude graphique
15 (et sémantique) avec les SMDs, comme mentionné dans la spécification de l'OMG : 1) les dia-
16 grammes d'activités sont définis comme des graphes ressemblant aux réseaux de Petri "Activities
17 [are] defined using Petri-net-like graphs" [?, p.298], and 2) les pseudo-états join sont similaire à
18 ceux des réseaux de Petri "similar to junction points in Petri nets" [?, p.327].

19 Plusieurs approches utilisent les réseaux de Petri colorés (CPNs) pour modéliser et analyser
20 les systèmes Dans [?] l'approche utilise les CPNs pour modéliser et valider les comportements
21 des objets concurrent modélisés avec UML. Les auteurs présentent comment passer des objets
22 actif/négatif ainsi que les messages de communications vers des CPNS. Bien que l'approche ne
23 traite pas en particulier les diagrammes états-transitions, elle montre une vue générale pour la
24 transformation des diagrammes UML en CPNs. Dans [?], les objets dépendant des états et leurs
25 statecharts sont mappés vers les CPNs. Chaque objet contient un statechart encapsulé qui sera
26 utilisé pour recevoir un événement. Dans cette approche, il n'est pas clair quel ensemble d'élé-
27 ments syntaxiques est considéré pendant la transformation vers les CPNs. De la même manière,
28 le travail dans [?] consiste à modéliser et à analyser les software concurrent orientés-objets en
29 utilisant des patterns de comportements et les CPNs. Cette approche utilise les diagrammes de
30 collaboration pour modéliser un ensemble d'objets (pour chaque objet est associé un diagramme
31 états-transitions UML). Chaque objet a un stéréotype afin d'indiquer quel pattern utilisé (sachant
32 qu'il y a un ensemble de patterns prédéfinis). Dans notre travail, nous nous intéressons à définir
33 une sémantique (en utilisant un algorithme de traduction) pour les diagrammes états-transitions.

34 Trowitzsch and Zimmermann [?] proposent une traduction d'un ensemble des éléments syn-
35 taxiques des diagrammes états-transitions avec du temps vers les réseaux de Petri stochastiques.
36 L'approche ne prend pas en compte un ensemble d'éléments syntaxiques UML, cependant les
37 événements avec du temps sont présentés.

38 Shartz *et al.* [?] présentent un outil qui permet d'effectuer un ensemble d'opérations pour ana-
39 lyser les diagrammes statecharts à plusieurs niveau de complexité. Les opérations d'analyses sont
40 basées sur l'analyse du modèle des réseaux de Petri résultat de la transformation des statecharts.

41 Hillah and Thierry-Mieg [?] présentent un outil qui permet de traduire des modèles UML
42 (tels que les diagrammes d'activités, etc.) vers une instance des réseaux de Petri. La traduction
43 est implémentée en utilisant l'outil BBC (behavioral consistency checker)

44 Plus récemment, une traduction des SMDs vers les réseaux de Petri a été proposée dans [?], où
45 les éléments syntaxiques considérés : la synchronisation, un aspect limité de la hiérarchie, pseudo-
46 états fork et join (sans transitions inter niveau); cependant sans considération des pseudo-états
47 histoire et les variables.

48 Dans [?], les auteurs proposent une approche différente de cette de [?], en prenant en compte

1 un ensemble large des éléments syntaxiques des diagrammes états-transitions tels que : la hié-
2 rarchie, les transitions locales et externes, les comportements d'entrée et de sortie et do etc.
3 Cependant, une limitation de cette approche est l'absence de la concurrence (et par conséquent
4 les pseudo-états fork et join ainsi que les états composites orthogonaux).

5 Dans [?] nous proposons une extension de [?] en introduisant la concurrence et en prenant
6 en compte les différents éléments syntaxiques considérés dans [?].

7 Enfin, Luciano *et al.* [?] propose une autre approche pour formaliser les diagrammes états-
8 transitions avec des réseaux de Petri haut-niveau, c'est-à-dire des réseaux de Petri où les places
9 peuvent être raffinées pour modéliser les états composites. Notez que les réseaux de Petri de haut-
10 niveau ne sont pas des réseaux de Petri colorés, cependant les deux formalismes donnent une
11 abstraction de haut niveau et engendrent des modèles compacts. Dans ce travail, les diagrammes
12 de classes, les diagrammes états-transitions et les diagrammes d'interactions sont considérés, ainsi
13 que une proposition de règles personnalisées pour chaque diagramme. Cependant, les auteurs ne
14 présentent pas les détails à propos de ces règles, en revanche ils illustrent les différentes étapes en
15 utilisant le problème des philosophes. L'analyse et la validation sont aussi discutées, en particulier
16 comment représenter les propriétés UML (tels que l'absence d'interblocage etc.) ainsi que la façon
17 avec laquelle il faut les transformer en réseau de Petri.

18 **Traduction vers les automates** D'autres approches traduisent la spécification UML vers
19 des modèles intermédiaires de certains modèles checker, par exemple, SPIN [?]. Le premier travail
20 avec SPIN comme destination est [?] qui présente un schéma de traduction pour les SMDs vers le
21 modèle Promela (langage d'entrée pour SPIN), ensuite utilise SPIN pour la phase de vérification.
22 Certains éléments syntaxiques importants ne sont pas considérés dans la traduction, tels que :
23 fork/join, pseudo-état histoire, comportements d'entrée et de sortie des états, les variables, et les
24 diagrammes états-transitions multiples.

25 Dans [?] une extension de [?] est proposée afin d'inclure les diagrammes états-transitions
26 multiples communiquant de façon asynchrone. L'approche considère uniquement un ensemble
27 d'éléments syntaxiques des diagrammes états-transitions et ne prend pas en compte les pseudo-
28 états (à l'exception du pseudo-états initial) ainsi que les actions.

29 Une traduction des SMDs vers SPIN est aussi considérée dans [?] avec des cas hiérarchiques et
30 non-hiérarchiques ; cependant les éléments syntaxiques suivants ne sont pas considérés : pseudo-
31 état histoire, fork et join, comportements d'entrée et de sortie. Dans [?], un prototype d'outil
32 est conçu pour lier SPIN avec RSARTE (IBM Rational Software Architect RealTime Edition),
33 un outil de modélisation des diagrammes UML. Ce travail prends en compte tout type de dia-
34 grammes UML avec du temps réel. Étant une partie d'UML, les diagrammes états-transitions
35 sont également traduis vers Promela. Cependant, leur objectif est de montrer les communica-
36 tions entre différents objets, et donc il n'y a aucun détail à propos des éléments syntaxiques des
37 diagrammes états-transitions pris en compte ou la vérification des diagrammes états-transitions.
38 Tandis que, nous ne traitons pas le temps réel, mais nous visons à considérer le maximum d'élé-
39 ments syntaxiques des diagrammes états-transitions UML dans notre traduction

40 **Autres approches de traductions** Une approche proposée dans [?] qui permet de repré-
41 senter les statecharts en utilisant des structures de transitions. Cette approche prend en compte
42 une version simplifiée des statecharts et ne considère pas les variables.

43 Dans [?] les auteurs proposent une transformation des diagrammes statecharts vers les graphes
44 avec une sémantique étendue Cependant, le travail ne précise pas quels sont les éléments syn-
45 taxiques considérés dans la transformation tels que : comportements d'entrée ou de sortie, les
46 transitions internes, pseudo-états (fork/join, junction, choice).

1 Un autre travail dans [?] qui propose une sémantique basée sur les processus d'algèbres pour
 2 les statecharts. La sémantique est accompagnée du langage SPL (statecharts process language).
 3 Cependant, l'approche ne prend en compte qu'un ensemble réduit des éléments syntaxiques des
 4 statecharts (les transitions de franchissement de frontière ne sont pas considérés)

5 Dans [?] une approche qui propose une formalisation des diagrammes états-transitions en
 6 utilisant les machines à états abstraites (ASMs). Les ASMs permettent la formalisation des
 7 conditions pour valider et vérifier les systèmes. La syntaxe du modèle couvre plusieurs éléments
 8 syntaxiques des diagrammes états-transitions, tels que : les événements différés, les événements
 9 de terminaisons et les comportements internes. Cependant, les pseudo-états tels que : fork, join,
 10 junction, choice, de terminaisons ne sont pas considérés.

11 Le travail dans [?] présente une formalisation logique de premier ordre pour *fUML* Foundatio-
 12 nal Subset of Executable UML), afin d'appliquer les techniques de model checking et ainsi vérifier
 13 ses modèles. *fUML* [?] est un standard OMG qui définit la sémantique d'exécution d'un sous-
 14 ensemble d'UML en utilisant une machine virtuelle. Cette machine est sous forme de pseudo-code
 15 *Java* pour exécuter les modèles *fUML* (des modèles UML mais utilisant que le sous-ensemble
 16 considéré par *fUML*). Elle comporte trois parties :

- 17 • Une syntaxe abstraite représentée par un sous-ensemble d'UML et composée principale-
 18 ment de diagrammes de classes et diagrammes d'activité.
- 19 • Le modèle d'exécution qui définit la sémantique d'exécution de la syntaxe abstraite
- 20 • Une bibliothèque de modèles qui définit les types primitifs et les comportements (par
 21 exemple, les entiers et l'addition entre deux entiers).

22 La formalisation proposée dans ce travail prend en compte les flux de contrôle, les flux de
 23 données, les ressources et des dimensions temporisées. Un travail d'implémentation basé sur le
 24 langage Alloy est proposé.

25 UML et temps

26 UML profile

27 La spécification de l'OMG concernant les diagrammes états-transitions et UML en général
 28 comporte une partie sur l'expression des contraintes temporelles. L'OMG définit plusieurs types
 29 d'événements y compris les *Time events* et qui représentent l'instant dans le temps dans lequel
 30 l'événement apparaît. Cet instant est spécifié en utilisant une expression de temps (*TimeExpres-*
 31 *sion*). Dans le cas où le temps concerne l'instant où l'événement apparaît alors le temps considéré
 32 est absolu et donc par conséquent l'évaluation de l'expression du temps est en temps absolu. Dans
 33 le cas où le *TimeEvent* est utilisé comme un déclencheur et que le temps d'occurrence est relatif
 34 à une date de début pour le déclencheur alors le temps est considéré comme relatif et l'évaluation
 35 de l'expression du temps est en temps relatif.

36 La [Figure 3.1](#) représente le méta-modèle utilisé pour la représentation du temps et des durées
 37 dans UML.

38 La syntaxe et la sémantique utilisées pour la représentation du temps dans UML sont pré-
 39 sentées sous forme de grammaire comme le montre la [Figure 3.2](#). Les deux éléments syntaxiques
 40 utilisés pour l'expression du temps dans UML sont **after** et **at**.

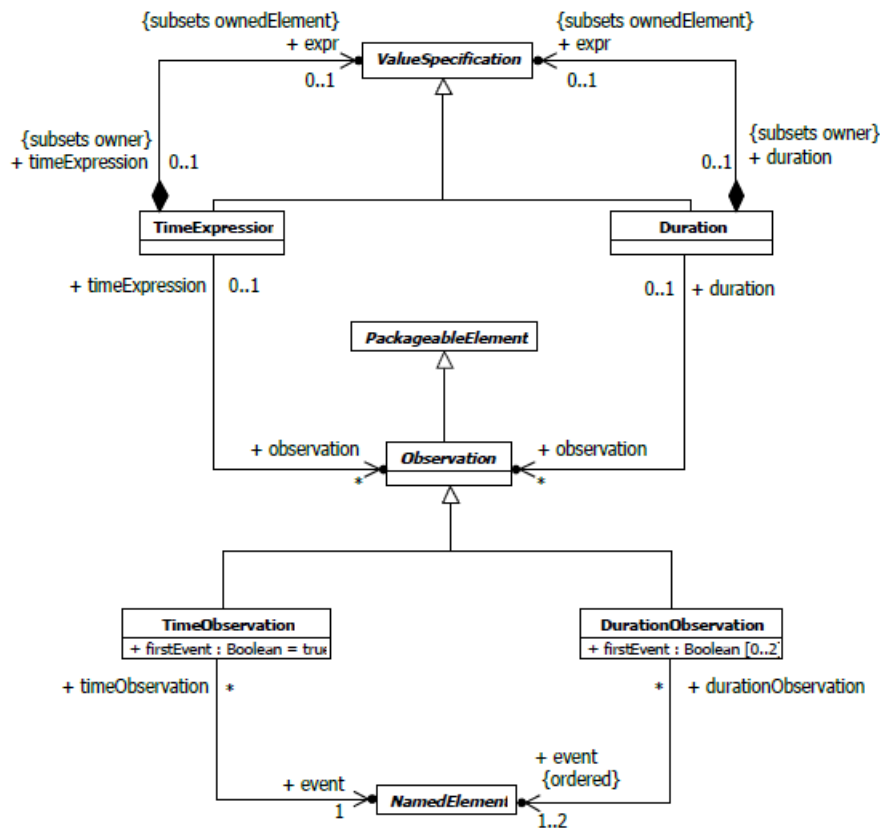


FIGURE 3.1 – Le temps et les durées dans UML

```

1 Grammaire
2 <time-event> ::= <relative-time-event> | <absolute-time-event>
3 <relative-time-event> ::= 'after' <time-expression>
4 <absolute-time-event> ::= 'at' <time-expression>

```

FIGURE 3.2 – Grammaire de la classification temporelle proposée dans Figure 8.1

MARTE pour UML

Introduction

MARTE [?] est une extension du langage de modélisation UML, créée par l'OMG, pour prendre en considération les systèmes temps réel et embarqués (STRE). Le changement le plus important dans cette extension est le profil UML qui a été remplacé par le profil MARTE. Ce nouveau profil apporte un support pour la spécification, la vérification, la validation ainsi que la conception des STRE. Le bénéfice apporté par cette extension est donc :

- Avoir une manière commune de modéliser les systèmes temps réel et embarqués.
- Permettre l'interopérabilité entre les différents outils de développement utilisés pour la spécification/vérification des STRE.
- Favoriser la conception de modèles qui peuvent être utilisés pour faire des prédictions quantitatives concernant le temps réel et les caractéristiques des systèmes embarqués.

1 La Figure 3.3 représente la relation de MARTE avec les autres standards de l'OMG.

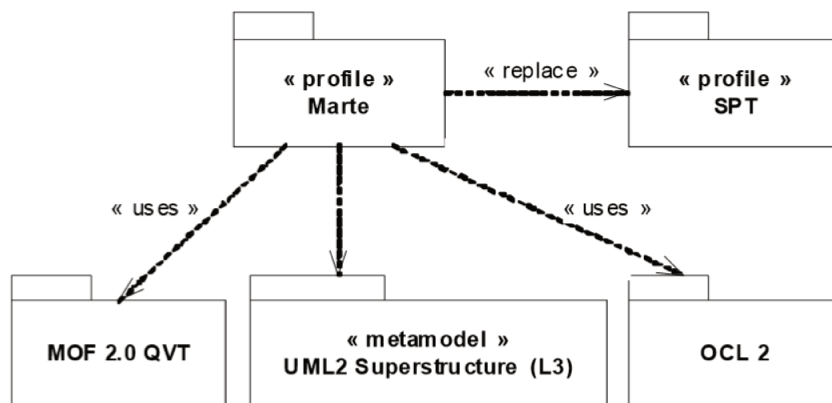


FIGURE 3.3 – MARTE et les standards de l'OMG

2 Comme le montre la Figure 3.3, le profil MARTE remplace l'ancien profil, il est basé sur le
3 méta-modèle d'UML et utilise les autres standards (QVT, OCL).

4 L'architecture du profil MARTE (Figure 3.4) est basée sur deux concepts principaux :

- 5 • Le premier concept, pour la modélisation des caractéristiques des STRE, est représenté
6 par *MARTE design Model*.
- 7 • Le second concept, pour l'annotation du modèle afin de prendre en compte l'analyse des
8 propriétés des systèmes, est représenté par *MARTE foundations*, *MARTE analysis model*
9 et *MARTE annexes*.

La Figure 3.4 montre l'architecture du profil MARTE.

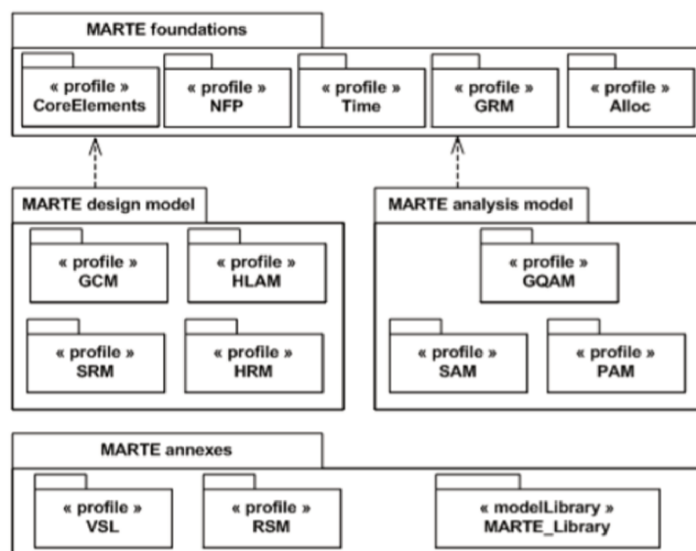


FIGURE 3.4 – L'architecture de MARTE.

10

11 Modélisation du temps dans MARTE

12 Le travail de [?] permet d'introduire la nouvelle extension de UML : MARTE (défini par
13 l'OMG) pour la modélisation et l'analyse des systèmes temps réel. Ce nouveau profil remplace

1 l'ancien profil UML SPT (UML profile for schedulability, performance and time) qui prenait
 2 en considération le temps mais avec des aspects limités. L'ancien profil prend en compte la
 3 notion de temps/ressources quantifiables, cela permet d'annoter le model avec des informations
 4 quantitatives concernant le temps, les performance et l'analyse de l'ordonnancement. L'ancien
 5 profil ne considère que le temps chronométrique (physique) sous forme de durée et d'instant afin
 6 de modéliser les événements dans le temps. Dans MARTE le temps peut être physique (discrétisé
 7 ou dense) ou bien logique (lié à des horloges définies par l'utilisateur).

8 La structure du temps prise en considération par MARTE est basé sur *TimeBase* tel que
 9 montré dans la Figure 3.5.

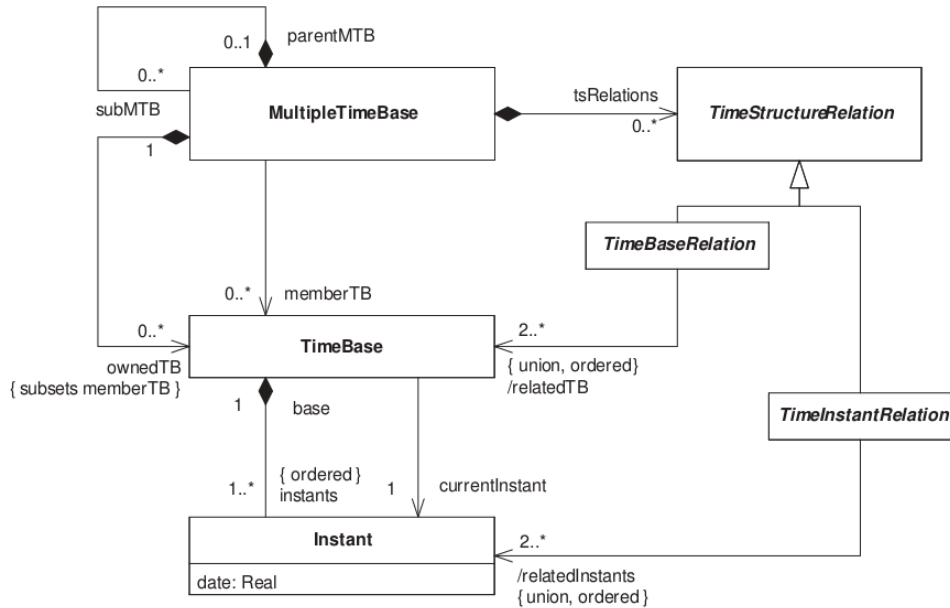


FIGURE 3.5 – La structure du temps dans MARTE ([?]).

10 *TimeBase* est un ensemble ordonné d'instants qui peut être dense ou discrétisé. Plusieurs
 11 bases existent afin de représenter le cas d'applications distribuées. Ces bases peuvent être dépen-
 12 dantes dans le cas où leurs instants sont liés par des relations de coïncidence ou de précédence.

13 *TimeBase* représente la structure du temps, cependant pour y accéder au temps MARTE
 14 utilise un autre concept : l'horloge. L'horloge (*clock*) est un élément du modèle donnant accès à
 15 la structure du temps (qui peut être logique, physique ou les deux). Chaque horloge fait référence
 16 à la base du temps et à ses instants (indirectement). Comme montré dans la Figure 3.6 l'horloge
 17 possède plusieurs attributs qui définissent sa nature ainsi que d'autres éléments nécessaires à sa
 18 représentation.

19 L'attribut *nature* définit la nature de l'horloge et donc des instants (dense ou discrétisé).
 20 L'attribut *resolution* définit la granularité de lecture de l'horloge et *origin* donne le décalage
 21 possible dans la lecture de l'horloge. L'horloge accepte l'unité (une propriété non fonctionnelle
 22 dans MARTE) et nous pouvons lui associer un événement qui apparaîtra à chaque changement
 23 dans cette horloge. Selon la nature du temps nous distinguons deux types d'horloges :

- 24 • L'horloge logique peut être définie par n'importe quel événement tel que, à chaque appa-
 25 rition de l'événement, l'horloge change. Le temps est logique et il est calculé en se basant
 26 sur le nombre de ticks (une unité est prédéfinie par défaut pour les horloges logiques).
- 27 • L'horloge chronométrique fait référence au temps physique. Nous pouvons attacher à ces
 28 horloges des informations quantitatives si nécessaire.

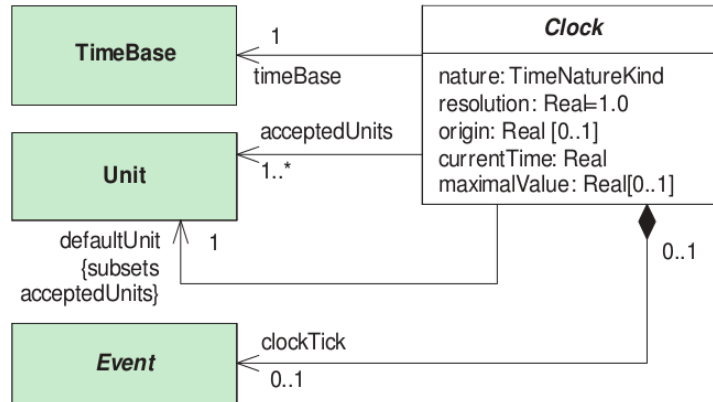


FIGURE 3.6 – L’horloge dans MARTE ([?]).

1 Un autre concept dans MARTE est la spécification de la valeur du temps *TimeValue* et
 2 deux notions sont proposées : *InstantValue* et *DurationValue*. Tel que montré dans la Figure 3.7,
 3 pour chaque horloge une valeur de temps (*TimeValue*) est associée. Cette association permet de
 4 définir l’unité correspondante à l’horloge, ses instants ainsi que ses durées. L’attribut *nature* de
 5 *TimeValue* permet de définir la nature du temps associé à l’horloge dans le cas d’un domaine
 6 discrétisé ou dense.

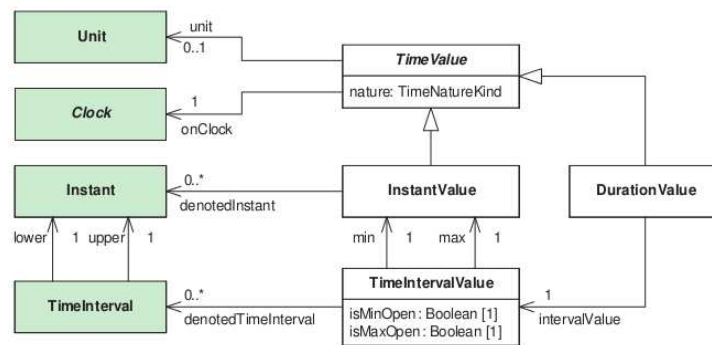


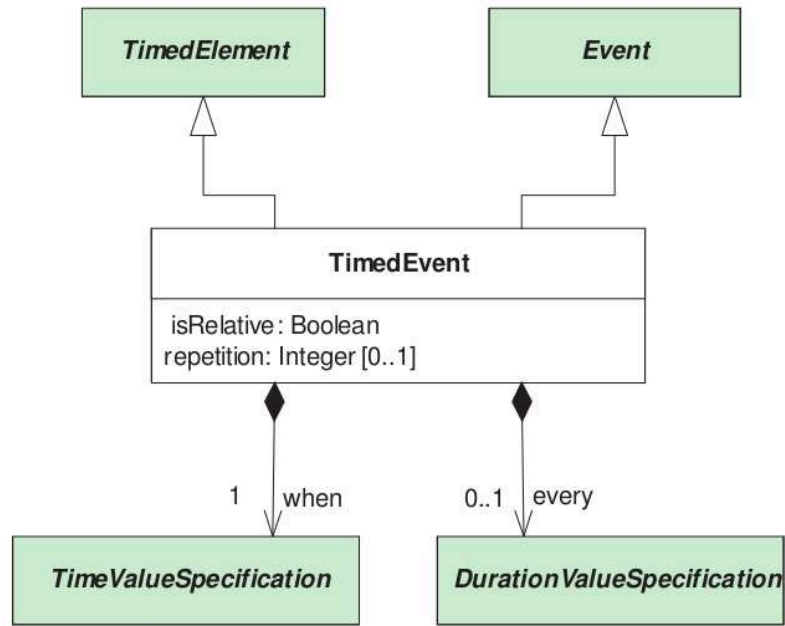
FIGURE 3.7 – TimeValue dans MARTE ([?]).

7 Comme mentionné précédemment, chaque horloge logique peut être liée à un événement mais
 8 aussi à un comportement. L’occurrence d’un événement peut être représentée par des points du
 9 temps (des instants) et l’exécution d’un comportement peut être représentée par des instants de
 10 début et de fin ou une durée d’exécution. Comme le montrent les Figure 3.8a et Figure 3.8b,
 11 *TimedEvent* et *TimedProcessing* sont deux concepts qui permettent la représentation des événe-
 12 ments et des comportements avec du temps (indirectement avec les horloges).

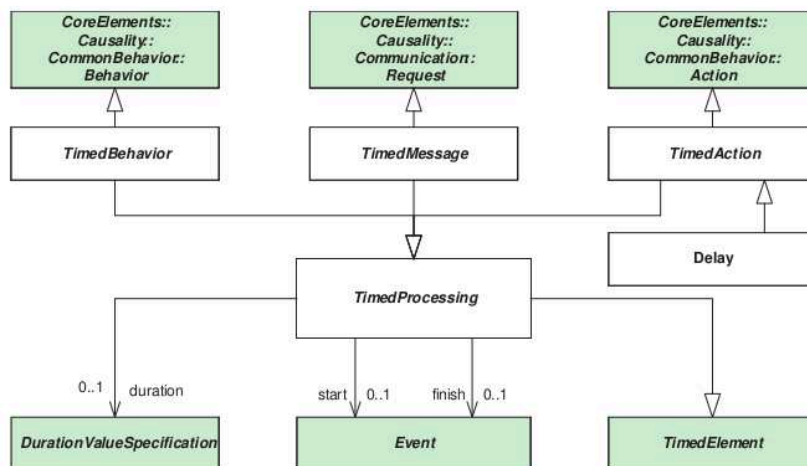
13
 14 Les différents concepts décrits ci-dessus sont représentés dans MARTE par un ensemble de
 15 *stéréotypes*. Chaque *stéréotype* va permettre l’utilisation des différents concepts dans la modéli-
 16 sation du temps mais aussi de permettre à l’utilisateur de définir ses propres horloges en utilisant
 17 la bibliothèque *TimeLibrary*.

18 La Figure 3.9 présente un exemple de diagramme états-transitions pour le cycle du moteur
 19 à quatre temps et une définition de l’horloge associée.

20 Dans la Figure 3.9a la définition de l’horloge qui va être utilisée pour la représentation du
 21 temps *AngleClock*. Cette horloge reprend les attributs du stéréotype de base *clockType* et elle
 22 prend comme unité *AngleUnitKind* (une unité définie par l’utilisateur). Les deux stéréotypes



(a) TimedEvent dans MARTE



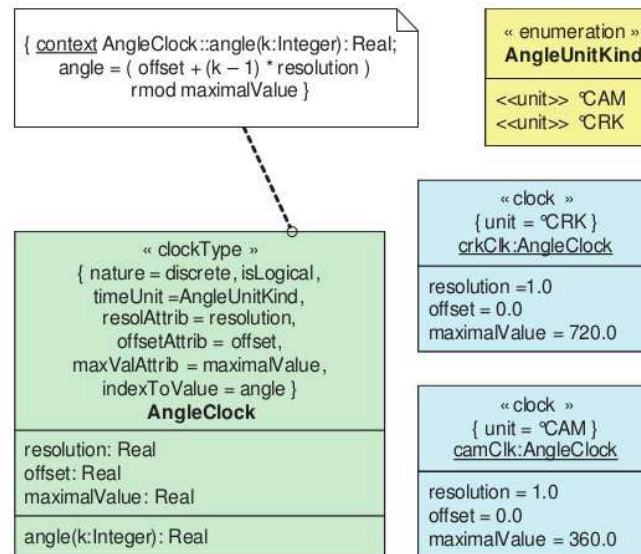
(b) TimedProcessing dans MARTE

FIGURE 3.8 – Liaison du temps avec l'événement et le comportement

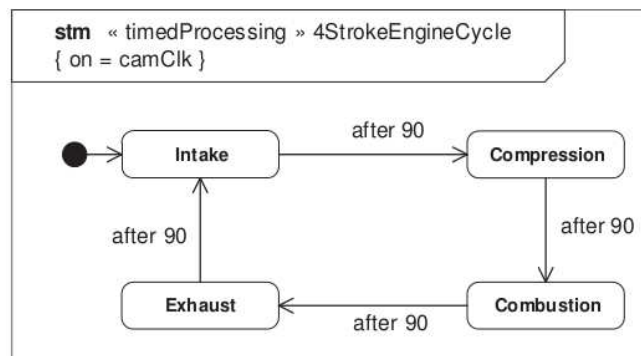
1 *camClk* et *crkClk* sont des instances de l'horloge principale avec des valeurs pour les attributs
 2 de bases.

3 Dans la Figure 3.9b le diagramme états-transitions est stéréotypé d'une part *timedProcessing*
 4 afin de lier le temps aux comportements (on parle de temps logique donc d'horloges logiques) et
 5 d'autre part *camClk* pour préciser l'horloge qui sera utilisée pour représenter le temps.

6 Le travail dans [?] étend MARTE avec un langage de spécification de contraintes d'horloges
 7 appelée *CCSL* (Clock Constraint Specification Language). Le langage *CCSL* est une part du
 8 profile MARTE mais qui n'est pas un standard. Il comprend une syntaxe pour la représentation
 9 graphique des différents éléments d'UML et une sémantique formelle. *CCSL* permet de spécifier
 10 dans le cadre d'UML, des contraintes aussi bien de temps chronométrique que de temps logique.
 11 L'auteur propose deux notations visuelles alternatives pour la représentation des contraintes afin
 12 d'obtenir des diagrammes plus compacts.



(a) Les stéréotypes de l'horloge utilisée



(b) Diagramme états-transitions du moteur à quatre temps

FIGURE 3.9 – Exemple d'utilisation du temps en MARTE dans les diagrammes états-transitions

1 Transformation des diagrammes états-transitions vers les réseaux de Petri stochastiques

2
3 Dans cette section, nous présentons une approche proposée dans les travaux de [?] pour
4 faire une transformation des diagrammes états transitions UML avec la notion du temps vers les
5 réseaux de Petri stochastiques. Les diagrammes états transitions UML ainsi que le profil UML
6 sont utilisés pour modéliser les systèmes temps réel. Cette combinaison est appelée RT-UML et
7 permet la spécification des propriétés quantitatives.

8 Diagrammes états transitions UML avec le temps

9 Différents éléments syntaxiques sont présentés mais la partie la plus importante est la notion
10 de temps. Le temps est représenté grâce à des *stereotypes* dans le profil UML. Chaque *stereo-*
11 *type* contient des *annotations* ainsi que des *taggedvalues* qui peuvent représenter des paramètres
12 quantitatifs (e.g. le temps), des performances et des ressources.

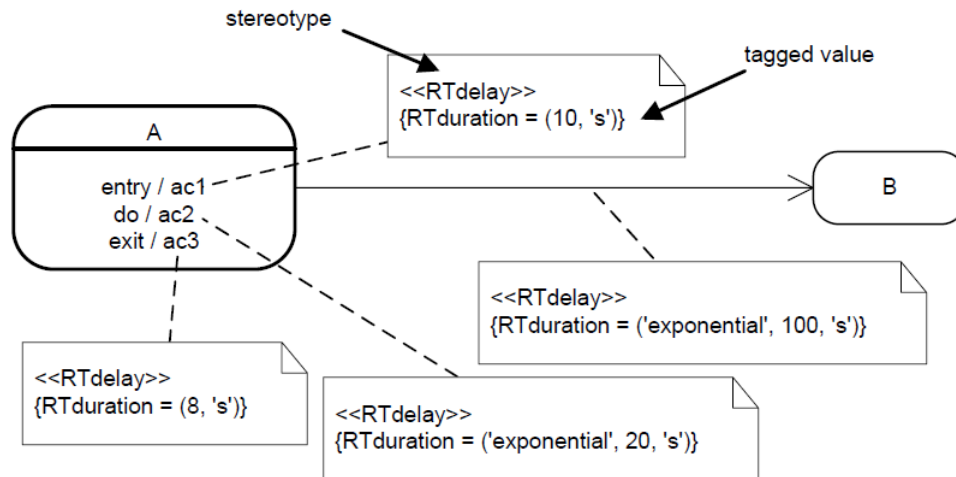


FIGURE 3.10 – Diagramme états transitions UML avec le temps

1 La Figure 3.10 représente un exemple de diagramme états transitions avec des stereotypes
 2 pour représenter le temps.

3 Réseaux de Petri stochastiques (SPN)

4 Les réseaux de Petri stochastiques comportent les mêmes éléments que les réseaux de Petri
 5 standard, mais ils introduisent différentes notions de transitions. Nous distinguons :

- 6 • *Transition immédiate* : une transition qui sera exécutée directement (e.g. une transition à
 7 partir d'un état initial dans le diagramme états transitions).
- 8 • *Transition déterministe* : une transition qui est exécutée après un passage d'une durée de
 9 temps représentée par une valeur.
- 10 • *Transition exponentielle* : une transition qui est exécutée après un passage d'une durée de
 11 temps représentée par une fonction exponentielle.
- 12 • *Transition générale* : dans les autres cas.

13 La Figure 3.11 représente un exemple de réseau de Petri stochastique avec les différents types
 14 de transitions : t1 est une transition exponentielle avec un taux $\lambda = 1/4$, t3 est une transition
 15 immédiate, t2 une transition déterministe avec une durée de 3 secondes et t4 une transition
 16 générale.

17 Transformation

18 Il s'agit dans cette section de présenter la transformation élaborée dans [?] et qui permet le
 19 passage des diagrammes états transitions avec le temps vers les réseaux de Petri stochastiques.

20 **États et transitions.** Chaque état (e.g. A sur la Figure 3.12) est représenté par une place
 21 SPN (A) qui modélise l'état en lui-même et deux autres places. L'une pour modéliser la phase
 22 avant l'exécution du comportement d'entrée de A (`ent_A`) et l'autre pour modéliser la phase
 23 après l'exécution du comportement de sortie de A (`out_A`). Chaque comportement de l'état est
 24 modélisé aussi par un ensemble de places et de transitions. Le comportement `do` est modélisé par

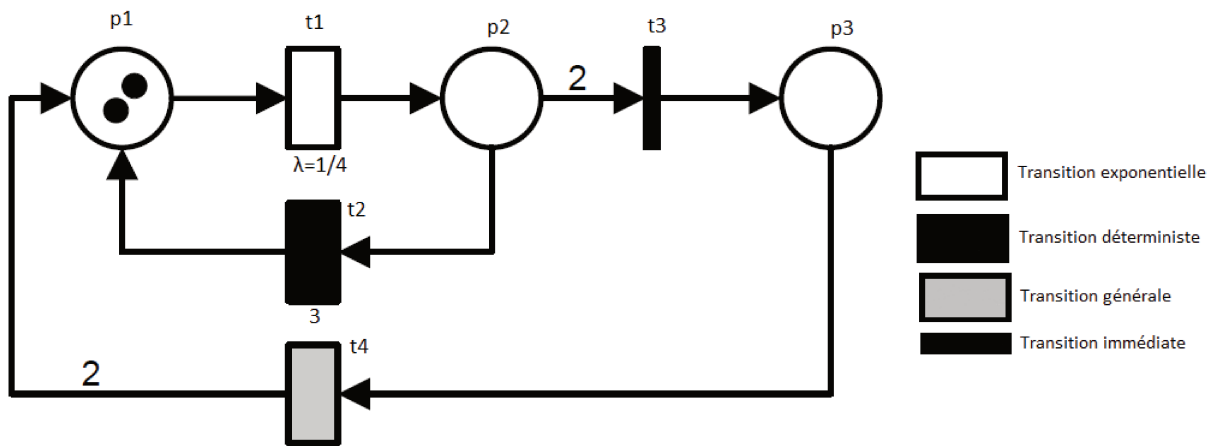


FIGURE 3.11 – Réseaux de Petri stochastiques

- 1 une transition (t_do_A), le comportement d'entrée par une transition (t_ent_A) et le comporte-
 2 tement de sortie par une place (ex_A) et une transition (t_ex_A). La Figure 3.12 représente la
 traduction d'un état avec ses comportements.

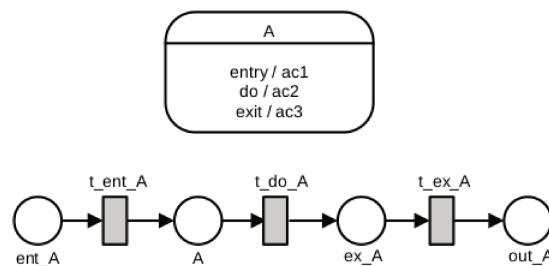


FIGURE 3.12 – Traduction d'un état

3
 4 Chaque transition UML est représentée par une transition SPN correspondant à sa nature
 5 (immédiate, exponentielle, générale ou déterministe).

6 Remarquons que la hiérarchie entre les états et les comportements d'entrée/sortie n'est pas
 7 claire. Notons également que, concernant le `do`, on n'a pas la possibilité de l'exécuter autant
 8 qu'on veut et on a seulement la possibilité d'ajouter une durée de temps à la transition. Dans
 9 notre travail le comportement `do` est exécuté autant de fois qu'on veut en le modélisant par une
 10 transition avec un arc qui va de la transition vers la place qui modélise l'état et un autre arc qui
 11 va de la place qui modélise l'état vers la transition.

12 **Pseudo-état initial** Chaque pseudo-état initial est représenté par une transition et une place
 13 comme montré dans la Figure 3.13.

14 La Figure 3.13a représente le cas simple d'un pseudo-état initial vers un état simple. Le
 15 pseudo-état initial est modélisé par une place `init_A` avec sa transition qui va vers la place
 16 `ent_A` qui modélise la phase avant l'entrée dans l'état `A` (`A` peut avoir la même modélisation que
 17 dans Figure 3.12). La Figure 3.13b représente un cas plus complexe avec un pseudo-état initial
 18 vers un état orthogonal. Chaque région a son pseudo-état initial (modélisé par une place et une
 19 transition, e.g. `init_A1`) avec chaque pseudo-état connecté avec le pseudo-état initial de l'état
 20 composite (`init_S1`).

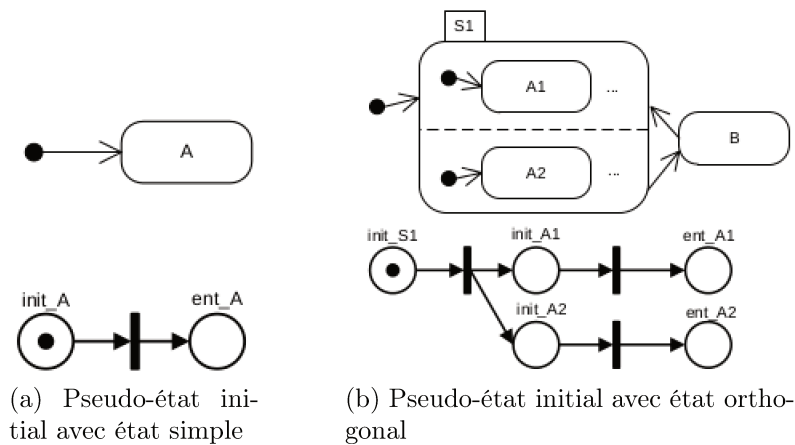


FIGURE 3.13 – Traduction d'un pseudo-état initial

- 1 **Pseudo-états Fork et Join** Un pseudo-état Fork est représenté par une transition principale et un couple place/transition associé à chaque région de l'état composite. Un pseudo-état
- 2 Fork est représenté par une place et deux transitions. La Figure 3.14 montre la traduction d'un
- 3 exemple contenant les pseudo-états Fork et Join.
- 4

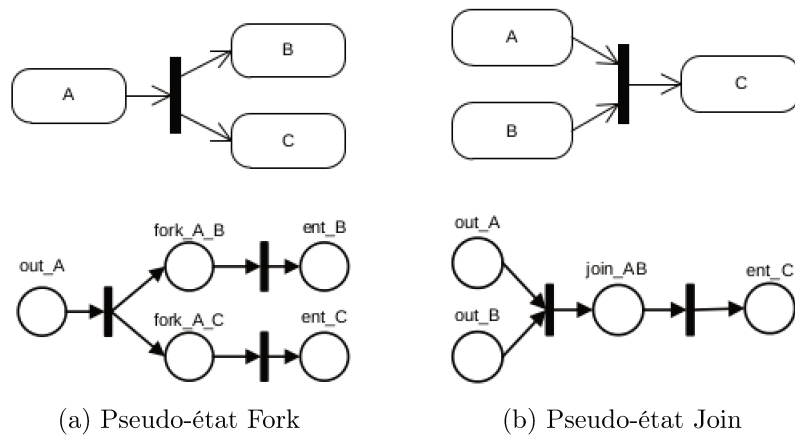


FIGURE 3.14 – Traduction des pseudo-états Fork et Join

- 5 Le pseudo-état Fork qui va de l'état A vers les états B et C dans la Figure 3.14a est modélisé
- 6 par une transition principale et un couple de place/transition pour chaque état B et C (e.g. la
- 7 place `fork_A_B` et sa transition qui va vers `ent_B`). Le pseudo-état Join qui va des états A et B
- 8 vers l'état C dans la Figure 3.14b est modélisé par une place `join_AB` et ses deux transitions, la
- 9 première qui relie `out_A` et `out_B` avec `join_AB` et la seconde qui relie `join_AB` avec `ent_C`.

- 10 **Pseudo-état choice** Le pseudo-état choice est représenté par une place/transition (e.g.
- 11 `choice_B` et sa transition qui va de `out_B` vers `choice_B`) principale avec une place/transition
- 12 (e.g. `choice_B_A` et `t_trans_BA`) pour chaque branche de ce pseudo-état comme montré dans
- 13 l'exemple de la Figure 3.15.

- 14 **Point de jonction** Un point de jonction est modélisé par un ensemble de place/transition
- 15 entre les différents états liés par le point de jonction. Comme l'exemple de la Figure 3.16 le

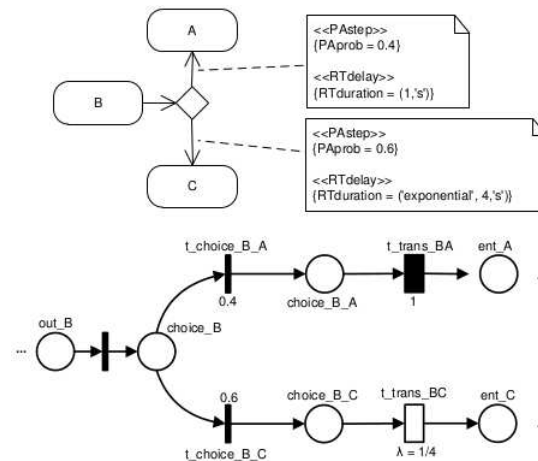


FIGURE 3.15 – Traduction d'un pseudo-état choice

1 montre, le point de jonction entre les états A et B ainsi que C et D est modélisé de la manière
 2 suivante :

- 3
- 4 • Une place (`junc_A_CD`) pour modéliser la jonction entre l'état A et les deux états C et
 5 D. Une transition qui lie `out_A` avec la place `junc_A_CD` et deux autres transitions pour
 6 faire la liaison entre le place `junc_A_CD` et les deux autres places `ent_C` et `ent_D` (les
 7 transitions sont modélisées avec des gardes `g1` et `g2`).
 - 8 • Une place (`junc_B_CD`) pour modéliser la jonction entre l'état B et les deux états C et
 9 D. Une transition qui lie `out_B` avec la place `junc_B_CD` et deux autres transitions pour
 10 faire la liaison entre le place `junc_B_CD` et les deux autres places `ent_C` et `ent_D` (les
 transitions sont modélisées avec des gardes `g1` et `g2`).

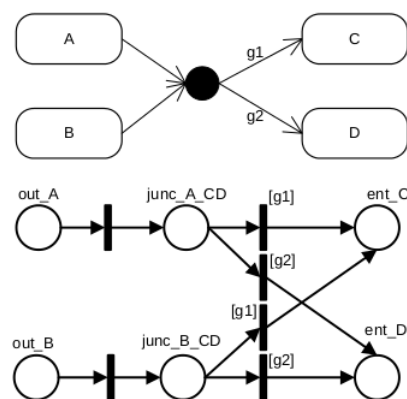


FIGURE 3.16 – Traduction d'un point de jonction

11 **Notion du temps** Le temps dans une transition UML définit le type de transition corres-
 12 pondante en réseaux de Petri stochastiques :

- 13
- (8,'s') : une transition déterministe qui dure 8 secondes.
 - 14 • ('exponential', 32, 's') : une transition exponentielle avec un taux $\lambda = 1/mean$.

- 1 • ('percentile', 80, (5,'s'),'exponential') : une transition exponentielle avec un taux calculé à
 2 partir de $F(x) = 1 - e^{-\lambda x}$.

3 **Événements** Un événement est modélisé par une place et une transition qui l'engendre (avec
 4 la possibilité d'avoir une durée de temps au niveau de la transition, e.g. la transition `t_gen_ev1`
 5 qui engendre l'événement `ev1` dure 2 secondes) comme montré dans la [Figure 3.17](#).

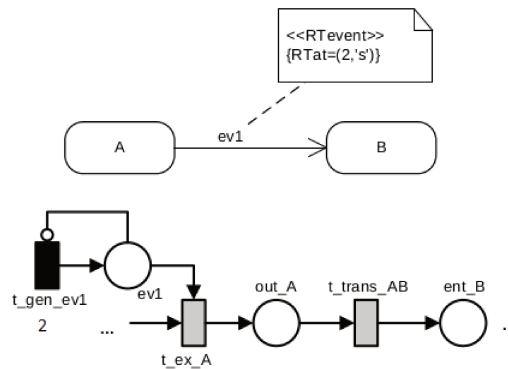


FIGURE 3.17 – Traduction d'une transition avec événement

6 **Remarques** Selon la conclusion de l'article, les pseudo-états historiques (les deux types) ainsi
 7 que d'autres notions dans le profil UML ne sont pas pris en compte. Remarquons aussi qu'il n'y
 8 a pas de description concernant les variables ni la hiérarchie des comportements des différents
 9 états.

10 Cette approche souffre de quelques inconvénients, le premier est le fait que, même si un état
 11 n'a pas de comportement, il sera quand même modélisé (e.g. dans la [Figure 3.17](#) l'état A n'a
 12 pas de comportement de sortie mais il y a une transition qui le modélise `t_ex_A`). Le second
 13 inconvénient est la présence de beaucoup de places/transitions qui peuvent être factorisées. Cet
 14 inconvénient conduit rapidement au problème de l'explosion combinatoire de l'espace d'états
 15 dans le cas de la vérification des réseaux de Petri stochastiques.

16 Validation des systèmes avec du temps en utilisant les statecharts 17 et les automates temporisés

18 Nous présentons dans cette section, une approche de vérification (définie dans [?]) basée
 19 sur les diagrammes états-transitions UML ainsi que les automates temporisés. Cette approche
 20 permet de définir les contraintes de temps ainsi que les schémas (patterns) des diagrammes états-
 21 transitions pour ensuite faire la transformation de ces schémas vers les automates temporisés.

22 Selon l'article [?] dans l'état de l'art il y a deux approches, d'une part celle basée sur les
 23 techniques de vérification de modèles. Les approches basées sur la vérification de modèles qui
 24 prend en entrée le système modélisé en un diagramme d'états-transitions avec les propriétés à
 25 vérifier (modélisées en logique temporelle). Le résultat sera oui dans le cas où la propriété est
 26 vérifiée et non avec un contre-exemple dans le cas inverse. D'autre part les approches qui sont
 27 basées sur l'utilisation d'observateurs. Les observateurs sont des modules synchronisés avec la
 28 spécification du système et qui permettent de vérifier si un état (caractérisant une violation de
 29 propriété) est atteignable. Pour prouver que la propriété observée par l'observateur est vraie, il
 30 suffit de prouver que l'état erreur n'est pas atteignable.

La comparaison des différents travaux de la littérature avec cette approche montre les résultats suivants (selon l'article [?]) :

- L'utilisation des diagrammes états-transitions UML est plus expressive pour la modélisation des systèmes temps réel, que l'utilisation des diagrammes de séquences ou de collaboration.
- Aucune logique temporelle n'est nécessaire pour la spécification du temps dans les systèmes, sachant que ces besoins sont introduits sous forme de modèles.
- Les schémas seront définis pour simplifier la spécification des besoins temporels.
- Surmonter la limitation des outils de vérification de modèles en restreignant la vérification à l'analyse d'accessibilité.

Idée générale

La première étape consiste à définir les schémas d'observation des SMD ensuite la seconde étape consiste à vérifier en se basant sur les bases des schémas d'observation.

Les schémas sont des modèles réutilisables dans les logiciels du système et qui garantissent un ensemble de fonctionnalités. L'utilisation des schémas permet de ne pas faire de modification dans l'outil de design et de garder une vue standard des outils utilisés. Dans l'approche de l'article [?], un ensemble de schémas est défini pour les besoins de temps. Le rôle de chaque pattern est d'exprimer un besoin temporel. La Figure 3.18 montre le processus général de l'utilisation des schémas.

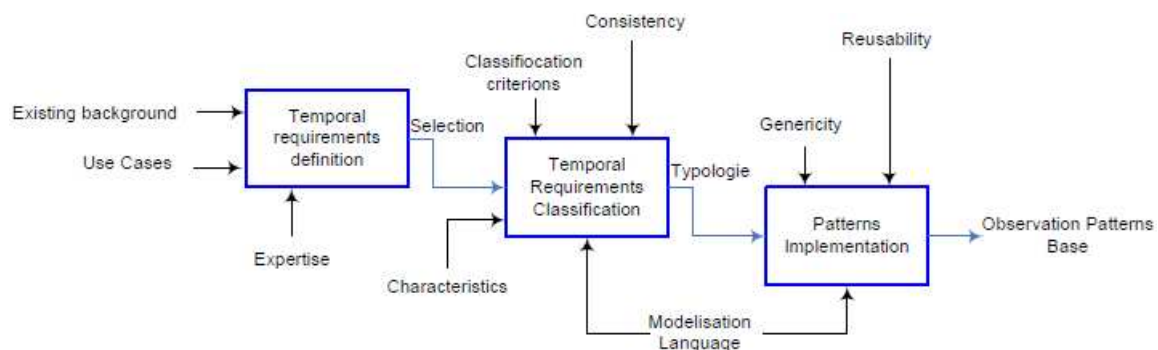


FIGURE 3.18 – Le processus général de l'utilisation des schémas [?]

Comme le montre la Figure 3.18 afin de définir les schémas et leurs implémentations nous avons besoin passer par deux étapes. La première consiste à définir les besoins temporels, leurs définitions en utilisant les définitions existantes. La seconde consiste à faire une classification des besoins en se basant sur le résultat trouvé dans l'étape précédente et en utilisant les critères adéquats.

La Figure 3.19 suivante montre comment les bases des schémas sont utilisées dans le processus de vérification. Pour une spécification d'un système donné, les besoins temporels de ce système sont identifiés. Pour chaque besoin temporel un pattern adéquat est instancié à partir des schémas de base. Cette instance engendre les SMD observateurs tels que chaque observateur représente un besoin. Ensuite chaque observateur est transformé en un automate temporisé en utilisant une technique de modèle de transformation. Les automates temporisés engendrés sont ensuite synchronisés avec la spécification du système pour engendrer le modèle global ce qui réduit la vérification à une simple analyse d'accès à un état erreur sur le modèle obtenu.

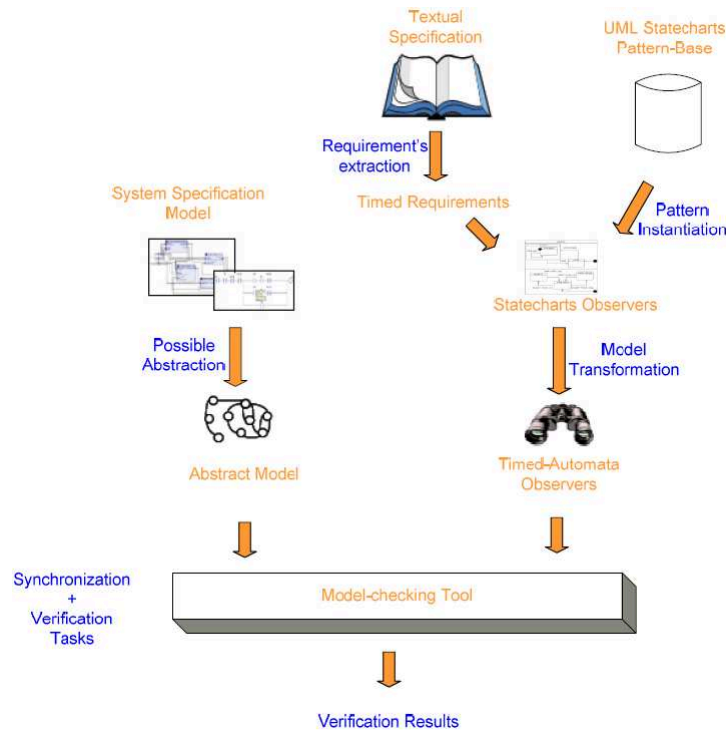


FIGURE 3.19 – Le processus de vérification [?]

1 Classification des contraintes de temps

L'approche de l'article [?] propose une nouvelle classification des contraintes de temps. Le temps dans cette approche est utilisé pour distinguer les apparitions des événements des différentes transitions entre états. La Figure 3.20 montre les contraintes temporelles prises en compte dans l'approche.

La typologie des besoins temporels est divisée en deux classes majeures : la première concerne les besoins quand le temps est exprimé d'une manière qualitative. Cette classe de besoins considère un ordre entre les événements. La seconde concerne les besoins quand le temps est exprimé d'une manière quantitative. Cette classe de besoins considère le temps vis-à-vis des événements (le temps de leurs apparitions, le séquençement entre les différents événements, etc). Selon les besoins il y a six types présentés tels que (si un système S satisfait un besoin R alors nous notons $S \models R$) :

- MinimumDelay : $S \models R$ est vrai si l'événement apparaît après T_{min} (R affirme que l'événement doit apparaître après un temps minimum T_{min})
- MaximumDelay : $S \models R$ est vrai si l'événement apparaît avant T_{max} (R affirme que l'événement doit apparaître avant un temps maximum T_{max})
- Latency : $S \models R$ est vrai si l'événement apparaît dans l'intervalle temporel $]T_{min}; T_{max}[$
- Delay : $S \models R$ est vrai si l'événement apparaît exactement au temps T
- Simultaneity : $S \models R$ est vrai si $Event1.occurence = Event2.occurence$
- Sequence : $S \models R$ est vrai si $Event1.occurence <> Event2.occurence$

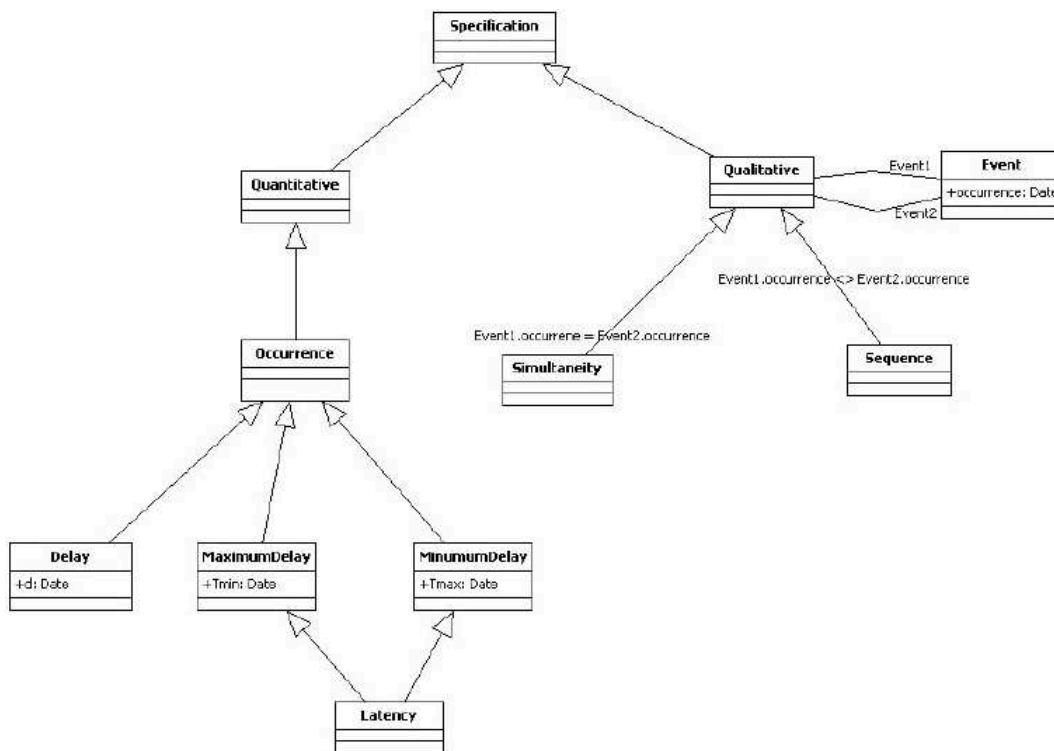


FIGURE 3.20 – Classification des contraintes de temps [?]

1 Schémas observateurs (patterns observateurs)

2 Un pattern est une représentation qui permet de décrire les comportements récurrents. La Figure 3.21 présente un exemple de pattern représentant une contrainte de temps "Delay". Afin

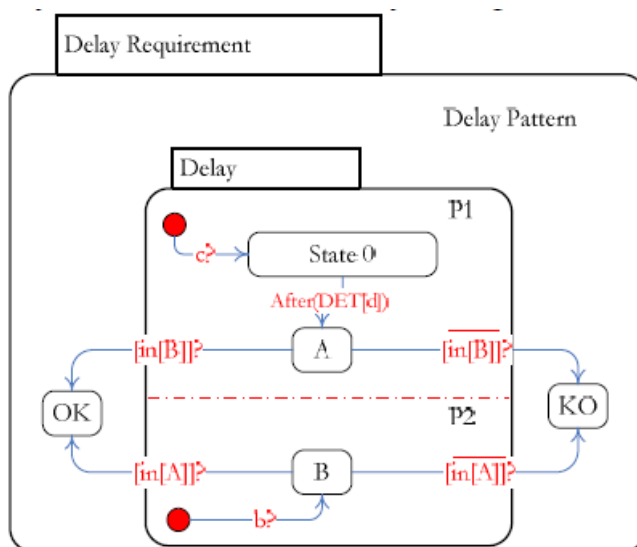


FIGURE 3.21 – Pattern de la contrainte "Delay" [?]

3
 4 d'assurer que le processus de vérification des besoins n'influe pas la spécification du système, les
 5 observateurs sont définis sous forme de blocs extérieurs qui seront composés avec le modèle de
 6 spécification du système. Les observateurs sont instanciés à partir des schémas qui sont spécifiés

1 avec le modèle SMD car le but de la tâche de vérification est d'assurer la non-violation des
 2 besoins. Dans la Figure 3.21 il y a un nœud KO tel que si ce nœud est atteignable alors il y a
 3 une violation du besoin. L'état KO est atteignable si nous ne sommes pas dans l'état A et dans
 4 l'état B en même temps. En d'autres termes, KO est atteignable si l'événement b n'apparaît pas
 5 exactement d temps après l'événement c.

6 Transformation des SMD vers les automates temporisés (TA)

7 La transformation est basée sur deux étapes, la première consiste à aplatir les SMD en
 8 supprimant la hiérarchie (car les automates ne prennent pas en considération ce concept). La
 9 seconde étape consiste à traduire la version aplatie des SMD vers les TA. La Figure 3.22 montre
 10 l'architecture de la transformation.

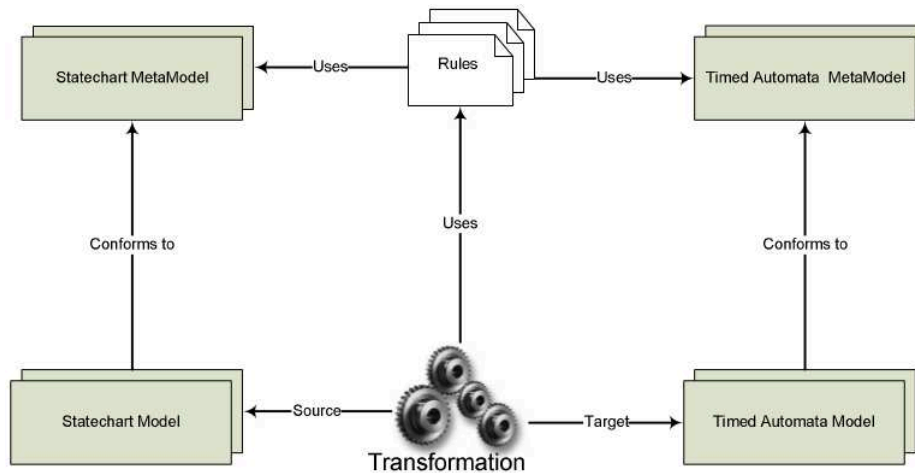


FIGURE 3.22 – Architecture de la transformation des SMD vers les TA [?]

11 Une fois les TA engendrés, ils seront synchronisés avec la spécification du système ensuite
 12 l'analyse d'accessibilité d'erreurs sera implémentée.

13 Model checking pour les diagrammes états-transitions et de colla- 14 borations avec du temps

15 Knapp *et al.* [?] propose une approche pour vérifier les systèmes temps-réel en utilisant les
 16 diagrammes états-transitions, les diagrammes de collaboration UML et les automates temporisés
 17 d'UPPAAL.

18 Afin d'illustrer leur approche, les auteurs de ce travail utilise l'exemple du passage à niveau
 19 du train. Les auteurs présentent les contraintes temporelles que le train doit respecter dans la
 20 Figure 3.23.

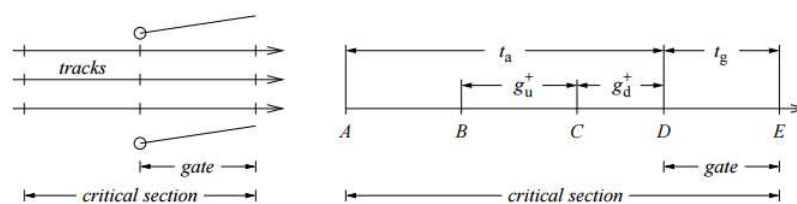


FIGURE 3.23 – Annotation temporelle de l'exemple du passage de train [?]

- 1 Ils présentent également le diagramme de classes et les diagrammes états-transitions (voir les
- 2 [Figures 3.24 à 3.27](#)) correspondants à l'exemple du train.

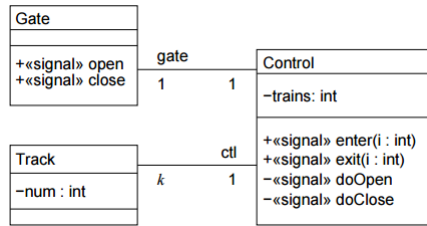


FIGURE 3.24 – Diagramme de classes de l'exemple du train [?]

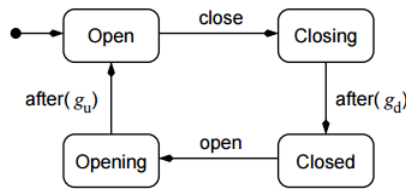


FIGURE 3.25 – Diagramme états-transitions de la barrière [?]

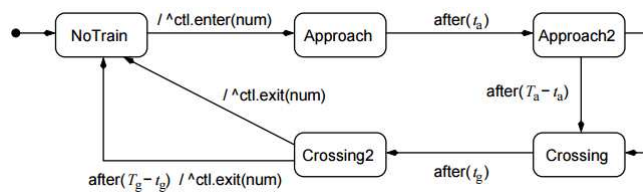


FIGURE 3.26 – Diagramme états-transitions du parcours du train [?]

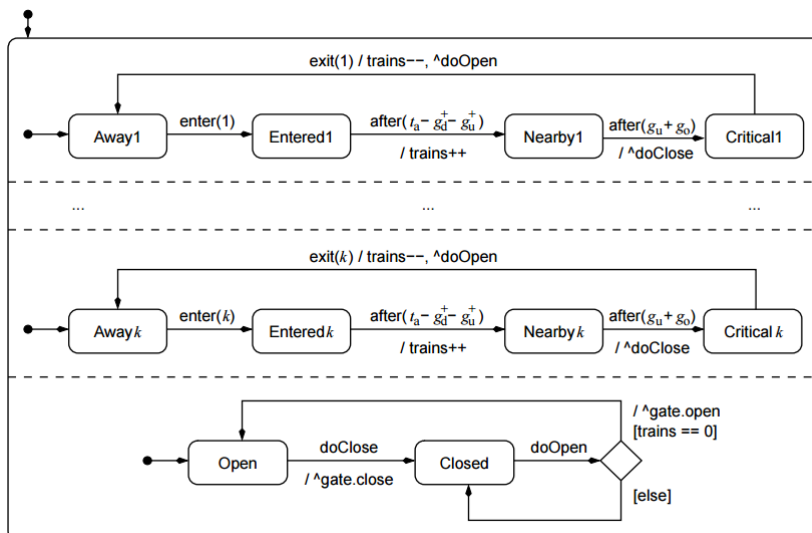


FIGURE 3.27 – Diagramme états-transitions du contrôleur [?]

- 3 Il est également présenté dans ce travail quelques propriétés concernant l'exemple du train
- 4 en utilisant le diagramme de séquence (voir la [Figure 3.28](#)).

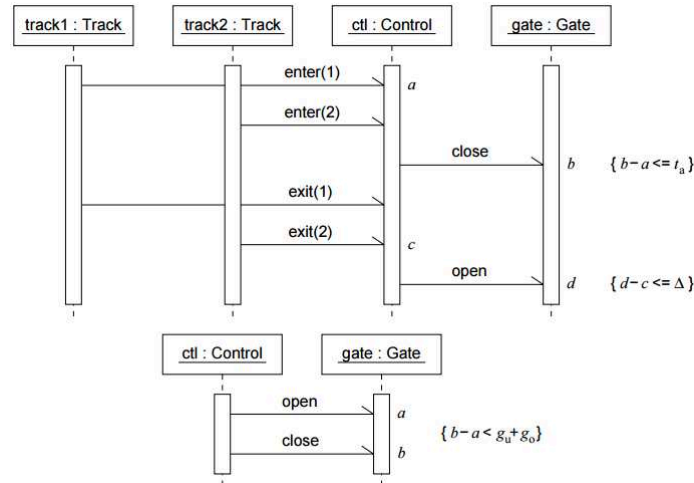


FIGURE 3.28 – Diagramme de séquence pour la propriété de sûreté et d'utilité [?]

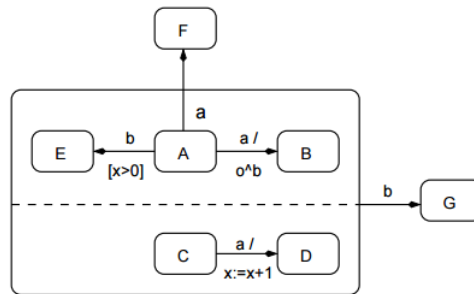


FIGURE 3.29 – Exemple diagramme états-transitions [?]

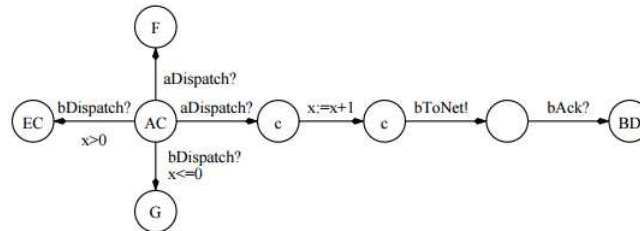


FIGURE 3.30 – Traduction de l'exemple de la Figure 3.29 [?]

1 Les Figures 3.29 et 3.30 montrent un exemple de traduction des diagrammes états-transitions
 2 vers les automates temporisés.

3 Dans ce travail les auteurs proposent un prototype d'outil appelé HUGO/RT qui permet
 4 de vérifier automatiquement si le scénario spécifié par le diagramme de collaboration avec les
 5 contraintes temporelles est réalisé avec l'ensemble des éléments temporels des diagrammes états-
 6 transitions. L'outil implémente une traduction des diagrammes états-transitions avec du temps
 7 vers les automates temporisés, et les diagrammes de collaboration annotés avec du temps vers
 8 les observateurs UPPAAL. Le modèle checking UPPAAL est utilisé pour vérifier l'automate
 9 temporisé.

Vérification formelle pour les systèmes temps réel

Tobias *et al.* [?] présente une extension des diagrammes états-transitions UML avec le temps. Les états simples, les états composites, les états orthogonaux, les événements, les pseudo-états histoire plat et profond ainsi que les points d'entrée et de sortie sont inclus dans l'ensemble des éléments syntaxiques considéré dans cette approche. Le modèle UML annoté avec le temps est traduit vers une extension des automates : les automates hiérarchique temporisés (HTA : hierarchical timed automata). Les HTA sont traduits ensuite vers les automates temporisés afin d'utiliser les outils existant tel que UPPAAL pour la vérification.

Afin d'expliquer et présenter leur approche, les auteurs de cet article utilise l'exemple du simulateur cardiaque (ou : Pacemaker). Les auteurs ont comparé le résultat de la vérification sur le modèle HTA et le modèle UPPAAL afin de démontrer que c'est faisable de faire de la vérification formelle sur les HTA.

SCCharts pour la vérifications des systèmes réactifs

Les auteurs de [?] présentent un nouveau langage visuel (les SSCharts : Sequentially Constructive Statecharts) pour la spécification des systèmes réactifs et critiques. Les SSCharts utilisent les notations des statechart et fournissent la concurrence en se basant sur un modèle asynchrone de calcul. La sémantique et les principales caractéristiques des SSCharts sont définis par un ensemble d'éléments. Les éléments syntaxiques considérés dans cette sémantique sont les suivants : les états (simples et composites avec régions), les variables, les transitions (avec ou sans événements), les pseudo-états fork, join et initial, les états finaux, le conflit entre transitions (dans le cas où les transitions ont le même état source) et enfin les "ticks" pour l'évaluation des transitions. Dans la [Figure 3.31](#) un exemple de SCCharts en utilisant différents éléments syntaxiques.

Dans cet article il y a également une approche de transformation modèle vers modèle (M2M) pour le passage des SSCharts vers les SC graphs (Sequentially Constructive graphs). La transformation des différents éléments est expliquée en utilisant des règles de transformation (voir [Figure 3.32](#)).

Il est nécessaire de préciser cependant que les auteurs ne précisent pas si dans la sémantique des SCCharts les éléments syntaxiques suivants sont considérés :

- Comportements d'entrée, de sortie et les comportements Do
- Les différents types de transitions : internes, locales ainsi que les fork et join implicite.
- Les priorités du franchissements des transitions avec événements et sans événements.
- Les pseudo-états histoire (profond ou plat).

Patterns pour les systèmes temps réel

L'auteur de [?] propose un ensemble de patterns fréquemment utilisés pour la spécification de l'exactitude des systèmes complexes et temps-réel. Ces patterns seront transformés ensuite en un ensemble de problèmes d'accessibilités afin d'éviter l'utilisation d'algorithmes complexes pour la vérification. Enfin, une instantiation de ces patterns vers les automates et CSP temporisés est proposée.

Les contraintes temporelles dans ce travail sont considérés en utilisant des horloges avec des valeurs réelles (donc du temps continu).

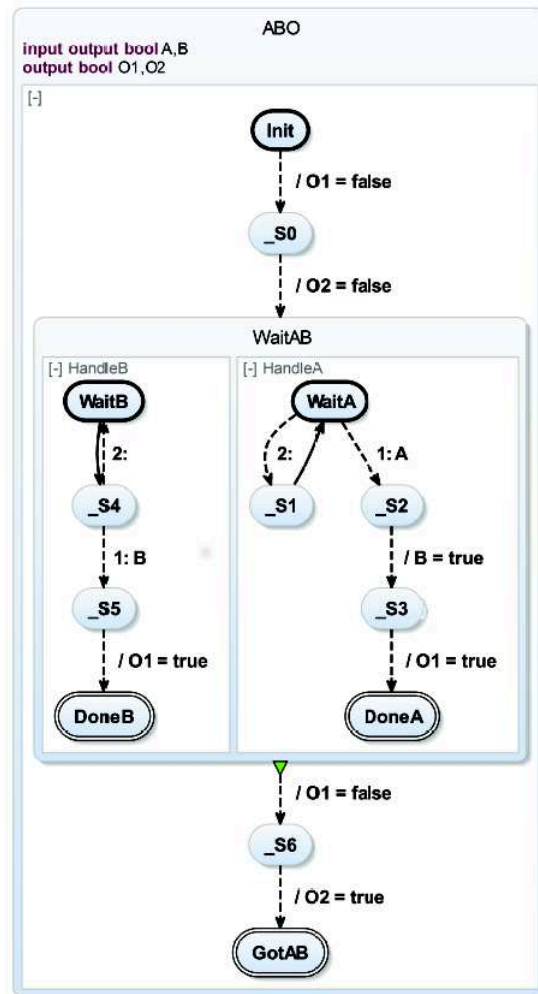


FIGURE 3.31 – Exemple de SCCharts [?]

1 L'auteur propose un ensemble de patterns (16 en tout) classifiés en 7 catégories. Chaque
 2 catégorie représente une propriété à spécifier et à vérifier dans les systèmes comme suit. La
 3 première propriété est celle de la non accessibilité d'un état.

4 La seconde propriété est celle de précédence où une action peut être effectuée seulement si
 5 une autre est effectuée avant elle (voir Figure 3.33).

6 La troisième propriété est celle de vivacité où une action est suivie d'un ensemble d'autres
 7 actions (voir Figure 3.34).

8 La quatrième propriété est celle de ponctualité où une action doit être effectuée au plus tard
 9 un certain délai (voir Figure 3.35) .

10 La cinquième propriété est celle de précédence avec un intervalle où une action peut être
 11 effectuée uniquement si une autre est effectuée dans un intervalle avant elle (voir Figure 3.36).

12 La sixième propriété est celle de latence où une action est suivie d'une autre dans un intervalle
 13 donné (voir Figure 3.37) .

14 La dernière propriété est celle de séquence, c'est-à-dire la séquence de plusieurs actions (voir
 15 Figure 3.38).

16 L'ensemble des patterns sont implémentés en utilisant l'outil *IMITATOR*. Cet outil permet
 17 la synthèse de paramètres pour une extension paramétrique des automates temporisés.

	Region (Thread)	Superstate (Parallel)	Trigger (Conditional)	Action (Assignment)	State (Delay)
Normalized SCCharts					
SCG					
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
Data-Flow Code	$d = g_{exit}$ $m = \neg \bigvee_{surf \in t} g_{surf}$	$g_{join} = (d_1 \vee m_1) \wedge$ $(d_2 \vee m_2) \wedge$ $(d_1 \vee d_2)$	$g = \bigvee g_{in}$ $g_{true} = g \wedge c$ $g_{false} = g \wedge \neg c$	$g = \bigvee g_{in}$ $x' = g ? e : x$	$g_{depth} = pre(g_{surf})$
Circuits					

FIGURE 3.32 – Règles de transformations des SCCharts vers les SC graphs [?]

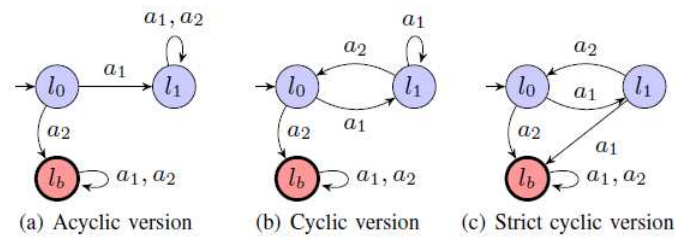


FIGURE 3.33 – Observateur automate temporisé pour la propriété de précédence[?]

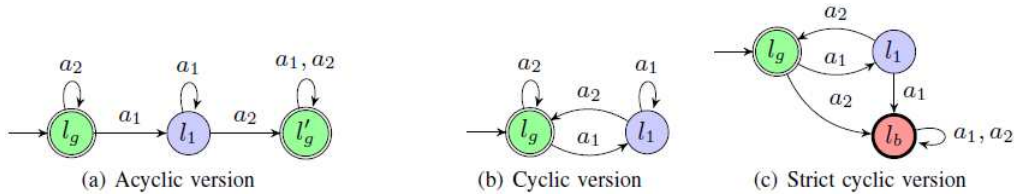


FIGURE 3.34 – Observateur automate temporisé pour la propriété de vivacité [?]

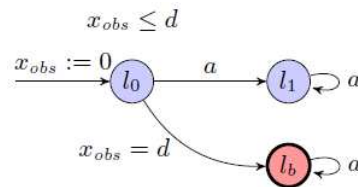


FIGURE 3.35 – Observateur automate temporisé pour la propriété de ponctualité [?]

1 Patterns pour la modélisation des systèmes temporisés

2 Afin de spécifier l'exactitude des systèmes complexes et temps-réel, les auteurs de [?] pro-
 3 posent un ensemble de patterns. Ces patterns encodent des composants de spécification ou de
 4 vérification qui ont relation avec les systèmes temps-réel. Les auteurs proposent également une

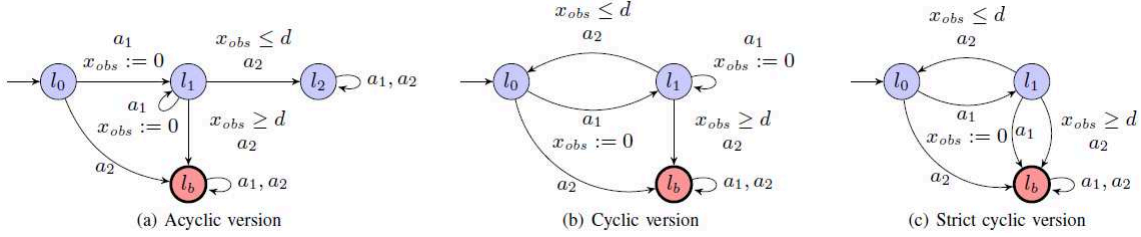


FIGURE 3.36 – Observateur automate temporisé pour la propriété de précédence avec un intervalle [?]

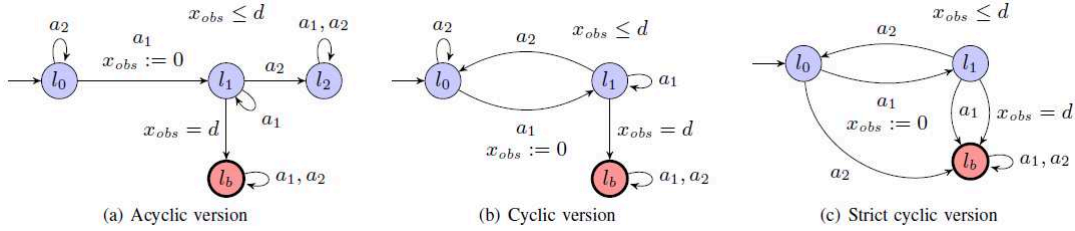


FIGURE 3.37 – Observateur automate temporisé pour la propriété de latence [?]

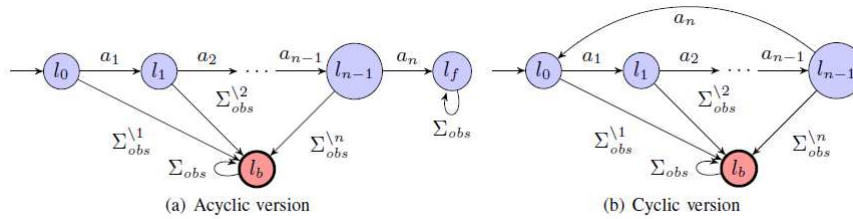


FIGURE 3.38 – Observateur automate temporisé pour la propriété de séquence [?]

1 sémantique pour ces patterns en utilisant les réseaux de Petri temporisés.

2 Les patterns proposés dans cet article permettent la spécification des systèmes temps-réel
 3 et la vérification de leurs propriétés. Ces patterns ont une syntaxe proche du langage humain
 4 afin de faciliter leur usage par les ingénieurs non-experts dans les méthodes formelles. Dans le
 5 contexte de la vérification, les patterns proposés sont traduits vers des propriétés d'accessibilité
 6 en utilisant de simples observateurs et cela afin d'éviter d'utiliser des algorithmes de vérification
 7 très complexes. Les auteurs présentent un ensemble de patterns existant ainsi que leurs nouveaux
 8 patterns unifiés et qui permettent également d'encoder les anciens patterns.

9 Les contraintes temporelles sont considérées dans ce travail en utilisant des notations syn-
 10 taxiques telles que : *AT LEAST*, *AT MOST*, *EXACTLY*. La Figure 3.39 montre la sémantique
 11 des patterns proposés dans ce travail en les traduisant vers les réseaux de Petri temporisés.

12 Passage des contraintes de planning vers les automates temporisés

13 Un travail pour transformer les contraintes de planning vers les automates temporisés en
 14 utilisant des règles est présenté dans [?]. Un ensemble de relations (ces relations sont similaires
 15 aux patterns) temporelles basées sur des intervalles est défini. Cet ensemble comporte 17 relations
 16 regroupées dans 6 catégories :

17 Les relations temporelles permettent de définir les contraintes temporelles entre les tâches.
 18 Ces tâches sont composés de deux événements qui sont le début et la fin de la tâche, et le temps
 19 est spécifié en utilisant des intervalles. Les relations utilisées dans ce travail sont présentées dans

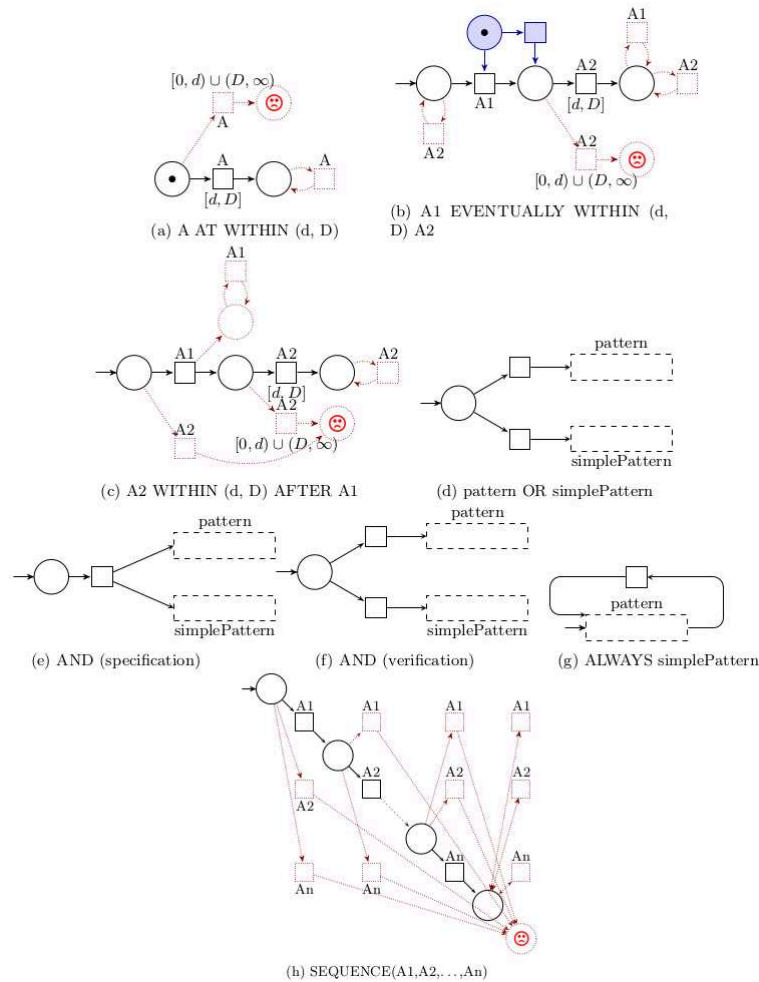


FIGURE 3.39 – Traduction des différents patterns vers les réseaux de Petri temporisés [?]

1 la Figure 3.40

2 Le profil UML hybride pour UML 2.0

3 Les auteurs de l'article [?] proposent une extension d'UML pour la spécification des systèmes
 4 hybrides, où le temps de nature discrète et continue est utilisé. La construction utilise le méca-
 5 nisme du profil UML 2.0 qui est la procédure standard pour l'extension d'UML. Étant donné que
 6 les constructions de la modélisation hybride n'existent pas dans le standard d'UML, les auteurs
 7 s'appuient sur un langage de spécification appelé *CHARON*. Ce langage permet la spécification
 8 modulaire entre systèmes hybrides interactifs et, il fournit des sémantiques formelles pour le rai-
 9 sonnement sur les modèles. Les auteurs présentent également une étude de cas sur les systèmes
 10 avioniques.

Temporal Relation	Inverse Temporal Relation	Topology
T1 before (d,D) T2 d=D=0, T1 meets T2	T2 after (d,D) T1 d=D=0, T2 met_by T1	
T1 starts_before (d,D) T2 d=D=0, T1 starts T2	T2 starts_after (d,D) T1	
T1 ends_before (d,D) T2 d=D=0, T1 ends T2	T2 ends_after (d,D) T1	
T1 starts_before_end (d,D) T2 T2 ends_after_start (d,D) T1		
T1 contains ((a A)(b B)) T2 T2 contained_by ((a A)(b B)) T1 a=A=b=B=0, T1 equals T2		
T1 parallels ((a A)(b B)) T2 T2 paralleled_by ((a A)(b B)) T1		

FIGURE 3.40 – Relations temporelles basées sur les intervalles [?]

Méthodes de spécification

Concurrence et types de données : méthode de spécification

Les auteurs de l'article [?] proposent une méthode de spécification qui prend en compte la spécification des activités concurrentes et la spécification des types de données. Dans cet article la méthode est appliquée pour la spécification en *LOTOS*, mais elle peut être utilisée pour d'autres formalismes. La méthode proposée est à la fois orientée contrainte (pour la décomposition des processus vers des sous-processus en parallèle), et orientée état (pour la conception des composants séquentiels). La partie concernant les données sera obtenue à partir des informations collectées au cours du travail, l'autre partie concernant les états sera représentée par la génération des automates avec des propriétés écrites en *LOTOS*.

La méthode a des concepts similaires à ceux que nous utilisons dans notre méthode tels que la table des états, les pré- et post-conditions, etc. La Figure 3.41 présente les différentes étapes de la méthode.

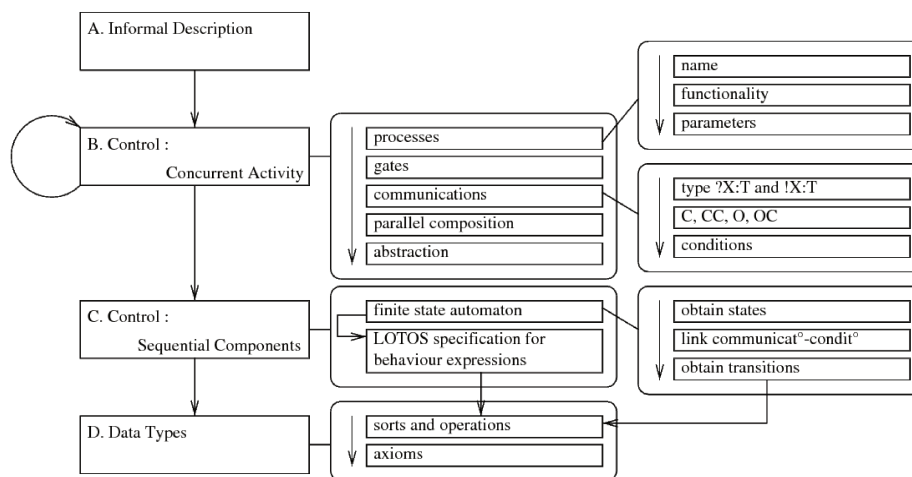


FIGURE 3.41 – La méthode pour la spécification *LOTOS*

La méthode est appliquée sur l'exemple de l'hôpital dont l'automate correspondant est montré dans la Figure 3.42.

Conception des systèmes adaptatifs avec du temps

Les auteurs de [?] proposent une approche pour modéliser les systèmes adaptatifs comme étant des logiciels avec du temps, où ces logiciels peuvent changer leurs caractéristiques lors de leurs exécutions. Les auteurs définissent un jeu d'automates temporisés, un formalisme qui combine les caractéristiques adaptatives avec un comportements temps-réel discret. Ils proposent également une nouvelle logique pour exprimer les besoins temporels vis à vis des systèmes adaptatifs, ainsi que des algorithmes pour vérifier les systèmes. Ce travail est implémenté comme étant une partie de PyECDAR, un outil de model checking.

Afin de représenter les systèmes temps réel adaptatifs, les auteurs de ce travail propose le modèle mathématique correspondant. Ce modèle inclut une représentation d'un environnement ouvert où le système interagit en temps réel. L'environnement évolue dans le temps et le système doit adapter ses comportements afin pour faire face aux différentes variations de ce dernier (c'est-

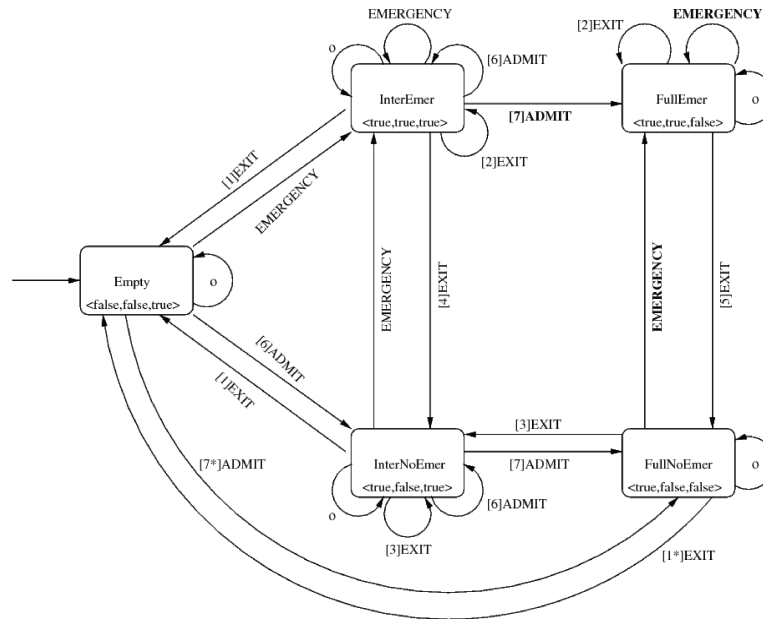


FIGURE 3.42 – L'automate de l'exemple de l'hôpital

1 à-dire l'environnement). Un exemple sur le protocole de routage est également présenté dans cet
 2 article.

3 Modélisation des systèmes temps réel en utilisant UML et des 4 annotations formelles

5 Afin de modéliser et de vérifier les systèmes temps réel les auteurs de l'article [?] présentent
 6 un travail en utilisant UML avec des notations formelles. Les auteurs s'appuient sur l'exemple du
 7 passage à niveau afin de présenter leur travail. L'approche consiste à modéliser cet exemple en
 8 utilisant les diagrammes de classes UML, ensuite pour chaque classe de ce diagramme à définir le
 9 diagramme états-transitions correspondant. L'aspect temporel est considéré dans cette approche
 10 en prenant en compte uniquement ce qu'UML propose pour la représentation des contraintes
 11 temporelles. Le langage OCL est également utilisé pour exprimer certaines propriétés telles que
 12 les invariants.

13 Les auteurs présentent également une traduction de chaque diagramme états-transitions vers
 14 *TRIO* (une logique temporelle de premier ordre) afin de pouvoir faire de la vérification de pro-
 15 priétés. Cependant, les auteurs de cet article ne précisent pas s'il y a une méthode de mo-
 16 délisation pour d'obtenir le diagramme de classes et les diagrammes états-transitions. D'autre
 17 part, ils ne précisent pas l'ensemble des éléments syntaxiques considérés dans les diagrammes
 18 états-transitions.

19 UML statecharts mobile

20 Les auteurs du travail présenté dans [?] proposent une extension des statecharts pour les
 21 calculs mobile. Les collections des objets UML étudiées sont ceux dont le comportement est
 22 représenté par les statecharts. Les auteurs présentent également une sémantique formelle (en
 23 utilisant les automates hiérarchiques) pour l'extension avec mobilité qui est basée sur une sé-
 24 mantique déjà définie pour les statecharts sans mobilité. Un exemple du service de réseau qui

- 1 exploite la mobilité pour l'équilibre de l'utilisation des ressources est présenté en utilisant l'ex-
 2 tension des statecharts. Cet exemple comporte un diagramme de classes (voir Figure 3.43) et
 3 trois diagrammes statecharts pour représenter les classes *Configurator*, *Server* et *Agent*.

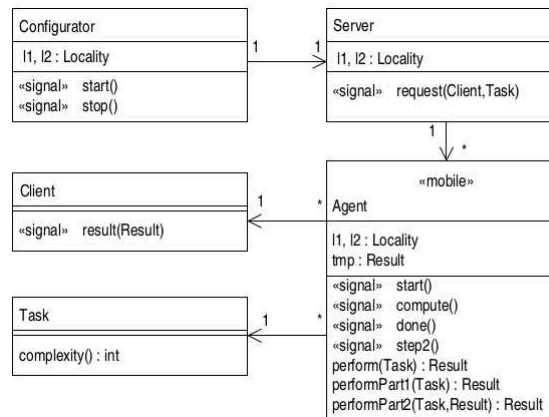


FIGURE 3.43 – Diagrammes de classes du service de réseau

4 Une approche de modélisation en utilisant les réseaux de Petri 5 colorés

- 6 Dans [?] les auteurs présentent une méthode pour spécifier les systèmes en utilisant les réseaux
 7 de Petri colorés qui s'inspire des travaux qui ont conduit à [?]. L'approche consiste à prendre
 8 en entrée une description textuelle du système et engendrée en sortie des modules de réseaux
 9 de Petri colorés. La Figure 3.44 montre les différentes étapes que la méthode adapte pour la
 10 modélisation des systèmes.

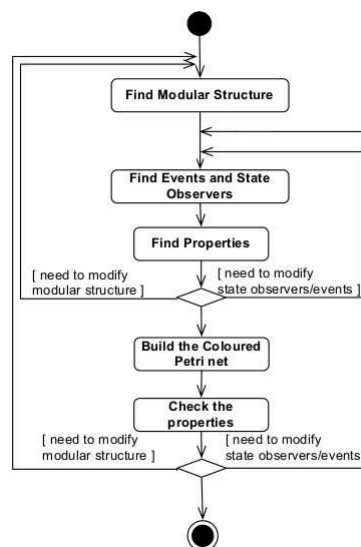


FIGURE 3.44 – Étapes de la méthode

- 11 La méthode comporte une étape d'analyse du texte afin de définir la structure modulaire
 12 ainsi que les observateurs d'état et les événements. Une analyse à base de grammaire pour
 13 une description informelle est utilisée. L'étape suivante consiste à définir les propriétés sur les

1 observateurs d'états et les pré-conditions/post-conditions sur les événements. Enfin, la dernière
 2 étape est la construction du réseaux de Petri coloré modulaire et la vérification des propriétés. Les
 3 observateurs d'état et les événements seront les places et les transitions, les pré-conditions/post-
 4 conditions seront les arcs. La méthode est appliquée sur l'exemple d'un train électrique (jouet),
 5 le résultat est donné dans la [Figure 3.45](#).

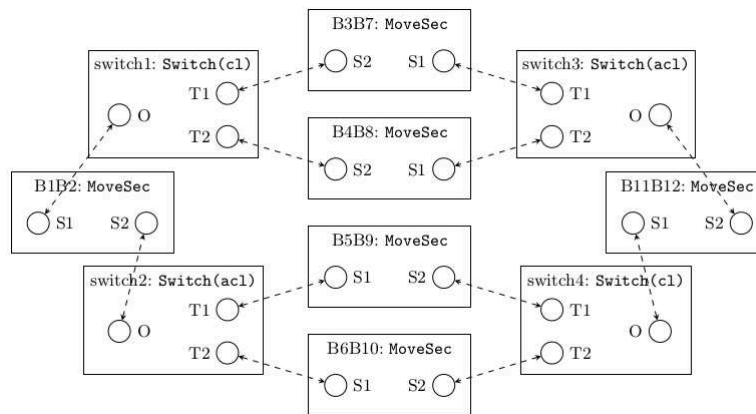


FIGURE 3.45 – Exemple du train électrique : réseau de Petri coloré

6 Conception des systèmes basés agents en utilisant UML

7 Les auteurs de l'article [?] proposent une méthodologie pour la conception des systèmes à base
 8 d'agents et qui permet également de modéliser (en utilisant les annotations UML) chaque phase
 9 du processus de développements des logiciels. Cette approche peut être gérée avec n'importe quel
 10 outil qui prend en compte UML. L'article comporte également une étude de cas d'un système
 11 d'évaluation des étudiants à base d'agents.

12 L'approche présentée dans cet article ne considère pas les multi-agents comme étant un
 13 ensemble d'agents indépendants dans un même environnement afin d'atteindre un but donné.
 14 Mais le principe considéré est un ensemble d'agents qui communiquent afin d'atteindre leurs
 15 buts mais également le but commun. Il est nécessaire de noter que les agents doivent respecter
 16 certaines règles d'organisation.

17 La méthode est basée sur une approche descendante qui analyse les besoins fonctionnels du
 18 système (considérés comme une organisation). La méthode comporte une phase de spécification
 19 des besoins, une phase d'analyse et enfin une phase de conception.

20 Les différentes étapes de la méthode sont les suivantes :

- 21 • Spécification de la fonctionnalité du système comme étant une organisation. La fonction-
 22 nalité du système est exprimée en terme de services fournis à l'utilisateur. Les services
 23 fournis à l'utilisateur peuvent être mappés sur les tâches sociales des agents membre de
 24 l'organisation.
- 25 • La conception du modèle de l'organisation est présentée en terme de modèle de rôles de
 26 règles sociales. Cela implique l'identification des positions et des fonctions qui doivent être
 27 réalisées par les agents pour leurs tâches individuelles mais aussi les règles sociales que les
 28 agents membres de l'organisation doivent suivre.
- 29 • L'identification des patterns d'interaction qui représentent les protocoles de communication
 30 entre différents rôles, et par conséquent la définition du modèle d'interaction.

- 1 • La conception du modèle de l'environnement est présentée en terme des ressources exis-
2 tantes ainsi que l'accès des protocoles à ces dernières.

3 Une traduction des statecharts vers le langage formel Promela est discutée avec une présen-
4 tation d'une étude de cas sur la gestion des étudiants dans une université virtuelle. La modéli-
5 sation de l'étude de cas est présentée sous forme de différents modules graphiques et différents
6 diagrammes UML (tels que le diagramme de séquence, le diagramme états-transitions et le dia-
7 gramme use case).

8 Conclusion

9 Nous avons présenté dans ce chapitre différents travaux de l'état l'art concernant les la
10 formalisation des diagrammes états-transitions, l'intégration des contraintes temporelles dans
11 ces diagrammes et enfin la spécification des systèmes.

12 Dans le chapitre suivant ([Chapitre 4](#)) nous présentons une méthode de spécification en uti-
13 lisant les diagrammes états-transitions et les différentes améliorations/extensions que nous pro-
14 posons.

Chapitre 4

Méthode de spécification en utilisant les diagrammes états-transitions

Contents

4.1	Introduction	64
4.2	Concepts utilisés dans la méthode de spécification	65
4.3	Méthode de spécification	66
4.4	Détails de la méthode	67
4.5	Étude de cas : exemple du paiement du parking	80
4.6	Hierarchie des états : états composites et états orthogonaux	97
4.7	Qualité du diagramme états-transitions	104
4.8	Expérimentation	109
4.9	Outil support de la méthode : <i>EasySM</i>	113
4.10	Conclusion	120

Introduction

Nous présentons dans ce chapitre une méthode de spécification permettant de spécifier/modéliser un système en utilisant les diagrammes états-transitions. Le but de la méthode est de guider l'utilisateur dans sa démarche de spécification via un ensemble d'étapes et pour spécifier/modéliser (avec les diagrammes états-transitions) le système en évitant les erreurs communes commises lors de la phase de spécification/modélisation. La méthode prend en entrée la description textuelle du système et le diagramme de classes pour engendrer le diagramme états-transitions correspondant.

Le travail présenté dans ce chapitre est une continuité et une amélioration de la méthode proposée dans [?]. Nous présentons d'abord les différents concepts utilisés dans la méthode (voir la [Section 4.2](#)) ; un aperçu de la méthode est ensuite présenté dans la [Section 4.3](#). Nous avons modifié et amélioré les différentes étapes de la méthode (voir [Section 4.4](#)) et nous l'appliquons sur une étude de cas (travail présenté dans la [Section 4.5](#)). Nous avons introduit la hiérarchie (par conséquent l'utilisation des états composites) (travail présenté dans la [Section 4.6](#)) et enfin des métriques ainsi que des facteurs de qualité (travail présenté dans la [Section 4.7](#)). Nous avons

1 également effectué une expérience afin de voir l'utilité et l'influence de la prise en compte des
2 métriques (et des facteurs de qualité) dans la compréhension du diagramme par l'utilisateur
3 (travail présenté dans la [Section 4.8](#)). Enfin, nous présentons dans la [Section 4.9](#) l'outil *EasySM*
4 qui implémente la méthode et qui permet à l'utilisateur d'automatiser certaines étapes (telles
5 que la génération des transitions).

6 Concepts utilisés dans la méthode de spécification

7 Nous présentons dans cette section les concepts que nous utilisons dans la description de la
8 méthode de spécification et des améliorations que nous apportons à cette dernière.

9 **Diagrammes de classes et description textuelle** La méthode de spécification prend
10 en entrée une description textuelle et un diagramme de classes. La description textuelle décrit le
11 système à traiter, les éléments qui le composent et ses interactions avec son environnement.

12 Le diagramme de classes regroupe les entités principales et secondaires que comporte le
13 système. Les informations regroupées dans ce diagramme correspondent à la description textuelle
14 d'entrée. Cette version sera mise à jour (avec des types de données, des associations ou d'autres
15 classes) à chaque étape de la méthode (si nécessaire) afin de la compléter. Nous définissons une
16 classe de contexte pour le diagramme de classes qui représente l'entité principale du système
17 contrôlant les interactions et les actions des autres entités.

18 Notons que c'est l'utilisateur qui doit fournir la description textuelle, le diagramme de classes
19 de départ et il doit indiquer quelle est la classe de contexte.

20 **Observateurs d'état** Afin d'observer l'état du système nous utilisons les observateurs d'état.
21 Un observateur d'état modélise une quantité (représentée par les valeurs d'un type) qui peut être
22 observée à chaque instant. Il est caractérisé par un nom, un type et une multiplicité qui permet
23 de déterminer si un observateur d'état prend tout le temps une valeur (la multiplicité est alors
24 à **One**) ou non (la multiplicité est alors à **ZeroOrOne**). En d'autres termes, cela signifie que
25 l'observateur d'état est partiellement défini si sa multiplicité est de **ZeroOrOne**. Nous définissons
26 un observateur d'état particulier (nommé *final*) qui prend la valeur *true* dans le cas où le système
27 arrête son exécution et *false* dans le cas où le système s'exécute en boucle.

28 **Événements** Les événements représentent les différentes interactions du système avec son
29 environnement mais également les interactions des différentes entités de ce dernier entre eux.
30 L'occurrence d'un événement fait changer le système d'état ; ce changement est représenté par la
31 modification des valeurs des observateurs d'état. Un événement est caractérisé par un nom et un
32 ensemble de paramètres (cet ensemble peut être vide). Nous définissons un événement particulier
33 (nommé *created*) qui permet d'établir les valeurs initiales que les observateurs d'états prendront.
34 Cet événement est similaire à l'événement (qui n'apparaît pas sur la transition) attaché à la
35 transition allant du pseudo-état initial vers un état donné.

36 **Diagramme de séquence** Nous utilisons dans cette méthode le diagramme de séquence
37 UML ; ce diagramme modélise un scénario d'exécution du système. Chaque entité présente dans
38 le diagramme de séquence est forcément une instance d'une classe présente dans le diagramme
39 de classes de départ. Cela permet de définir le séquençage d'occurrence des événements, la
40 source d'occurrence de l'événement et l'entité qui change à l'occurrence de cet événement.

41 Une autre utilité du diagramme de séquence est la validation, c'est-à-dire valider si le dia-
42 gramme états-transitions final est en cohérence avec le diagramme de séquence défini au départ.

1 Le diagramme de séquence modélise un scénario du fonctionnement du système, et le diagramme
2 états-transitions, modélise les changements d'états du système au cours de son fonctionnement.
3 S'il y a une incohérence entre les deux, elle sera visible et donc cela permet de savoir si le dia-
4 gramme états-transitions obtenu après avoir appliqué la méthode est cohérent. En effet, l'ordre
5 d'occurrence des événements doit être le même pour les deux diagrammes, car dans les deux
6 cas il y a une représentation du même scénario d'exécution. Dans le cas inverse, il n'y a pas de
7 cohérence entre les deux diagrammes.

8 Le diagramme de séquence sera mis à jour au fur à mesure de l'application de la méthode.

9 **Invariants**

10 Nous définissons pour chaque observateur d'état un invariant qui exprime une relation toujours
11 vérifiée pour les valeurs prises par cet observateur d'état. Cette relation sera exprimée en utilisant
12 les valeurs des autres observateurs d'états. Les invariants serviront pour définir les états lors de
13 la construction de la table des états du diagramme états-transitions final.

14 **Conditions/Réactions** Pour chaque événement nous définissons le couple condition/réac-
15 tion correspondant. Ce couple servira à définir les transitions entre les états dans le diagramme
16 états-transitions final. Les conditions ou les réactions sont données par des valeurs données d'un
17 ensemble d'observateurs d'état. La condition représente la situation où l'événement apparaît,
18 c'est-à-dire l'état dans lequel le système se trouve au moment de l'apparition de l'événement. La
19 condition sera utilisée pour définir l'état source (ou les états sources) de la transition étiquetée
20 par le même événement : en effet, les valeurs des observateurs d'état de la condition vont corres-
21 pondre aux valeurs des observateurs d'état d'un état ou plus. La réaction modélise le changement
22 appliqué sur le système une fois qu'il réagit à l'occurrence de l'événement. La réaction sera uti-
23 lisée pour définir l'état destination (ou les états destinations) de la transition étiquetée par le
24 même événement : en effet, les valeurs des observateurs d'état de la réaction vont correspondre
25 aux valeurs des observateurs d'état d'un état ou plus.

26 Nous définissons le couple condition/réaction de chaque événement de la même manière
27 (sauf pour l'événement *created*), cependant les valeurs pour chaque couple sont différentes d'un
28 événement à un autre. En effet, la condition de cet événement n'est pas définie car elle ne
29 correspond à aucun état du diagramme états-transitions mais seulement à un pseudo-état initial.
30 La réaction correspond à l'initialisation de tous les observateurs d'état, et donc ces valeurs
31 correspondront à l'état initial du système.

32 **Une méthode pour le développement des diagrammes** 33 **états-transitions UML [?]**

34 Nous présentons dans cette section une description générale des travaux effectués dans l'article
35 [?] qui concernent une méthode de spécification en utilisant les diagrammes états-transitions. La
36 méthode est implémentée avec l'outil *EasySM* présenté dans la [Section 4.9](#).

37 La méthode prend en entrée la description textuelle du système ainsi que le diagramme de
38 classes correspondant afin d'engendrer le diagramme états-transitions, et elle est basée sur 5
39 étapes principales.

- 40 1. La première étape consiste à faire l'analyse du texte du système à modéliser et à extraire les
41 différents éléments qui seront des candidats pour les observateurs d'état et les événements.
- 42 2. La seconde étape consiste à trouver les différents observateurs d'état ainsi que les événe-
43 ments.

- 1 3. La troisième étape complète la seconde étape car il s'agit d'exprimer les invariants caractérisant les observateurs d'état et les conditions/réactions des événements.
- 2
- 3 4. La quatrième étape consiste à engendrer les états et les transitions du diagramme états-transitions en nous basant sur les résultats trouvés dans les étapes précédentes.
- 4
- 5 5. Enfin, dans la dernière étape, le diagramme de classes de départ est mis à jour pour le
- 6 compléter avec les différents éléments ajoutés à chaque étape de la méthode.

Le schéma général des étapes de la méthode (présentée dans [?]) est fourni la [Figure 4.1](#).

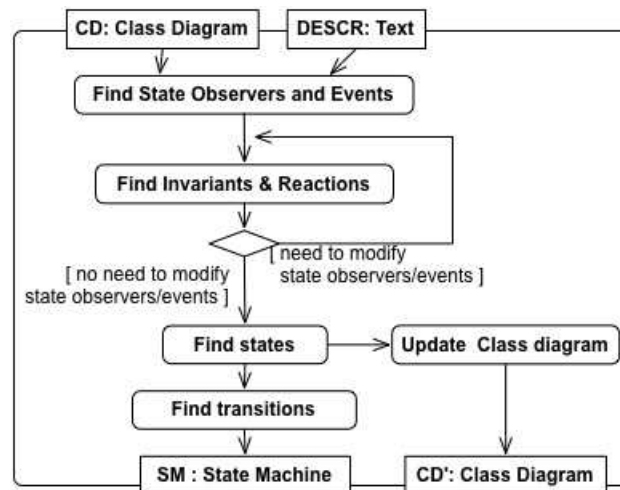


FIGURE 4.1 – Schéma des étapes de la méthode de spécification [?]

7 La méthode ne traite pas les systèmes temps-réel et nécessite des améliorations telles que
 8 l'amélioration du schéma général des étapes de la méthode, la prise en compte de la hiérarchie.
 9 Nous présentons également dans ce chapitre un ensemble de métriques, de critères de qualités
 10 et d'améliorations. Cette partie nous servira pour la validation du modèle engendré et pour
 11 l'amélioration de sa qualité. Outre les améliorations apportées à la méthode de spécification,
 12 nous avons également apporté des améliorations et des ajouts à l'outil *EasySM* qui met en
 13 œuvre la méthode ([Section 4.9](#)).

15 Détails de la méthode de spécification et améliorations

16 Ce serait lourd de fournir une description détaillée de la méthode existante [?] puis de sa
 17 nouvelle version avec les améliorations que nous proposons. Aussi dans ce qui suit, nous nous
 18 efforcerons de référencer soigneusement l'existant [?] pour le distinguer de nos apports.

19 Vue générale de la méthode

20 La méthode appliquée dans l'article [?] se base sur différentes étapes décrites sous forme d'un
 21 diagramme d'activité ([Figure 4.1](#)). Le but de la méthode est d'engendrer le diagramme états-
 22 transitions d'un système donné en suivant un ensemble d'étapes. Il est nécessaire d'avoir à la
 23 fois la description textuelle du système, ainsi que le diagramme de classes correspondant avant
 24 l'application de la méthode.

1 Afin que l'application de la méthode soit plus facile à comprendre pour l'utilisateur, nous
2 proposons des améliorations et des modifications des différentes étapes de la méthode dans ce
3 qui suit comme sous-sections. La [Figure 4.2](#) montre le nouveau diagramme d'activité des diffé-
4 rentes étapes de la méthode utilisées après nos améliorations. Ces modifications permettent à
5 l'utilisateur d'appliquer la méthode pas à pas en étant plus précis dans la spécification de son
6 système. Notons que l'étoile qui apparaît dans chaque étape (par exemple, l'étape de la détection
7 des observateurs d'état, [Figure 4.2](#)) signifie que à cette étape on met à jour le diagramme de
8 classe (la mise à jour est représentée par le carré *Update Class Diagram*). De même le symbole
9 plus (+) dénote une mise à jour du diagramme de séquence.

10 Afin que nos explications soient plus claires, nous utilisons un exemple de la gestion de
11 bibliothèque (présenté dans [?]), ensuite nous présentons un exemple plus long celui du paiement
12 de parking, voir [Section 4.5](#)) en utilisant la syntaxe utilisée dans l'outil *EasySM* (l'outil est
13 présenté dans la [Section 4.9.1](#)). Cela permettra à l'utilisateur de voir comment appliquer la
14 méthode en utilisant l'outil.

15 Analyse du texte

16 Le contenu de cette sous-section est une amélioration d'une étape existante dans la méthode
17 de [?]. Le but de cette étape est d'analyser la description textuelle du système (à modéliser) afin
18 d'extraire les informations importantes qui permettront la définition des observateurs d'état et
19 des événements. Cela permettra aussi la définition de la classe de contexte dans le diagramme
20 de classes.

21 Chaque classe du diagramme de classes représente une entité influençant le comportement
22 de notre système. Dans le diagramme de classes il y a également des "types de données" qui
23 permettent la définition des types utilisés au niveau des attributs et des opérations des différentes
24 classes.

25 En partant de cette étape, il est possible de définir la liste des événements candidats en se
26 basant sur ce qui permet de changer le système d'un état à un autre.

27 Notons quelques indications concernant la conception du diagramme de classes de départ :

- 28 • Les différentes composantes qui permettent de gérer la réception ou la génération des
29 événements, ou la réaction à ces derniers (les événements) seront modélisées sous forme de
30 classes (avec les opérations correspondantes).
- 31 • Il est possible que certains éléments du système ne soient pas considérés dans le diagramme
32 de classes car leur présence ne rajoute rien d'important dans le déroulement du système
33 (par exemple les actions internes dans la classe qui modélise le moteur dans l'exemple du
34 paiement de parking [Section 4.5](#)).

35 Rappelons que le diagramme de classes de départ est mis à jour à chaque étape de la mé-
36 thode. Cette mise à jour peut être l'ajout de types nécessaires pour les observateurs d'état ou
37 les paramètres des événements, ou bien l'ajout d'opérations dans les classes, etc. Cela permet
38 d'engendrer le diagramme de classes final au fur à mesure de l'application de la méthode.

39 Nous présentons dans ce qui suit la description textuelle de l'exemple de gestion de biblio-
40 thèque (exemple présenté dans [?]) ainsi que le diagramme de classes de départ (voir [Figure 4.3](#)).

41 **Texte** *Nous souhaitons modéliser le système de gestion d'une bibliothèque. La copie d'un livre*
42 *est insérée dans la bibliothèque ce qui permet de lui attribuer un code d'identification. Le biblio-*
43 *thécaire peut marquer en rouge une copie d'un livre ce qui signifie que le livre ne peut pas être*
44 *prêté, et il sera ainsi tant que la marque n'est pas supprimée. Bien naturellement il est impossible*

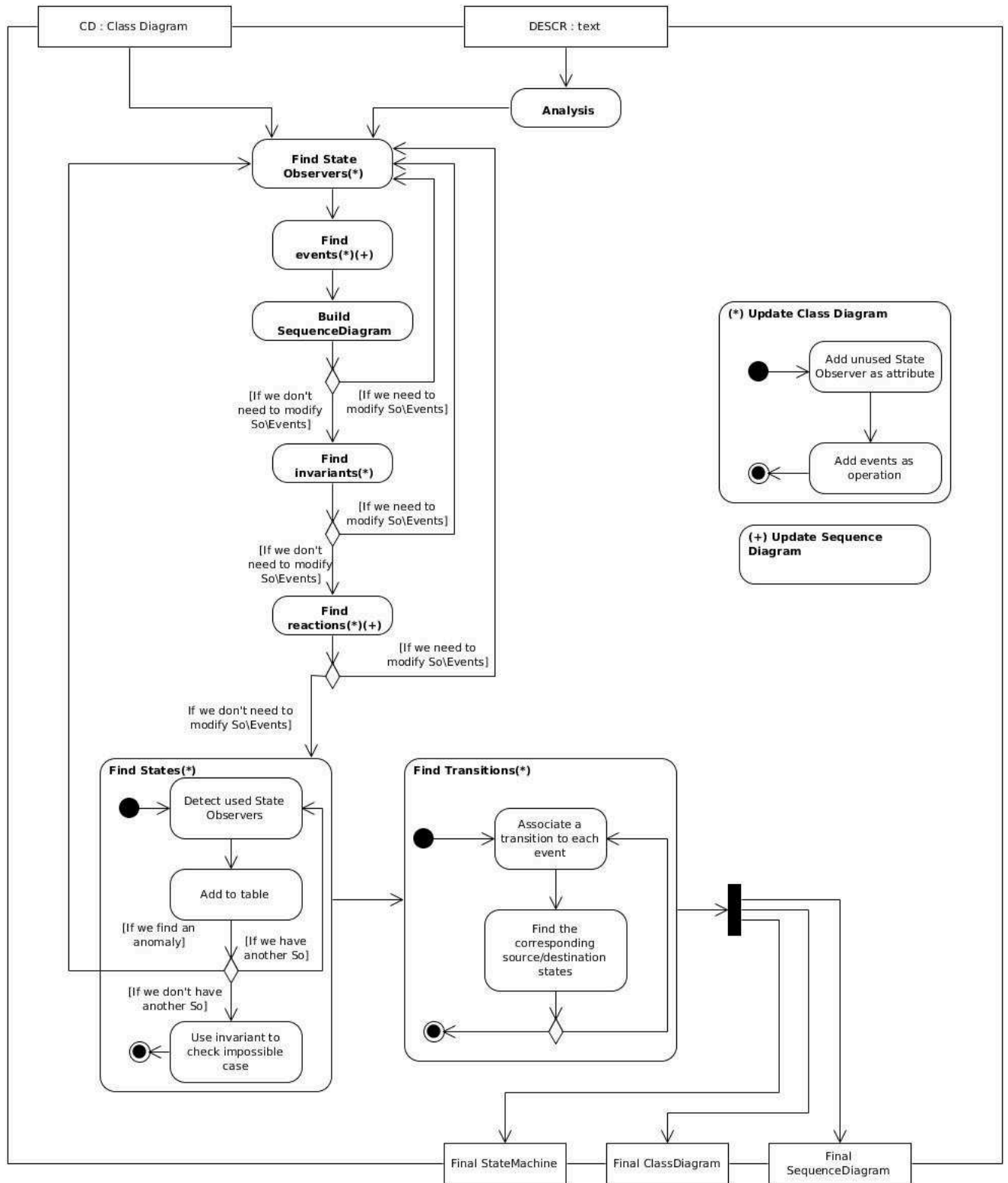


FIGURE 4.2 – Diagramme d'activité des étapes de la méthode

- 1 de mettre cette marque si le livre est déjà prêté. La copie peut être prêtée par un utilisateur de la
- 2 bibliothèque et retournée plus tard. Dans le cas où la copie est prêtée, elle peut être réservée par

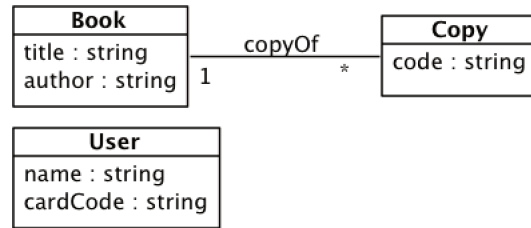


FIGURE 4.3 – Diagramme de classes de la gestion de bibliothèque

1 un autre utilisateur et il peut la récupérer une fois que le premier utilisateur la rend. Plusieurs
 2 prêts simultanés ne sont pas autorisés.

3 Recherche des observateurs d'état [?]

4 Le contenu de cette sous-section est une amélioration d'une étape existante dans la méthode
 5 de [?]. À la suite de l'analyse du texte, nous cherchons à extraire les observateurs d'état. Le
 6 [Tableau 4.1](#) montre la présentation proposée pour les observateurs d'état.

Observateur d'état	Type	Multiplicité
obs_1	type	multiplicité
...
obs_k	type	multiplicité
$final$	type	multiplicité

TABLE 4.1 – Observateurs d'état

- 7 • Nous définissons un observateur d'état particulier (nommé *final* qui prend la valeur *true*
 8 dans le cas où l'exécution du système n'est pas en boucle et peut se terminer, et prend la
 9 valeur *false* dans le cas inverse.
- 10 • Le type des valeurs d'un observateur d'état peut être un type énuméré. Le travail effectué
 11 à ce stade permet d'affiner les valeurs pertinentes et le typage.
- 12 • **Remarque sur le type d'un observateur d'état :** Si un observateur d'état $obsI$ peut
 13 prendre un ensemble infini de valeurs possibles, cet observateur ne peut être utilisé tel
 14 quel pour caractériser des états, car il y aurait une infinité d'états associés à ces valeurs.
 15 Cependant, ignorer cet observateur peut entraîner un manque d'information concernant
 16 les états. Afin de remédier à cela, nous rajoutons un nouvel observateur d'état $obsF$ (de
 17 type énuméré prenant un nombre fini de valeurs) qui permet de combler l'absence de l'ob-
 18 servateur d'état $obsI$, et qui sera utilisé pour caractériser les états. L'observateur d'état
 19 $obsF$ permet de pallier à l'absence de $obsI$. L'observateur d'état $obsI$ n'aura pas d'inva-
 20 riant dans la table des invariants et sera ajouté comme un attribut dans le diagramme de
 21 classes final. Dans le [Tableau 4.2](#), nous présentons un ensemble d'observateurs d'état avec
 22 différents types. L'observateur d'état *code* peut prendre un ensemble infini de valeurs et
 23 par conséquent il est inutilisable dans la table des états. Cependant, son absence enlève
 24 l'information à propos de la présence du livre dans la bibliothèque ; c'est pour cette raison

un autre observateur d'état est ajouté pour le remplacer : `inLib`. Le nouveau observateur d'état est de type `Boolean` (donc prend un ensemble fini de valeurs) et représente la présence du livre ou non dans la bibliothèque.

Observateur d'état	Type	Multiplicité
<i>code</i>	<i>String</i>	<i>ZeroOrOne</i>
<i>inLib</i>	<i>Boolean</i>	<i>One</i>
<i>onLoan</i>	<i>Boolean</i>	<i>One</i>
<i>booked</i>	<i>Boolean</i>	<i>One</i>
<i>booker</i>	<i>User</i>	<i>ZeroOrOne</i>
<i>tagged</i>	<i>Boolean</i>	<i>One</i>
<i>final</i>	<i>Boolean</i>	<i>One</i>

TABLE 4.2 – Gestion de la bibliothèque[?] : observateurs d'état

Le travail de typage lors de la création des événements et des observateurs d'état peut conduire à mettre à jour le diagramme de classes de départ. Notez que dans le cas où il est nécessaire d'avoir un nouveau type pour les observateurs d'état, il suffit de définir le type souhaité et de l'ajouter ensuite dans le diagramme de classes (la classe aura ses attributs et ses opérations).

Recherche des événements [?]

Le contenu de cette sous-section est une étape existante dans la méthode de [?]. Un événement peut avoir un argument avec son type qui sera indiqué (par exemple, *event1 (type)*). Le [Tableau 4.3](#) montre la présentation proposée des événements.

Événement
<i>event₁</i>
...
<i>event_k(type)</i>
<i>created</i>

TABLE 4.3 – Événements

L'occurrence d'un événement permet au système de changer d'état ; ce changement est représenté par la modification des valeurs des observateurs d'état. Par conséquent, il est possible de définir une correspondance entre les événements et les observateurs d'état. Chaque observateur d'état change de valeur à l'occurrence d'un événement (ou de plusieurs). Par exemple, il y a une correspondance entre l'observateur d'état `inLib` (dans le [Tableau 4.2](#)) et l'événement `insertLib` (dans le [Tableau 4.4](#)). L'occurrence de l'événement `insertLib` change la valeur de l'observateur d'état `inLib` (et c'est ce qui fait la correspondance entre les deux).

Notons que dans le [Tableau 4.3](#), il y a un événement particulier (nommé *created*). Cet événement permet d'établir les valeurs initiales que les observateurs d'états prendront (voir la [Section 4.4.7](#)).

Le [Tableau 4.4](#) représente les événements de l'exemple de la gestion de la bibliothèque.

Événement
<i>insertLib()</i>
<i>lend(User)</i>
<i>bringBack()</i>
<i>book()</i>
<i>putTag()</i>
<i>removeTag()</i>
<i>created()</i>

TABLE 4.4 – Gestion de la bibliothèque[?] : événements

1 Construction du diagramme de séquence

2 Le contenu de cette sous-section est une nouvelle étape qui n'existe pas dans la méthode de [?].
 3 Afin de construire le diagramme de séquence, nous allons utiliser, le diagramme de classes, les
 4 événements et les observateurs d'états. Certaines classes (il n'y a pas la prise en compte des types
 5 de données ou des classes abstraites et dans le cas d'un système complexe il est impossible de
 6 représenter toutes les classes) du diagrammes de classes sont instanciées dans le diagramme de
 7 séquence, avec la présence d'un acteur principal (par exemple l'utilisateur du système). Les in-
 8 teractions entre les différentes instances des classes se font à travers l'occurrence des événements.
 9 Chaque événement aura une source de son occurrence (c'est-à-dire l'instance qui l'engendre)
 10 et une destination qu'il va affecter (c'est-à-dire l'instance qui sera affectée par l'occurrence de
 11 l'événement).

12 Nous présentons dans la Figure 4.4 le format du diagramme de séquence construit lors de
 13 l'application de la méthode, où :

- 14 • `ClassInstance_i` représente l'instance d'une classe donnée
- 15 • `eventi` représente un événement faisant partie de l'ensemble des événements extraits au
 16 cours des étapes précédentes.
- 17 • `obs` représente un observateur d'état faisant partie de l'ensemble des observateurs d'état
 18 extraits au cours des étapes précédentes. Notons que la valeur de l'observateur d'état
 19 permet de conditionner l'occurrence d'événements ; c'est-à-dire dans le cas où la valeur est
 20 à *true* alors il y a l'occurrence de l'événement `event3` et dans le cas où c'est à *false* alors
 21 il y a l'occurrence de l'événement `event2`.

22 Recherche des invariants des observateurs d'état [?]

23 Le contenu de cette sous-section est une amélioration d'une étape existante dans la méthode
 24 de [?]. Afin d'extraire les invariants des observateurs d'état, nous allons utiliser la description
 25 textuelle du système mais surtout le diagramme de séquence défini dans l'étape précédente. Le
 26 Tableau 4.5 montre la présentation des observateurs d'état avec leur invariants.

27 Afin de déterminer l'invariant d'un observateur d'état `obs`, il est nécessaire de trouver les
 28 relations que cet observateur a avec les autres observateurs d'états. Connaitre ces relations va
 29 nous permettre de déterminer quel observateur d'état sera utilisé pour la définition de l'invariant
 30 d'`obs`.

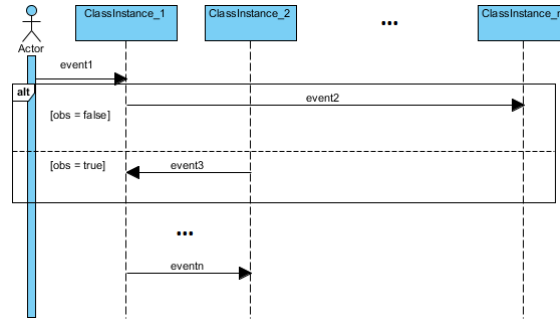


FIGURE 4.4 – Diagramme de séquence

Observateur d'état	Invariant
obs_1	inv_1
...	...
obs_k	inv_k
$final$	inv_{final}

TABLE 4.5 – Invariants des observateurs d'état

Notons que pour connaître les valeurs adéquates des observateurs d'état, il suffit de voir le séquençage modélisé dans le diagramme de séquence. L'occurrence d'un événement permet de modifier la valeur d'un ou de plusieurs observateurs d'état. Par conséquent, connaître l'ordre d'occurrence des événements nous permet de savoir à quel moment les valeurs des observateurs d'états sont modifiées et donc définir l'invariant.

Remarques

- Un observateur d'état peut avoir plusieurs invariants, dans le cas où par exemple il a un type énumératif (c'est-à-dire il peut prendre plusieurs valeurs).
- La recherche des invariants peut s'avérer délicate et source de diverses erreurs. Or ces invariants seront utilisés pour caractériser les états du système (à l'aide d'une combinaison de valeurs des observateurs d'état). Il est crucial que ces invariants soient correctement exprimés. Nous essayons ci-après de noter des points méthodologiques pour aider à une juste expression des invariants.
 - Une manière de chercher les invariants associés à un observateur d'état obs est de chercher les conditions remplies pour qu'il prenne une valeur donnée.
 - Ces conditions font intervenir les autres observateurs d'état, et il faut être vigilant à (i) ne pas en oublier (ii) ne pas « en mettre trop » (certains en effet peuvent être indépendants).
 - Pour certaines valeurs, on ne peut obtenir des conditions sur plusieurs autres observateurs d'état, et on ne peut donc établir un invariant pertinent pour caractériser un état. Dans ce cas ces valeurs ne sont pas retenues.

1 Le [Tableau 4.6](#) représente les invariants des observateurs d'état pour l'exemple de la gestion
 2 de la bibliothèque. Chaque ligne du tableau représente un observateur d'état avec son invariant
 correspondant.

Observateur d'état	Invariant
inLib	not inLib implies (not onLoan and not booked and booker=under and not tagged)
onLoan	onLoan implies not tagged and inLib
booked	booked implies (not tagged)
tagged	tagged implies (not onLoan and not booked)
final	not final

TABLE 4.6 – Gestion de la bibliothèque[?] : invariants

3

4 Recherche des conditions / réactions des événements [?]

5 Le contenu de cette sous-section est une étape existante dans la méthode de [?] à laquelle nous
 6 apportons quelques remarques (précisions). Le [Tableau 4.7](#) montre la présentation des événements
 7 avec leurs conditions / réactions.

Événement	Condition	Réaction
<i>event₁</i>	<i>condition₁</i>	<i>reaction₁</i>
...
<i>event_k</i>	<i>condition_k</i>	<i>reaction_k</i>
<i>created</i>	/	<i>Initialisation</i>

TABLE 4.7 – Conditions/réactions des événements

8 Remarques

9

- 10 • Dans certains cas il est nécessaire de réinitialiser les valeurs des observateurs d'état (cela
 11 représente généralement l'état initial du système à l'absence d'un état final par exemple).
 12 Par conséquent, il faut chercher le moment pertinent pour le faire (c'est-à-dire dans la
 13 partie réaction de quel événement il faut faire cela).
- 14 • Dans le cas où il est nécessaire d'utiliser dans la partie réaction des événements les opéra-
 15 tions de différentes classes, alors il est impératif d'établir des associations entre ces classes
 16 et la classe de contexte. Cette modification impliquera de mettre à jour le diagramme de
 17 classes avec les différentes associations établies. Les associations sont également essentielles
 18 pour le bon fonctionnement de l'outil *EasySM* qui implémente la méthode de spécification.

- Dans le cas où il y a un observateur d'état avec des valeurs infinies dans l'ensemble des observateurs d'état représentant la condition d'un événement, alors il sera la garde de la transition étiquetée par cet événement. Dans le cas où il y a un observateur d'état avec des valeurs infinies dans l'ensemble des observateurs d'état représentant la réaction d'un événement, alors il sera l'action de la transition étiquetée par cet événement.

Dans le [Tableau 4.8](#) nous présentons l'application de la méthode sur l'exemple de la gestion d'une bibliothèque. Chaque ligne représente la condition et la réaction de chaque événement trouvé précédemment.

Événement	Condition	Réaction
created	true	tagged := false ; inLib := false ; final := false
inLib()	not inLib	inLib := true
lend(U : user)	-inLib and not onLoan and not tagged and not booked -inLib and not onLoan and not tagged and booked and booker=U	-onloan := true -onLoan := true ; booked := false ; booker := undef
bringBack()	inLib and onLoan	onLoan := false
book(U : User)	inLib and onLoan and not booked	booked := true ; booker := U
putTag()	inLib and not onLoan and not tagged and not booked	tagged := true
removeTag()	inLib and tagged	tagged := false

TABLE 4.8 – Gestion de la bibliothèque [?] : conditions/réactions des événements

Construction de la table des états [?]

Le contenu de cette sous-section est une étape existante dans la méthode de [?]. Nous terminons dans les phases précédentes les observateurs d'état (avec leurs invariants) et les événements (avec leurs conditions / réactions) afin de les utiliser pour engendrer le diagramme états-transitions final. Dans cette sous-section nous définissons les états ([Tableau 4.9](#)) en utilisant les observateurs d'états mais également les conditions / réactions des événements.

Seuls les observateurs d'états ayant un nombre fini de valeurs sont utilisés pour définir les états. Chaque ligne (représentée par un ensemble de valeurs des observateurs d'état) peut correspondre à un état. Il y a des états impossibles, c'est-à-dire des situations impossibles, dans le cas où la ligne est incohérente.

Afin de savoir si une ligne correspond à une situation possible, à un état donc, il suffit de vérifier si la ligne (une combinaison de valeurs) n'entre pas en contradiction avec les invariants des observateurs d'état. Dans le cas où la vérification se fait correctement alors la ligne est cohérente et donc elle correspond à un état que le système peut prendre, dans le cas inverse l'état est considéré impossible. Notons également qu'il est possible de vérifier les valeurs de la ligne d'un état avec les conditions/réactions des événements. En effet, les valeurs de cette ligne correspondent à une condition ou à une réaction d'un événement (ou plusieurs, et dans ce cas il faut choisir l'événement correspondant).

obs₁	...	obs_i	État
<i>valeur₁</i>	...	<i>valeur₁</i>	<i>nom₁</i>
...
<i>valeur_k</i>	...	<i>valeur_k</i>	<i>nom_k</i>
<i>valeur_m</i>	...	<i>valeur_m</i>	<i>impossible</i>
...
<i>valeur_n</i>	...	<i>valeur_n</i>	<i>impossible</i>

TABLE 4.9 – États

1 Le [Tableau 4.10](#) montre les différents états trouvés lors du traitement de l'exemple de la
 2 gestion de bibliothèque. Nous remarquerons qu'il y a un ensemble d'état possibles et un en-
 3 semble d'états impossibles (dont la lignes entre en contradiction avec l'un des invariants trouvés
 4 précédemment).

inLib	onLoan	booked	tagged	État
true	true	true	false	OnloanBooked
true	true	false	false	OnLoanNotBooked
true	false	true	false	Booked
true	false	false	true	Tagged
true	false	false	false	Available
false	false	false	false	NotInLibrary
false	false	false	true	impossible
...	impossible

TABLE 4.10 – Gestion de la bibliothèque [?] : table des états

5 Recherche des transitions [?]

6 Le contenu de cette sous-section est une étape existante dans la méthode de [?]. Après
 7 l'identification des états, nous allons voir dans ce qui suit comment trouver les transitions de
 8 notre diagramme états-transitions final en utilisant les événements. En effet, chaque événement
 9 est un candidat pour étiqueter une nouvelle transition dans le diagramme états-transitions. Une
 10 première étape sera de définir une représentation d'une transition avec l'événement en question
 11 (voir [Figure 4.5](#)).

12 Une seconde étape sera de chercher les états source et cible de chaque transition en utilisant la
 13 condition / réaction de l'événement et la table des états. La condition de l'événement détermine
 14 une combinaison de valeurs possibles pour les observateurs d'état et il en est de même pour la
 15 réaction à l'événement. Une recherche dans la table des états permet de déterminer les états qui
 16 satisfont ces conditions. En effet, chaque ligne de la table représente une combinaison de valeurs
 17 que les observateurs d'état prennent. Cet ensemble peut correspondre à l'ensemble de valeurs
 18 données pour les observateurs d'état qui représente la condition (ou réaction) d'un événement.
 19 Dans le cas où les valeurs sont les mêmes alors cela nous permet de définir l'état source de la

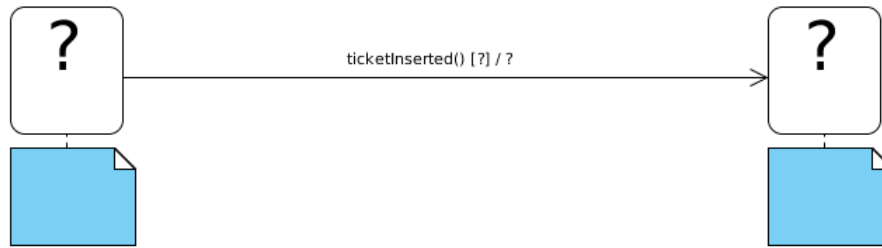


FIGURE 4.5 – Modèle de recherche d'une transition

1 transition (l'état destination) Cependant, il est possible d'avoir des anomalies lors de l'attribution
 2 des états aux transitions.

- 3 • Dans le cas où il n'y a aucun état qui correspond à la condition ou à la réaction d'une
 4 transition. Cela est dû soit à une erreur dans la définition des invariants des observateurs
 5 d'états, soit à la définition de la condition / réaction de la transition.
- 6 • Dans le cas où il y a plusieurs états qui correspondent à la condition ou à la réaction d'une
 7 transition. Cela est dû à une erreur de définition des invariants des observateurs d'états.

8 Dans le cas où il y a une des anomalies lors de l'application de la méthode soit il faut
 9 vérifier l'une des étapes précédentes (par exemple les conditions/réactions des événements ou les
 10 invariants des observateurs d'états), soit il faut considérer que la modélisation du système avec
 11 les données présentées en entrée n'est pas faisable.

12 Tout observateur d'état qui est un attribut dans le diagramme de classes et qui apparaît
 13 dans la condition de l'événement, doit apparaître dans une garde pour la transition résultante.
 14 Tout observateur d'état qui est un attribut dans le diagramme de classes et qui apparaît dans la
 15 réaction de l'événement, doit apparaître dans une action pour la transition résultante.

16 Génération du diagramme états-transitions [?]

17 Le contenu de cette sous-section est une étape existante dans la méthode de [?]. Une fois
 18 faite la définition des états et des transitions, la dernière étape de la méthode est d'engendrer le
 19 diagramme états-transitions. Les états sont liés entre eux avec les différentes transitions trouvées
 20 dans la [Section 4.4.9](#).

21 Nous ajoutons un pseudo-état initial qui sera lié par une transition sans événement, avec l'état
 22 initial du système. La transition est censée avoir l'événement *created*, cependant dans UML cet
 23 événement est un événement interne et donc n'est pas représenté dans la transition. L'état cible
 24 de cette transition est l'état ayant les mêmes valeurs d'observateurs d'état que la réaction de
 25 l'événement *created*.

26 Dans le cas où le système se termine, alors le diagramme états-transitions contiendra un état
 27 final (ou des états finaux, dans le cas où il y a plusieurs états finaux) qui sera lié avec l'état
 28 correspondant et avec la transition correspondante. Cela sera indiqué depuis le départ par la
 29 valeur de l'observateur d'état *final* qui sera égale à *true*.

30 Vérification de la cohérence

31 Le contenu de cette sous-section est une nouvelle étape qui n'existe pas dans la méthode de
 32 [?]. Comme précisé précédemment, l'utilisation du diagramme de séquence a pour but d'aider à
 33 extraire les observateurs et les événements mais également de valider la cohérence du résultat

1 final (c'est-à-dire le diagramme états-transitions). Nous proposons trois phases afin de valider
2 cette cohérence comme suit :

- 3 • La première phase consiste à vérifier (d'une manière informelle) les invariants des observa-
4 teurs d'état. En effet, il est possible de vérifier si les valeurs des observateurs d'état utilisées
5 dans l'invariant sont les bonnes et cela en nous basant sur la séquence de l'occurrence
6 des événements dans le diagramme de séquence. À chaque occurrence d'un événement il
7 y a généralement un changement d'état du système et par conséquent un changement de
8 valeur de certains observateurs d'état.
- 9 • La seconde phase est la vérification des conditions/réactions de chaque événement. Pour
10 chaque occurrence d'un événement il est possible de vérifier sa condition (réaction) en utili-
11 sant le diagramme de séquence. La condition modélise la situation dans laquelle se trouvait
12 le système avant l'occurrence de l'événement. Par conséquent, dans le diagramme de sé-
13 quence, cela est modélisé par les différents événements qui sont apparus avant l'événement
14 en question, mais aussi par la valeur des observateurs d'état.
- 15 • La troisième phase consiste à vérifier si le scénario que le diagramme de séquence repré-
16 sente est le même que celui du diagramme états-transitions. Cela est pertinent car les
17 deux diagrammes tracent les changements d'états et l'occurrence des événements lors du
18 fonctionnement de notre système. Si l'application de la méthode est correcte et si les ré-
19 sultats des différentes étapes sont cohérents, alors les deux diagrammes seront identiques
20 du point de vue du fonctionnement du système. Cette vérification est faite manuellement
21 (sans l'utilisation d'outil).

22 Améliorations apportées à la méthode de spécification

23 Nous présentons dans cette sous-section le résumé des différentes améliorations que nous
24 apportons à la méthode de spécification [?]. Les améliorations portent sur les points suivants
25 (voir [Figure 4.2](#)) :

- 26 • Nous intégrons dans un premier temps une phase d'analyse (basée sur le texte fourni
27 par l'utilisateur) pour préciser le déroulement du fonctionnement du système mais aussi
28 détecter les observateurs d'état / événements candidats.
- 29 • Nous séparons l'identification des observateurs de celle des événements afin d'avoir deux
30 phases où chaque partie est traitée en profondeur.
- 31 • Nous ajoutons un diagramme de séquence (basé sur les observateurs d'état et les événe-
32 ments identifiés mais surtout sur l'analyse du texte) pour deux raisons : la première est
33 qu'avoir un séquençement du fonctionnement du système peut aider à exprimer par la
34 suite les invariants des observateurs d'état et les conditions / réactions des événements.
35 La seconde raison est de pouvoir vérifier la cohérence entre le diagramme états-transitions
36 obtenu par la méthode et le diagramme de séquence de départ.
- 37 • Nous séparons également la définition des invariants et les conditions / réactions en deux
38 phases, cela pour à la fois approfondir le traitement de chaque partie mais également pour
39 avoir la possibilité de revenir en arrière s'il y a une erreur ou une incohérence.
- 40 • Enfin, nous détaillons la manière de définir les états, les transitions et de faire la mise à
41 jour du diagramme de classes qui apparaît à chaque étape. Afin de définir les états nous
42 commençons par la sélection et l'ajout des observateurs d'état utilisés dans l'identification

1 des états, ensuite nous vérifions les invariants pour détecter les états impossibles. À chaque
2 événement, nous associons une transition (avec un état source et un état cible) de telle
3 manière que les états source et cible de chaque transition soient conformes aux conditions
4 et réactions de l'événement. Concernant les transitions, nous dessinons les événements sous
5 forme de transition (avec un état source et un état cible) et selon les conditions réactions
6 nous définissons les états sources et cibles correspondants.

Étude de cas : exemple du paiement du parking

Dans cette section nous présentons la modélisation d'un exemple de paiement de parking. Dans un premier temps, nous avons traité cet exemple pour acquérir une compréhension approfondie de la méthode présentée dans [?]. Dans les faits, cela a entraîné de nombreuses discussions avec les auteurs, et cette étude de cas nous conduit à effectuer une analyse critique de la méthode et à proposer des précisions et des améliorations présentées dans la section précédente. Nous avons ainsi fait plusieurs essais avant d'arriver à une version de la modélisation qui nous semble satisfaisante et qui aille de pair avec les précisions et les améliorations proposées. Le texte de départ est le suivant :

*Nous souhaitons réaliser un système de paiement de parking. Le système comporte une barrière qui se lève pour laisser passer des voitures, un **lecteur de ticket et de carte bancaire**, un **détecteur de présence de voitures**, un **moteur de barrière**, et un **détecteur de position de la barrière** :*

- *L'utilisateur souhaite sortir du parking. Pour cela, il insère son ticket de parking dans le lecteur de la borne qui le lit. Si la lecture échoue, le ticket est restitué à l'utilisateur.*
- *Si la lecture réussit, le système attend que l'utilisateur insère sa carte bancaire dans le lecteur de carte pour effectuer le paiement du parking. Le système lit la carte, et contacte une banque pour réaliser le paiement du parking, puis restitue la carte de l'utilisateur. L'utilisateur passe à l'étape suivante si le paiement a réussi (sinon, il revient à état où le système attend l'insertion d'un ticket).*
- *Le système envoie une commande au moteur de la barrière pour lever cette dernière. Dès que le détecteur de position de la barrière indique que celle-ci est en position haute, le système envoie un signal d'arrêt au moteur.*
- *Le système attend que le détecteur de présence de véhicules signale qu'il n'y a plus de voitures pour envoyer au moteur un signal de descente. Le système envoie au moteur un signal d'arrêt dès que le détecteur de position de la barrière indique que celle-ci est en position basse.*

Analyse du texte et diagramme de classes

Lors de notre travail d'analyse, nous avons remarqué que, dans le texte ci-dessus, le système agit comme un contrôleur des différentes entités en présence (**lecteur de ticket et de carte bancaire**, **détecteur de présence de voitures**, **moteur de barrière**, **détecteur de position de la barrière**), nous utiliserons donc le terme **contrôleur**. L'analyse de la version textuelle (informelle) de l'exemple donne la description suivante (la [Figure 4.8](#) montre le diagramme de séquence correspondant à la description) :

1. Détection de l'arrivée d'une voiture à la barrière du parking (détecteur de présence de voitures)
2. L'utilisateur insère son ticket dans le lecteur de ticket
3. Si la lecture échoue : le lecteur de ticket informe le contrôleur de l'échec de la lecture et le contrôleur lui ordonne d'éjecter le ticket.
4. Si la lecture réussit : le contrôleur est informé de la réussite de la lecture par le lecteur de ticket alors le contrôleur ordonne au lecteur de carte bancaire de donner la main à l'utilisateur pour payer.

- 1 5. L'utilisateur insère sa carte bancaire : le contrôleur est informé de l'insertion de la carte
- 2 bancaire par le lecteur de carte bancaire qui contacte la banque pour la vérification du
- 3 paiement.
- 4 6. Si le paiement échoue alors la banque informe le contrôleur de l'échec du paiement qui
- 5 ordonne au lecteur de carte bancaire d'éjecter la carte et l'utilisateur revient à l'état où il
- 6 insert du ticket.
- 7 7. Si le paiement réussit alors la banque informe le contrôleur de la validité du paiement, ce
- 8 dernier ordonne au lecteur de carte bancaire d'éjecter la carte bancaire et ordonne la mise
- 9 en marche du moteur de la barrière.
- 10 8. Le détecteur de barrière informe le contrôleur que la barrière est en position haute alors
- 11 le contrôleur ordonne l'arrêt du moteur de la barrière.
- 12 9. Le détecteur de présence de voiture informe le contrôleur du départ de la voiture alors le
- 13 contrôleur ordonne la mise en marche du moteur de la barrière pour la descente.
- 14 10. Le détecteur de barrière informe le contrôleur que la barrière est en position basse alors le
- 15 contrôleur ordonne l'arrêt du moteur de la barrière.
- 16 11. Revenir à l'état où une voiture arrive à la barrière.

17 La Figure 4.6 présente le diagramme de classes et le travail va porter sur la spécification du
 18 comportement de la classe contrôleur choisie comme classe de contexte. Comme indiqué dans la
 19 Section 4.4.2, le diagramme de classes est mis à jour dans les différentes étapes de la méthode (à
 20 l'ajout des types pour les observateurs d'état, des associations pour les conditions/réactions des
 21 événements, etc). Nous utilisons une classe de type «datatype» (*CreditCard*) afin de modéliser
 22 les valeurs décrivant une carte bancaire.

23 Nous définissons la liste des événements candidats en nous basant sur la notion de réception
 24 de messages décrite dans la Section 4.4.2.

25 Notons que les étapes (3-10) de l'analyse du texte correspondent à l'interaction avec le contrô-
 26 leur. Le premier point de l'analyse du texte est implicite (la détection de présence de voitures)
 27 dans le texte et nous a paru nécessaire notamment lorsque nous analysons la sortie des véhicules
 28 (la classe *carSensor*).

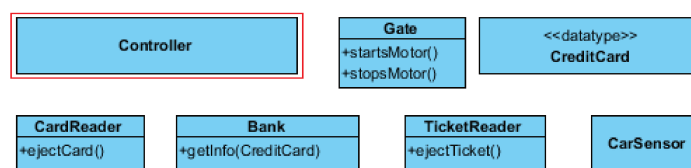


FIGURE 4.6 – Diagramme de classes de l'exemple du paiement du parking

29 Observateurs d'état et événements

30 La phase suivante consiste à extraire les observateurs d'état et les événements tout en mettant
 31 à jour le diagramme de classes.

32 Les événements candidats ainsi que les observateurs d'état candidats sont présentés dans le
 33 Tableau 4.11. Les événements comportent éventuellement un argument dont le type est indiqué
 34 et les observateurs d'état sont donnés avec le type de leur résultat et leur multiplicités.

Événement	Observateur d'état	Type	Multiplicité
ticketInserted ()	ticketActive	Bool	One
ticketFailure ()	ticketReadOk	BoolS	One
ticketSuccess ()	cardInfo	CreditCard	ZeroOrOne
card_Inserted (CreditCard)	cardKnown	Bool	One
receivePaymentOk (CreditCard)	paymentOk	BoolS	One
receivePaymentFail (CreditCard)	gateState	GateS	One
gateUp ()	carAtGate	Bool	One
gateDown ()			
carArrives ()			
carLeaves ()			

TABLE 4.11 – Événements et observateurs d'état

- Nous avons décrit dans la [Section 4.4.3](#) la notion de multiplicité qui sert à préciser si un observateur d'état peut ou non avoir tout le temps une valeur. Dans le [Tableau 4.11](#) tous les observateurs d'état ont une multiplicité *One* (c'est-à-dire ils ont tout le temps une valeur) sauf l'observateur d'état `cardInfo` qui a une multiplicité *ZeroOrOne*. En effet, l'observateur d'état `cardInfo` a une valeur uniquement si la carte bancaire est insérée dans le lecteur de carte bancaire, autrement il n'a pas de valeur (sa valeur n'est pas définie).
- L'observateur d'état `cardInfo` peut avoir un ensemble infini de valeurs et par conséquent nous ne pouvons pas l'utiliser pour définir les états car il y aurait un ensemble infini d'états (voir explication dans la [Section 4.4.3](#)). Cependant, ignorer cet observateur d'état entraîne un manque d'information et pour cette raison nous proposons un nouvel observateur d'état `cardKnown` ayant un type fini. L'observateur d'état `cardKnown` remplacera `cardInfo` dans la table des états et aura un invariant dans la table des invariants. L'observateur d'état `cardInfo` n'aura donc pas d'invariant dans la table des invariants et sera ajouté comme attribut dans le diagramme de classes final.
- Le type des valeurs d'un observateur d'état peut être un type énuméré, par exemple le type *GateS* permet de définir les deux états de la barrière : *up* (quand la barrière est en haut) et *down* (quand la barrière est en bas). Nous avons également le type **BoolS** qui ressemble au type **Bool** avec la différence que ces valeurs sont *trueS* (c'est l'équivalent de *true*) et *naS* (qui veut dire *not available*, c'est-à-dire non disponible). Le diagramme de classes (voir [Figure 4.7](#)) est mise à jour avec ces types. Par exemple, pour `ticketReadOk` et `paymentOk`, nous avons d'abord choisi le type **Bool** mais constaté que l'information n'est pas toujours disponible. Une opération partielle ne répondait pas non plus au problème. Finalement nous avons opté pour le type **BoolS** avec deux valeurs *trueS* et *naS* (*not available*). Une remarque sur le choix des noms pour *true* et *na* : *true* est un mot réservé pour le type **Bool** donc nous avons changé en *trueS* et *na* devient *naS* par homogénéité. La valeur *naS* (*not available*) signifie « non disponible », comme par exemple l'observateur d'état `ticketReadOk` qui prend la valeur *naS* dans le cas d'absence de ticket.

La [Figure 4.7](#) montre le nouveau diagramme de classe avec l'ajout de deux classes de types énumérés *BoolS* et *GateS*. L'outil *EasySM* permet ce type d'ajout et met à jour systématiquement la liste des types proposés lors de la création des observateurs d'état. Les types Boolean, Integer, Set, etc (présents dans les types OCL) sont prédéfinis dans l'outil.

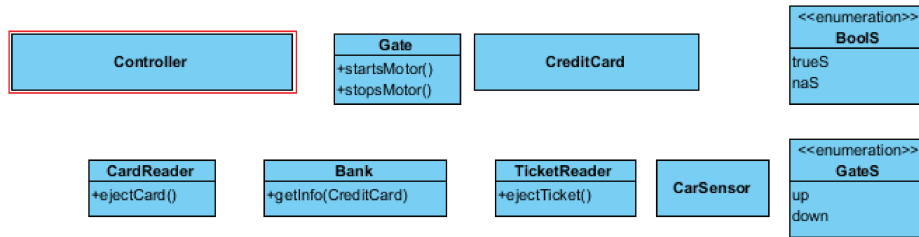


FIGURE 4.7 – Diagramme de classe de l'exemple du paiement du parking : mise à jour des nouvelles classes

- 1 Afin d'éclaircir l'usage des événements et des observateurs d'état, nous présentons dans ce
 2 qui suit l'explication de chaque événement/observateur d'état utilisé :
- 3 • `ticketInserted()` : l'événement modélise l'insertion du ticket (l'observateur d'état `ticketActive`
 4 reflète le fait qu'un ticket a été inséré et est en cours de traitement).
 - 5 • `ticketFailure()` : l'événement modélise l'échec de la lecture du ticket.
 - 6 • `ticketSuccess()` : l'événement modélise le succès de la lecture du ticket (la valeur `true`
 7 pour l'observateur d'état `ticketReadOk` reflète que la lecture a été bien effectuée).
 - 8 • `card_Inserted(CreditCard)` : l'événement modélise l'insertion de la carte bancaire (l'ob-
 9 servateur d'état `cardInfo` fournit les informations sur la carte bancaire).
 - 10 • `receivePaymentFail(CreditCard)` : l'événement modélise l'échec du paiement.
 - 11 • `receivePaymentOk(CreditCard)` : l'événement modélise la réussite du paiement (la valeur
 12 `trueS` pour l'observateur d'état `paymentOk` indique que ce paiement a bien été effectué).
 - 13 • `gateUp()` : l'événement modélise la détection de la position de la barrière en haut.
 - 14 • `gateDown()` : l'événement modélise la détection de la position de la barrière en bas.
 - 15 • `carArrives()` : l'événement modélise la détection de l'arrivée d'une voiture (qui est ob-
 16 servée par `carAtGate`).
 - 17 • `carLeaves()` : l'événement modélise la détection du départ d'une voiture.

18 Pendant l'analyse du texte ainsi que l'extraction des différents observateurs d'état/événements,
 19 l'observateur d'état `motorOn` a été écarté car l'état du moteur de la barrière n'a aucune
 20 influence sur le déroulement du système (c'est un élément interne). La modélisation est ici faite
 21 grâce à des opérations dans le diagramme de classes (pour la classe `Gate`) : `startsMotor()` et
 22 `stopsMotor()`.

23 Les noms choisis pour les différents événements et opérations sont utilisés dans le diagramme
 24 de séquence [Figure 4.8](#).

25 Ce diagramme permet de montrer le scénario que le système suit en utilisant les différentes
 26 valeurs des observateurs d'état ainsi que l'occurrence des événements. Le diagramme peut être
 27 aussi utilisé comme un diagramme de validation en le comparant au diagramme états-transitions
 28 final. Cela permet de voir si les deux diagrammes modélisent le même déroulement (scénario) et
 29 ainsi voir la cohérence du résultat (voir la dernière partie de la [Section 4.5.4](#)).

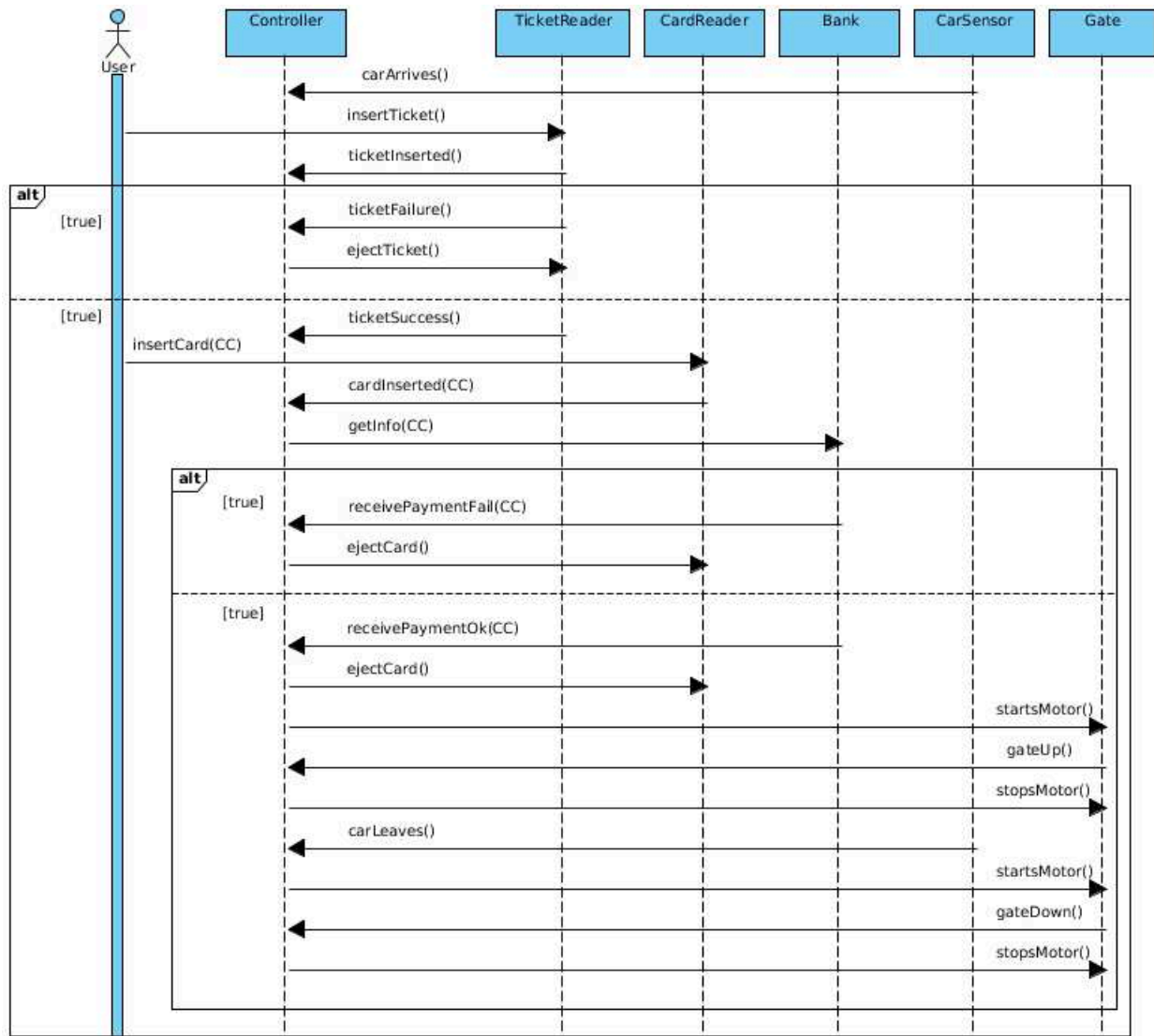


FIGURE 4.8 – Diagramme de séquence du déroulement du système

1 Invariants et réactions

2 Après l'extraction des observateurs d'état et des événements en nous basant sur la méthode
3 de l'article [?], nous cherchons à trouver les invariants et les couples condition/réaction.

4 Invariants

5 Les invariants concernent les observateurs d'état et les couples condition/réaction concernent
6 les événements.

7 Initialement la barrière est vide et il n'y a aucune voiture. Cela nous permet de définir la
8 valeur initiale de chaque observateur d'état (extrait dans l'étape précédente) :

9 `carAtGate := false; ticketActive := false; ticketReadOk := BoolS.naS;`

10 `cardKnown := false; paymentOk := BoolS.naS; gateState := GateS.down.`

11 Ces valeurs initiales changent dans le cas où le système change d'état en réaction à un événement.

CAG	TA	TRO	CK	PO	GS
<i>false</i>	<i>false</i>	<i>BoolS.naS</i>	<i>false</i>	<i>BoolS.naS</i>	<i>GateS.down</i>

CAG : carAtGate
 TA : ticketActive
 TRO : ticketReadOk
 CK : cardKnown
 PO : paymentOk
 GS : gateState

1 Invariant de carAtGate

2 À la réception de l'événement `carArrives`, la valeur de l'observateur d'état `carAtGate` change
 3 en *true* car la voiture est présente au niveau de la barrière. Nous définissons l'invariant de
 4 `carAtGate` quand sa valeur est *false*, car il y a beaucoup de situations où sa valeur est égale à
 5 *true* (moins pertinent). En partant de ce raisonnement, `carArrives` a la valeur *false* si seulement
 6 si notre système est à son état initial (c'est à dire chaque observateur d'état est à sa valeur
 7 initiale). Par conséquent, nous définissons l'invariant de l'observateur d'état `carAtGate` comme
 8 suit :

9 $\text{carAtGate} = \text{false} \Rightarrow \text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and}$
 10 $\text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down}.$

CAG	TA	TRO	CK	PO	GS
<i>false</i>	<i>false</i>	<i>BoolS.naS</i>	<i>false</i>	<i>BoolS.naS</i>	<i>GateS.down</i>

11 Invariant de ticketActive

12 Une fois à la barrière, l'utilisateur insère son ticket ; cette situation est représentée par l'occur-
 13 rence de l'événement `ticketInserted`. En nous basant sur le diagramme de séquence, l'insertion
 14 du ticket vient après l'arrivée de la voiture à la barrière, cela implique que `ticketActive` est à
 15 *true* uniquement si `carAtGate` = *true*. Cependant caractériser la situation où le ticket est inséré
 16 uniquement avec la présence de la voiture est nécessaire mais pas suffisant, car la présence de la
 17 voiture caractérise également d'autres situations (par exemple, l'insertion de la carte bancaire, la
 18 validation du paiement, etc). Pour cette raison, nous allons reprendre l'ensemble des observateurs
 19 d'état avec leurs valeurs initiales à l'exception de `carAtGate`. L'invariant est le suivant :

20 $\text{ticketActive} = \text{false} \Rightarrow \text{carAtGate} = \text{true} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and}$
 21 $\text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down}.$

CAG	TA	TRO	CK	PO	GS
<i>true</i>	<i>false</i>	<i>BoolS.naS</i>	<i>false</i>	<i>BoolS.naS</i>	<i>GateS.down</i>

22 Invariant de ticketReadOk

23 Après l'insertion du ticket, le lecteur de ticket lit ce dernier afin de déterminer s'il est bon
 24 ou non et, selon le résultat, il y aura ou bien l'occurrence de l'événement `ticketFailure` si la
 25 lecture échoue ou bien `ticketSuccess` si la lecture réussit. Dans le cas où la lecture échoue
 26 alors l'événement `ticketFailure` apparaît et le ticket est éjecté, le système revient à un état
 27 précédent (où le ticket n'est pas encore inséré). Dans le cas où la lecture réussit alors l'événement
 28 `ticketSuccess` apparaît et la valeur de l'observateur d'état `ticketReadOk` change en *true*. Donc

1 afin que la lecture réussisse il faut que la voiture soit au niveau de la barrière, le ticket soit inséré
 2 et le reste du système à son état initial (voir le diagramme de séquence). Cela nous permet de
 3 définir l'invariant suivant :

4 $\text{ticketReadOk} = \text{BoolS.trueS} \Rightarrow \text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{true} \text{ and}$
 5 $\text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down}.$

CAG	TA	TRO	CK	PO	GS
<i>true</i>	<i>true</i>	<i>BoolS.trueS</i>	<i>false</i>	<i>BoolS.naS</i>	<i>GateS.down</i>

6 Invariant de cardKnown

7 La validation du ticket est suivie par l'insertion de la carte bancaire pour le paiement, et cela est
 8 modélisé par l'occurrence de l'événement `card_Inserted(CC)` (CC : Credit Card). L'observateur
 9 d'état `cardKnown` permet de savoir si la carte bancaire est insérée et a comme valeur *true* dans
 10 le cas où la carte est insérée et *false* dans le cas inverse. En suivant le même raisonnement que
 11 les autres observateurs d'état, pour que la carte bancaire soit insérée (c'est-à-dire `cardKnown`
 12 = *true*) il est nécessaire que la voiture soit à la barrière uniquement. Les observateurs d'état
 13 `ticketActive` et `ticketReadOk` reviennent à leurs valeurs initiales car la phase du traitement
 14 de ticket est terminée. L'invariant de l'observateur d'état `cardKnown` est le suivant :

15 $\text{cardKnown} = \text{true} \Rightarrow \text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} =$
 16 $\text{BoolS.naS} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down}.$

CAG	TA	TRO	CK	PO	GS
<i>true</i>	<i>false</i>	<i>BoolS.naS</i>	<i>true</i>	<i>BoolS.naS</i>	<i>GateS.down</i>

17 Invariant de paymentOk

18 À l'insertion de la carte bancaire ou bien dans le cas où le paiement est refusé, l'événement
 19 `receivePaymentFail` apparaît et par conséquent la carte bancaire est éjectée et on revient à un
 20 état précédent, ou bien le paiement est accepté et dans ce cas là l'événement `receivePaymentOk`
 21 apparaît. L'occurrence de l'événement `receivePaymentOk` permet de changer l'état du système et
 22 donc la valeur de l'observateur d'état `paymentOk` (qui aura *true*). Si le paiement est accepté alors
 23 la voiture est au niveau de la barrière et le reste des observateurs d'état sont à leur état initial
 24 (la même chose pour `cardKnown`, car la carte bancaire est éjectée). L'invariant de l'observateur
 25 d'état `paymentOk` est le suivant :

26 $\text{paymentOk} = \text{true} \Rightarrow \text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} =$
 27 $\text{BoolS.naS} \text{ and } \text{cardKnown} = \text{false} \text{ and } \text{gateState} = \text{GateS.down}.$

CAG	TA	TRO	CK	PO	GS
<i>true</i>	<i>false</i>	<i>BoolS.naS</i>	<i>false</i>	<i>BoolS.trueS</i>	<i>GateS.down</i>

28 Invariant de gateState

29 Une fois que le paiement est validé l'utilisateur peut partir, pour cela le moteur de la barrière
 30 est mis en marche afin que la barrière soit en position haute. L'utilisateur peut partir une fois que

1 la barrière est en position haute, l'événement `carLeaves` (qui indique que la voiture est partie)
 2 apparaîtra. Le moteur se met en marche afin que la barrière revienne à une position basse. Ce
 3 déroulement permet le changement de l'état de la barrière en position haute et en position basse.
 4 Pour que la barrière soit en position haute il faut que le système soit à son état initial et que la
 5 voiture soit au niveau de la barrière. L'invariant du pseudo-état `gateState` est le suivant :
 6 `gateState = up` \Rightarrow `ticketActive = false` and `ticketReadOk = BoolS.naS` and
 7 `cardKnown = false` and `paymentOk = BoolS.naS`.

8 L'observateur `carAtGate` n'apparaît pas dans l'invariant de l'observateur d'état `gateState`
 9 car il y a une situation où `carAtGate` est égale à `true` quand la barrière est en position haute, et
 10 il y a une situation où `carAtGate` est égale à `false` (dans le cas où la voiture quitte la barrière).

CAG	TA	TRO	CK	PO	GS
-	<i>false</i>	<i>BoolS.naS</i>	<i>false</i>	<i>BoolS.naS</i>	<i>GateS.up</i>

11 Les invariants des observateurs d'état sont présentés dans le [Tableau 4.12](#). Le choix a été de
 12 chercher les invariants en fonction des valeurs possibles pour les observateurs d'état. Nous avons
 13 ici choisi de chercher, pour chaque valeur possible d'un observateur d'état, si nous pourrions
 14 formuler un invariant. Dans le [Tableau 4.12](#), chaque entrée présente un observateur d'état avec
 15 les invariants correspondants à ses valeurs. Par exemple, `ticketActive` peut prendre deux valeurs
 16 (`true` et `false`).

17 Une première remarque concerne l'invariant de l'observateur d'état `ticketActive`. Comment
 18 caractériser la condition ou la situation du système où l'observateur d'état a une valeur `true`,
 19 c'est-à-dire la situation où le ticket est inséré? Un premier raisonnement serait que, pour le ticket
 20 soit inséré, il est nécessaire que la voiture soit au niveau de la barrière, c'est-à-dire `ticketActive`
 21 $= true \Rightarrow carAtGate = true$. Cependant, la présence de la voiture n'est pas suffisante pour ca-
 22 ractériser la situation où le ticket est inséré. En effet, la présence de la voiture est nécessaire
 23 également pour les invariants des autres observateurs d'état (par exemple, la validation du paie-
 24 ment, etc). Par conséquent, il est nécessaire, lors de la définition de l'invariant de `ticketActive`,
 25 de prendre en compte la valeur des autres observateurs d'état au moment de l'insertion du ticket.
 26 Cela permet de définir l'invariant suivant (qui permettra de caractériser la situation où le ticket
 27 est inséré) :

28 `ticketActive = true` \Rightarrow `carAtGate = true` and `ticketReadOk = BoolS.naS` and
 29 `cardKnown = false` and `paymentOk = BoolS.naS` and `gateState = GateS.down`.

30 Dans le cas où `ticketActive` prend la valeur `false` il n'est pas nécessaire de préciser l'invariant
 31 car l'information apportée par la non présence du ticket n'est pas utile dans notre système.
 32 L'absence d'invariants dans le [Tableau 4.12](#) est illustrée par des lignes blanches.

33 Notons que notre système (la barrière du parking) ne se termine pas alors l'état *final* (présenté
 34 dans la [Section 4.4.6](#)) a comme invariant la valeur `false`. L'outil *EasySM* prend en compte cet
 35 observateur d'état et l'engendre automatiquement à la fois dans la phase des observateurs d'état
 36 (en ajoutant un observateur d'état *finalavec* comme invariant `true`) et dans la table des états.

37 Événements

38 Pour chaque événement les couples condition/réaction sont présentés dans les [Tableaux 4.13](#)
 39 et [4.14](#). Les conditions pour qu'un événement puisse avoir lieu sont exprimées à l'aide de va-
 40 leurs prises par certains observateurs d'état, de même que les réactions après l'occurrence d'un
 41 événement.

State Observer	Invariant
ticketActive	$\text{ticketActive} = \text{true} \Rightarrow (\text{carAtGate} = \text{true} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and } \text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down})$
ticketReadOk	$\text{ticketReadOk} = \text{BoolS.trueS} \Rightarrow (\text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{false} \text{ and } \text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down})$
cardInfo	
cardKnown	$\text{cardKnown} = \text{true} \Rightarrow (\text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{gateState} = \text{GateS.down})$
paymentOk	$\text{paymentOk} = \text{BoolS.trueS} \Rightarrow (\text{carAtGate} = \text{true} \text{ and } \text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and } \text{cardKnown} = \text{false} \text{ and } \text{gateState} = \text{GateS.down})$
gateState	$\text{gateState} = \text{GateS.up} \Rightarrow (\text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and } \text{paymentOk} = \text{BoolS.naS} \text{ and } \text{cardKnown} = \text{false})$
carAtGate	$\text{carAtGate} = \text{false} \Rightarrow (\text{ticketActive} = \text{false} \text{ and } \text{ticketReadOk} = \text{BoolS.naS} \text{ and } \text{cardKnown} = \text{false} \text{ and } \text{paymentOk} = \text{BoolS.naS})$
<i>final</i>	<i>final</i> = false

TABLE 4.12 – Observateurs d'état et invariants

1 Nous définissons l'événement *created* (voir la [Section 4.4.7](#)) avec une condition à *false* (afin
2 de correspondre au pseudo-état initial) et une réaction comportant l'initialisation de tous les
3 observateurs d'état.

4 Dans l'exemple du paiement du parking, la transition avec l'événement *created* permet de lier
5 le pseudo-état initial dans le diagramme états-transitions avec l'état qui sera le point de départ
6 de notre système (que nous avons appelé *init* dans le [Tableau 4.15](#)). Notons que l'événement
7 *created* est engendré automatiquement dans l'outil *EasySM*.

8 Comme montré dans les [Tableaux 4.13](#) et [4.14](#), dans certaines situations nous avons besoin de
9 faire une *réinitialisation* de la valeur des observateurs d'état. Il faut donc chercher le(s) moment(s)
10 pertinent(s). Nous avons déterminé qu'à la réception de l'événement `receivePaymentFail` le

1 système doit revenir à un état précédent et donc il y a besoin de réinitialiser les valeurs des
 2 observateurs d'état. Le même principe s'applique pour l'événement `gateDown`, on revient à l'état
 3 initial.

4 Afin d'avoir la possibilité d'appeler les opérations des différentes classes (par exemple : `starts-`
 5 `Motor()`) nous avons besoin d'établir des associations entre les différentes classes et la classe de
 6 contexte. Ces associations sont aussi nécessaires pour le bon fonctionnement de l'outil qui implé-
 7 mente la méthode de spécification. La Figure 4.9 montre le diagramme de classes mis à jour
 8 après l'ajout des associations entre les classes (par exemple l'association `GateG` entre la classe
 9 de contexte `Controller` et la classe `Gate`). Nous utilisons la syntaxe de l'outil *EasySM* lors des
 10 appels des fonctions, par exemple, l'appel de la fonction `startsMotor()` sera noté comme suit :
 11 `self.GateG.startsMotor()`, où :

- 12 • `self` représente l'appel à travers la classe de contexte
- 13 • `GateG` représente le nom de l'association entre la classe de contexte et la classe `Gate`
- 14 • `startsMotor()` représente la fonction qui met en marche le moteur de la barrière

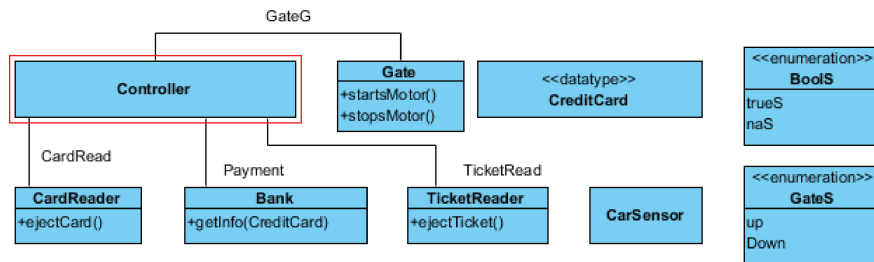


FIGURE 4.9 – Diagramme de classe de l'exemple du paiement du parking : mise à jour des classes avec les associations

Event	Condition	Reaction
created		<pre> carAtGate := false; ticketActive := false; ticketReadOk := BoolS.naS; cardInfo := Undef; cardKnown := false; paymentOk := BoolS.naS; gateState := GateS.down </pre>
ticketInserted ()	<pre> carAtGate = true and ticketActive = false and ticketReadOk = Bool.naS and cardKnown = false and paymentOk = BoolS.naS and gateState = GateS.down </pre>	<pre> ticketActive := true </pre>
ticketFailure ()	<pre> carAtGate = true and ticketActive = true and ticketReadOk = BoolS.naS and cardKnown = false and paymentOk = BoolS.naS and gateState = GateS.down </pre>	<pre> self.TicketRead.ejectTicket(); ticketActive := false </pre>
ticketSuccess ()	<pre> carAtGate = true and ticketActive = true and ticketReadOk = BoolS.naS and cardKnown = false and paymentOk = BoolS.naS and gateState = GateS.down </pre>	<pre> ticketActive := false; ticketReadOk := true </pre>
card_Inserted (CC)	<pre> carAtGate = true and ticketActive = false and ticketReadOk = BoolS.trueS and cardKnown = false and paymentOk = BoolS.naS and gateState = GateS.down </pre>	<pre> cardInfo := CC; self.Payment.getInfo(CC); ticketReadOk := BoolS.naS; cardKnown := false; </pre>

TABLE 4.13 – Événements et Conditions/Réactions (Partie 1)

Event	Condition	Reaction
receivePaymentOk (CC)	carAtGate = <i>true</i> and ticketActive = <i>false</i> and ticketReadOk = BoolS.naS and cardKnown = <i>true</i> and paymentOk = BoolS.naS and gateState = GateS.down	paymentOk := BoolS. <i>true</i> S; cardInfo := Undef; cardKnown := <i>false</i> ; self.CardReader.ejectCard(); self.GateG.startsMotor()
receivePaymentFail (CC)	carAtGate = <i>true</i> and ticketActive = <i>false</i> and ticketReadOk = BoolS.naS and cardKnown = <i>true</i> and paymentOk = BoolS.naS and gateState = GateS.down	self.CardReader.ejectCard(); cardKnown := <i>false</i>
gateUp ()	paymentOk = <i>true</i>	gateState := GateS.up; paymentOk := BoolS.naS; self.GateG.stopsMotor()
gateDown ()	carAtGate = <i>false</i> and gateState = GateS.up	gateState := GateS.down; self.GateG.stopsMotor();
carArrives ()	carAtGate = <i>false</i> and gateState = GateS.down	carAtGate := <i>true</i>
carLeaves ()	carAtGate = <i>true</i> and gateState = GateS.up	carAtGate := <i>false</i> ; self.GateG.startsMotor()

TABLE 4.14 – Événements et Conditions/Réactions (Partie 2)

1 **Les états et les transitions.** En nous basant sur les résultats des phases précédentes,
 2 dans cette étape nous trouvons les différents états et transitions qui constituent le diagramme
 3 d'états-transitions final. Les états sont résumés dans le [Tableau 4.15](#).

CAG	TA	TRO	CK	PO	GS	Final	State
<i>true</i>	<i>true</i>	naS	<i>false</i>	naS	down	<i>false</i>	ticketProcessing
<i>true</i>	<i>false</i>	<i>true</i> S	<i>false</i>	naS	down	<i>false</i>	waitCard
<i>true</i>	<i>false</i>	naS	<i>true</i>	naS	down	<i>false</i>	waitBankAnswer
<i>true</i>	<i>false</i>	naS	<i>false</i>	<i>true</i> S	down	<i>false</i>	paymentValidated
<i>true</i>	<i>false</i>	naS	<i>false</i>	naS	up	<i>false</i>	gateRaised
<i>true</i>	<i>false</i>	naS	<i>false</i>	naS	down	<i>false</i>	carPresent
<i>false</i>	<i>false</i>	naS	<i>false</i>	naS	up	<i>false</i>	carGone
<i>false</i>	<i>false</i>	naS	<i>false</i>	naS	down	<i>false</i>	init
<i>true</i>	<i>true</i>	<i>true</i> S	<i>true</i>	<i>true</i> S	up	<i>true</i>	impossible
<i>true</i>	<i>true</i>	<i>true</i> S	<i>true</i>	<i>true</i> S	up	<i>false</i>	impossible
...	impossible

CAG : carAtGate
 TA : ticketActive
 TRO : ticketReadOk
 CK : cardKnown
 PO : paymentOk
 GS : gateState

TABLE 4.15 – Les états du diagramme états-transitions

4 Comme le [Tableau 4.15](#) le montre, nous regroupons les observateurs d'état qui sont pertinents
 5 pour la détection des états du diagramme. Les observateurs d'état sont ordonnés dans les colonnes
 6 selon l'ordre dans lequel ils sont considérés.

7 Chaque ligne représente une combinaison des différentes valeurs des observateurs d'état et, en
 8 nous basant sur les invariants ou les conditions des événements, nous décidons si cette combinai-
 9 son représentera un état (voir plus de détails dans la [Section 4.4.8](#)). Toute combinaison qui entre
 10 en contradiction avec l'un des invariants ou les conditions des événements ne peut caractériser
 11 un état (noté par *impossible* dans la table). Il y a de nombreux états impossibles ([Tableau 4.15](#)).

12 Une remarque concerne l'événement *created* dans le [Tableau 4.13](#) et l'état *init* dans [Ta-](#)
 13 [bleau 4.15](#). En effet, l'événement *created* correspond à la situation initiale du système (tel que
 14 chaque observateur d'état a une valeur initiale). Les valeurs de la ligne de l'état *init* corres-
 15 pondent aux valeurs des observateurs d'état de l'état cible de *created* et donc cet état sera l'état
 16 initial.

17 Une manière de vérifier si une ligne peut représenter un état possible est de vérifier si les
 18 valeurs des observateurs d'état dans cette ligne entrent en contradiction avec un invariant. Par
 19 exemple, l'état *gateRaised* est possible car la valeur de *gateState* (up) n'entre pas en contradic-
 20 tion avec la valeur des autres observateurs d'état (c'est-à-dire, CAG = *true*, TA = *false*, TRP
 21 = naS, CK = *false* et PO = naS).

22 Nous pouvons remarquer qu'une ligne corresponde à un état possible il faut vérifier que les
 23 valeurs de toute la ligne correspondent aux conditions d'un ou plusieurs événements, et ensuite
 24 de choisir l'événement qui a la réaction correspondante à l'état. Par exemple, les valeurs de la
 25 ligne de l'état *waitCard* correspondent aux conditions de l'événement *ticketSuccess*.

26 Si une anomalie (par exemple l'absence d'états remplissent les conditions d'un événement)
 27 est détectée dans cette phase cela implique que les conditions utilisées dans les invariants sont
 28 incorrectes et donc il faut remonter jusqu'à l'étape 2 (détection des invariants) afin d'y remédier.

29 Après avoir identifié les différents états du diagramme états-transitions nous passons à la
 30 détection des transitions. Chaque événement est un candidat pour déclencher une transition
 31 dans le diagramme. La première étape consiste à mettre en clair l'événement qui sera celui de la

- 1 transition (en partant des Tableaux 4.13 et 4.14), la possibilité d'avoir des gardes et des actions,
 2 les conditions que les états source et cible devraient respecter. La Figure 4.10 montre l'application
 3 de cette étape sur l'événement `ticketInserted`.

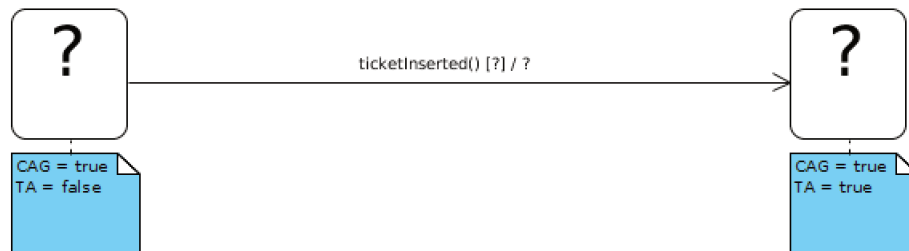


FIGURE 4.10 – La première étape pour trouver la transition de l'événement `ticketInserted`

- 4 La seconde étape consiste à chercher dans le Tableau 4.15 les états qui vont correspondre
 5 aux conditions de l'état source et l'état cible. La Figure 4.11 montre l'application de la seconde
 6 étape sur l'événement `ticketInserted`.

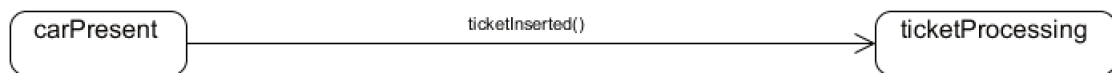


FIGURE 4.11 – La seconde étape pour trouver la transition de l'événement `ticketInserted`

- 7 Les Figure 4.12 et Figure 4.13 montrent l'application de la première et seconde étape sur
 8 l'événement `card_Inserted`.

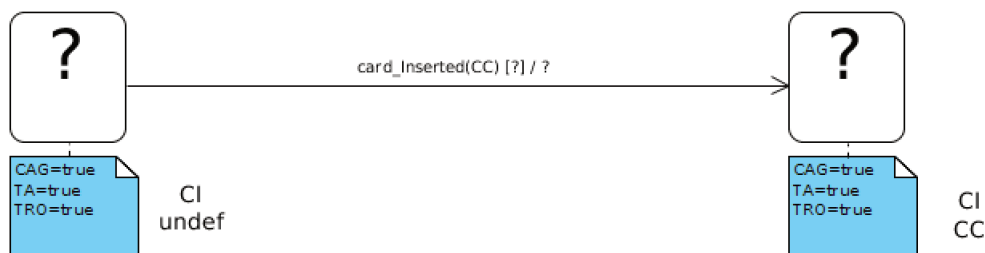


FIGURE 4.12 – La seconde étape pour trouver la transition de l'événement `card_Inserted`

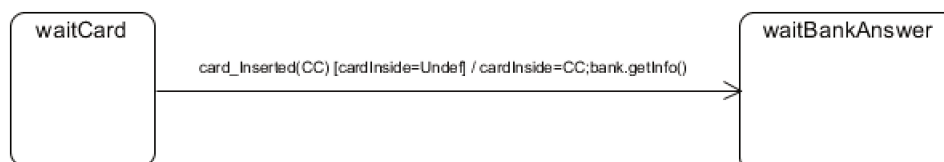


FIGURE 4.13 – La seconde étape pour trouver la transition de l'événement `card_Inserted`

- 9 La particularité de cet événement est la présence d'un observateur d'état (`cardInfo`) qui
 10 n'est pas utilisé dans le Tableau 4.15 pour l'extraction des états. L'apparition de l'observateur

1 d'état dans les conditions de l'événement permet de l'ajouter comme une garde dans la transition.
 2 L'apparition de l'observateur d'état dans les réactions de l'événement permet de l'ajouter dans
 3 une action de la transition.

4 **Gestion des transitions par l'outil *EasySM*.** Nous avons présenté ci-dessus comment
 5 engendrer les transitions à partir des événements (avec leurs conditions/réactions) et la table des
 6 états. Cette explication a pour but de montrer le principe de la génération des transitions dans
 7 le cas où l'utilisateur n'utilise pas l'outil *EasySM*. En effet, l'outil *EasySM* engendre automati-
 8 quement les transitions sans que l'utilisateur n'ait besoin de le faire (en nous basant sur le même
 9 principe expliqué ci-dessus).

10 **Construction du diagramme états-transitions.** La dernière étape consiste à engen-
 11 drer le diagramme états-transitions en partant des différents éléments trouvés dans les étapes
 12 précédentes. Chaque transition aura comme source et cible un état parmi les états trouvés. L'état
 13 source aura les mêmes conditions que l'événement alors que l'état cible aura les mêmes valeurs
 14 que l'état résultant de l'exécution de l'événement. Chaque transition aura si nécessaire des gardes
 15 ainsi que des opérations qui représentent les différentes réactions vis à vis de l'événement de la
 16 transition.

17 La Figure 4.15 représente le diagramme états-transitions résultat de l'exemple du paiement
 du parking.

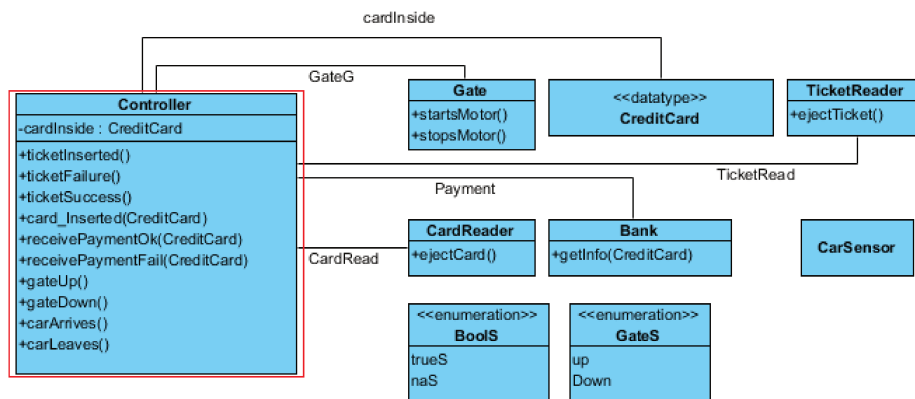


FIGURE 4.14 – Diagramme de classes final de l'exemple du paiement du parking

18 Dans la Figure 4.15, `cardInfo` est la variable qui permet de savoir si la carte bancaire est
 19 dans le lecteur de carte et `CC` est la carte bancaire (Credit Carde).

20 En parallèle de la construction du diagramme états-transitions, le diagramme de classes
 21 de départ est mis à jour. Tout événement qui n'apparaît pas comme une opération dans le dia-
 22 gramme de classes sera une opération dans la classe du contrôleur (par exemple, `ticketInserted`,
 23 `ticketFailure`, etc). Tout observateur d'état qui ne contribue pas à l'extraction des différents
 24 états du diagramme figurera comme un attribut dans la classe du contrôleur (par exemple,
 25 `cardInfo`). La Figure 4.14 montre la nouvelle version du diagramme de classes.
 26

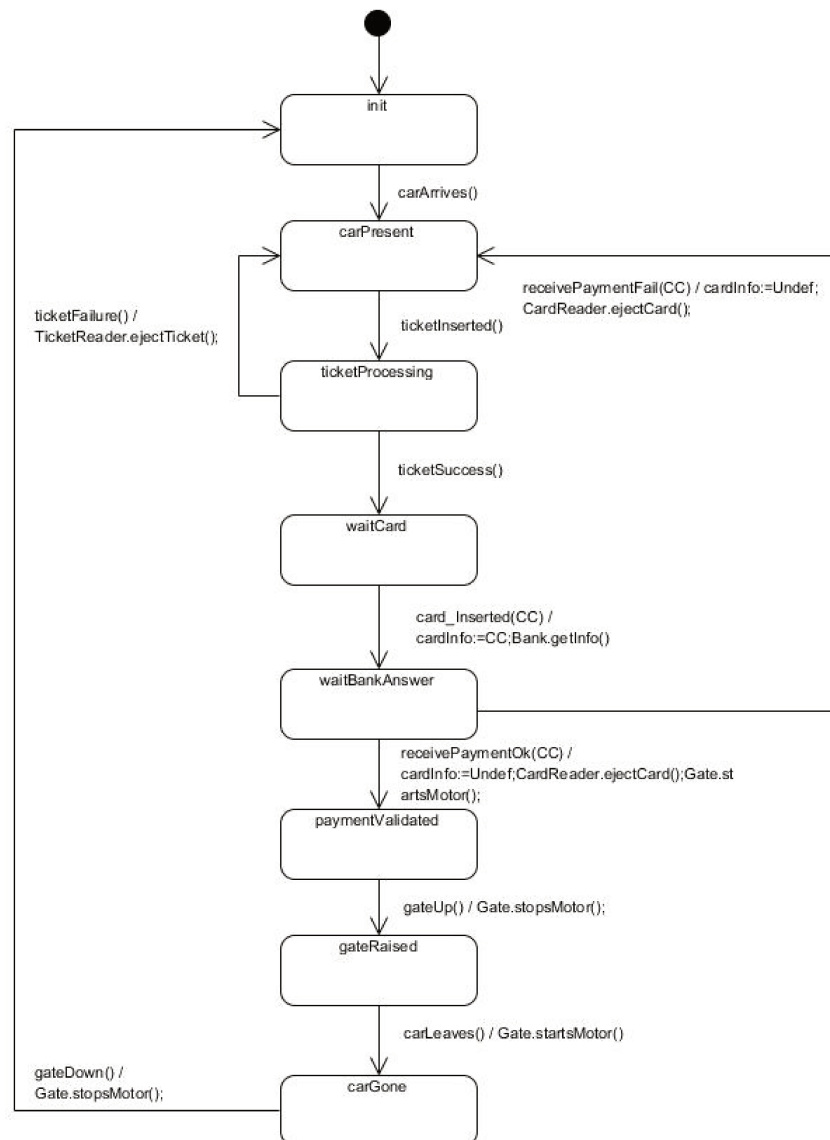


FIGURE 4.15 – Diagramme états-transitions de l'exemple du paiement du parking

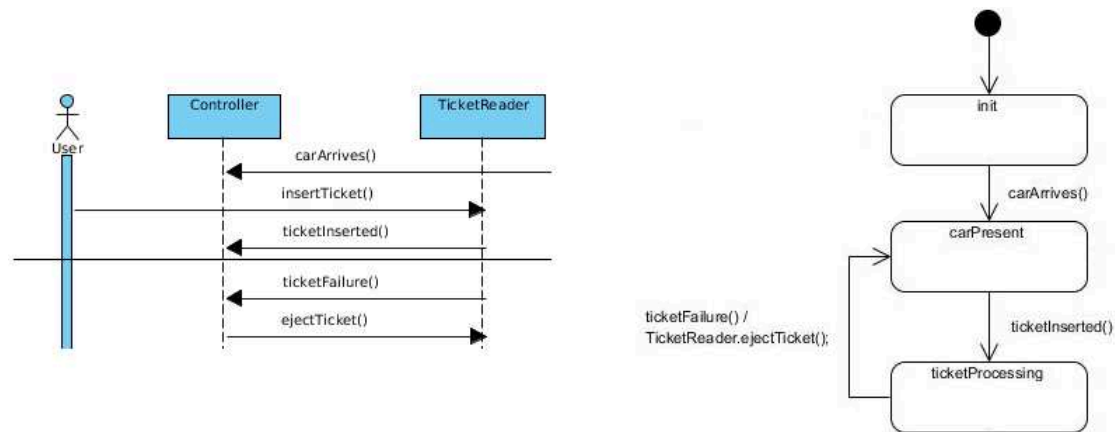


FIGURE 4.16 – Diagramme de séquence et états-transitions : Comparaison

Vérification en utilisant le diagramme de séquence

Afin de consolider la cohérence et la validité des différentes étapes de la méthode nous avons introduit le diagramme de séquence, à la fois pour avoir le séquençement du fonctionnement de notre système, mais aussi pour vérifier la cohérence du résultat obtenue par la méthode (voir les détails dans la [Section 4.4.11](#)).

Nous vérifions dans un premier temps la validité des invariants des observateurs d'état. Par exemple, l'invariant associé à la valeur `true` de l'observateur d'état `ticketActive` est le suivant : `carAtGate = true and ticketReadOk = BoolS.naS and cardKnown = false and paymentOk = BoolS.naS and gateState = GateS.down`. Le séquençement dans le diagramme de séquence avant que `ticketActive` soit égal à `true` est le suivant : `carArrives () ⇒ ticketInserted`. En comparant les deux cas, c'est-à-dire l'invariant et le séquençement d'occurrence des événements, nous constatons que le séquençement d'occurrence des événements confirme la validité de l'invariant. En d'autres termes, avant que le ticket soit inséré il y a uniquement l'observateur d'état modélisant la présence de la voiture qui soit à une valeur différente de sa valeur initiale (c'est-à-dire `true` au lieu de `false`), le reste des observateurs d'état ont la valeur initiale. Cela est concrétisé par à la fois l'occurrence d'événements où dans le diagramme de séquence il y a uniquement l'événement `carArrives` qui se produit, et par la valeur des invariants où uniquement `carAtGate` est à `true`. Nous avons constaté que cette validité est pour tous les invariants et pas uniquement celui de `ticketActive`.

Nous vérifions ensuite la validité des couples condition / réactions des événements. Par exemple, avant l'occurrence de l'événement `carArrives` il n'y a eu aucun événement qui est apparu et par conséquent les valeurs des observateurs d'état de notre système sont à leurs valeurs initiales (ce qui est le cas dans la condition de l'événement qui est : `carAtGate = false and gateState = GateS.down`). La vérification de la réaction se fait de la même manière en comparant les valeurs des observateurs d'état dans la réaction de l'événement et ceux dans le diagramme de séquence. Nous appliquons le même principe pour le reste des événements, et nous constatons qu'il y a aucune anomalie ou incohérence.

Nous vérifions enfin la validité du diagramme états-transitions final en le comparant au diagramme de séquence. Dans le diagramme états-transitions de l'exemple du paiement du parking, le système commence dans un état initial, ensuite il y a la détection de la présence d'une voiture, qui est suivi par l'insertion du ticket (et sa vérification) puis de la carte bancaire (et sa vérifica-

tion). Le paiement sera ensuite validé, la barrière mise en position haute pour que la voiture sorte du parking. Enfin la remise de la barrière en position basse et le retour à l'état initial du système. Le même scénario est représenté (en utilisant l'occurrence des événements) dans le diagramme de séquence. Par conséquent, cela nous permet de dire que notre diagramme états-transitions correspond au diagramme de séquence ce qui rajoute un aspect de cohérence aux étapes de la méthode. Une comparaison d'une partie du diagramme de séquence et du diagramme états-transitions est présentée dans la [Figure 4.16](#).

Hiérarchie des états : états composites et états orthogonaux

Nous présentons dans cette section l'introduction de la hiérarchie dans la méthode de spécification avec les diagrammes états-transitions. Nous présentons dans un premier temps les différentes situations où il est possible d'intégrer de la hiérarchie (et par conséquent les états composites). Ensuite nous présentons, sous forme de phases, comment introduire les états composites.

Nous avons modifié légèrement le diagramme états-transitions correspondant à l'exemple de la barrière de péage présenté dans la [Figure 4.15](#) pour introduire un cas particulier de hiérarchie : nous avons ajouté dans chaque état une transition allant vers l'état d'échec `failure` et modélisant l'occurrence de l'événement d'échec `error` (voir [Figure 4.17](#)).

Situations La question que nous pouvons nous poser est : quelles sont les situations où il est pertinent d'introduire un état composite ? Nous avons identifié trois situations où introduire de la hiérarchie, et par conséquent des états composites, que nous présentons ci-dessous :

- Situation 1 : il s'agit d'une situation où une étape du processus est décomposée en plusieurs parties contribuant à son exécution. En terme d'événement, c'est un "macro" événement (en anglais macro-event) composé de plusieurs événements, par exemple, le paiement se fait en trois temps : l'insertion de la carte bancaire, l'attente de la réponse de la banque et la validation du paiement. On peut donc rassembler ces différentes étapes au sein d'un état composite (par exemple, `payment` dans la [Figure 4.17](#)).

Cette situation peut être détectée en examinant le diagramme états-transitions et à l'aide de l'analyse du texte. D'autre part, les états qui sont rassemblés dans un état composite doivent bien sûr être connectés entre eux par des transitions.

- Situation 2 : c'est une situation où la valeur d'un observateur d'état permet de délimiter un ensemble de composantes qui sont connectées, c'est-à-dire les regrouper dans une seule composante. Par exemple, la présence de la voiture permet de regrouper un ensemble de situations liées tels que : l'insertion du ticket, la lecture de la carte bancaire, etc. Pour détecter cette situation il faut disposer de la table des états fournie par la méthode (avec les valeurs des observateurs d'états pour chaque état). Toutefois, dans le [Tableau 4.15](#) on voit que les observateurs d'états conservent une même valeur pour plusieurs états et donc ce seul critère ne suffit pas : il faut que les états sélectionnés soient connectés. Par exemple, la lecture du ticket ne permet le regroupement que d'un seul état qui est le traitement du ticket et donc il n'est pas intéressant.

- Situation 3 : c'est une situation où des éléments du système partagent un événement qui change l'état de du système vers un autre état (le même pour tous les éléments). Par exemple, le cas où l'événement `error` conduit à l'état d'échec `failure` dans la [Figure 4.17](#).

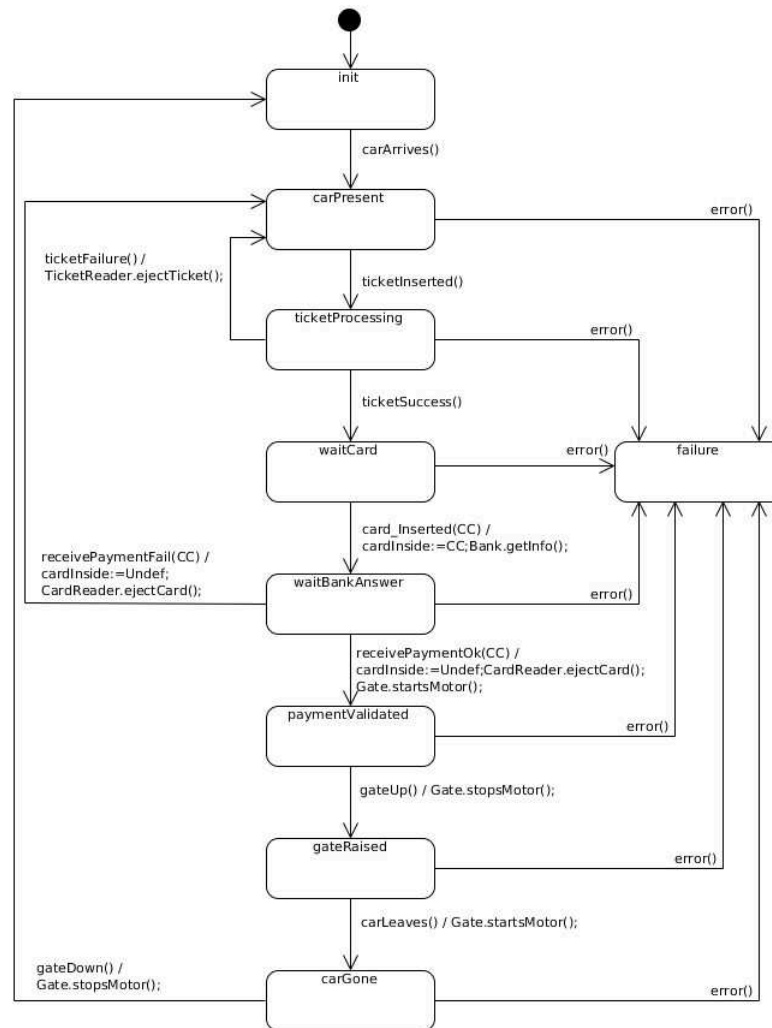


FIGURE 4.17 – Diagramme états-transitions de la barrière de péage : nouvelle version

En se basant sur les trois situations présentées ci-dessus, nous formalisons les règles suivantes :

- Règle 1 : est définie en se basant sur la première situation : *pour toute étape composée de plusieurs parties, on définit un état composite la représentant, en précisant les sous-états qui seront contenus (en utilisant le diagramme états-transitions et l'analyse du texte).*
- Règle 2 : est définie en se basant sur la deuxième situation : *pour tout observateur d'état délimitant un ensemble d'état connectés, on définit un état composite qui contiendra ces états (nous utilisons la table des états et le diagramme états-transitions).*
- Règle 3 : est définie en se basant sur la troisième situation : *pour tout ensemble d'états ayant une transition avec le même événement et allant vers le même état destination, on définit un état composite qui contiendra ces états (nous utilisons le diagramme états-transitions).*

L'ordre d'application de ces règles n'est pas défini ce qui peut laisser de l'indéterminisme par rapport au résultat. Chaque résultat engendré par un ordre d'application est considéré comme une possibilité de modéliser le système. Cependant il se peut que deux règles (ou plus) soient incompatibles, c'est-à-dire que l'intersection des deux états composites engendrés par deux règles contienne des états (simples ou composites) en commun. Dans ce cas de figure, une seule règle

1 (et non pas deux) sera appliquée. Le choix de la règle à appliquer n'est pas déterminé (cela laisse
2 un choix à l'utilisateur).

3 Notez que l'état composite introduit (par l'une des règles ci-dessus) doit avoir un nom différent
4 des états existants dans le diagramme (qu'ils soient simples ou composites).

5 **Condition**

6 En plus des règles que nous définissons au-dessus, nous présentons dans ce qui suit une règle
7 que nous pensons nécessaire pour compléter ces règles.

8 *Pour chaque état composite $S_1 \in \mathbf{S}$ résultat de l'application des règles d'introduction de la*
9 *hiérarchie, $\forall s_1 \in S_1, \exists s_2 \in S_1, \exists t \in T$ tel que t est une transition entre s_1 et s_2*

10 Les sous-états d'un état composite (engendré par l'application de la règle 1) doivent être liés
11 par des transitions, en effet leur présence dans le même état composite signifie qu'ils traitent le
12 même événement. Par exemple dans la [Figure 4.17](#), dans le cas de la présence de la voiture, il y
13 a une situation où la voiture est présente (`carPresent`) et une autre où la voiture est absente
14 (`carGone`), alors une première réflexion est d'avoir un état composite qui inclut ces deux états
15 (`carPresent` et `carIn`). Cependant nous remarquons qu'il n'y a aucune transition directe entre
16 ces deux états ni de transitions semblables allant vers un état d'échec, de ce fait l'état composite
17 sera rejeté car sa présence est inutile.

18 En se basant sur le résultat de l'application de la méthode de spécification ainsi que les trois
19 règles d'introduction de la hiérarchie, nous définissons les deux phases suivantes (introduction
20 de la hiérarchie) :

21 **Phase 1 [identification des états composites candidats et de leurs sous-états]**

22 Rappelons que nous nous basons sur les résultats obtenus par la méthode de spécification ainsi
23 que les règles d'introduction afin d'introduire la hiérarchie et par conséquent les états composites.
24 Nous utilisons dans cette phase la table des états, le diagramme états-transitions et le description
25 textuelle afin d'introduire les états composites. Dans un premier temps, nous détectons les états
26 composites candidats possibles à introduire en se basant sur les deux premières situations ainsi
27 que les deux premières règles. En utilisant la première règle nous définissons les états composites
28 qui modélisent les états du système décomposables en sous états. Par exemple, l'état composite
29 `payment` correspond à la phase de paiement de l'entrée au parking qui se déroule en trois étapes :
30 l'insertion de la carte bancaire, le contact de la banque et la validation du paiement. En utilisant
31 la seconde règle nous définissons les états composites qui modélisent le regroupement d'états en
32 se basant sur la valeur des observateurs d'états. Par exemple, l'état composite `carIn` modélise le
33 phase d'exécution du système à la présence de la voiture au niveau de la barrière (la valeur de
34 l'observateur d'état `carAtGate` est égale à `true`).

35 Dans un second temps, et en utilisant le diagramme états-transitions, nous définissons les
36 sous-états qui vont être inclus dans chaque état composite. Nous définissons une table indi-
37 quant, pour chaque état composite à ajouter, les états (composites ou simples) qui y sont
38 contenus comme montré dans le [Tableau 4.16](#). Le [Tableau 4.16](#) sera mis à jour à chaque fois
39 qu'un état composite est ajouté. L'application de la règle 1 permet de définir les états com-
40 posites qui vont contenir d'autres états simples (ou composites). Par exemple, l'état composite
41 `payment` (l'exemple de la barrière de péage) contiendra les états : `waitCard`, `waitBankAnswer`
42 et `paymentValidated` (voir [Figure 4.17](#)). L'état composite `carIn` va contenir : `carPresent`,
43 `ticketProcessing`, `payment`, `gateRaised`.

État composite	in	cp	tp	wc	wba	pv	gr	cg	fl	pa	ci
payment				X	X	X					
carIn		X	X				X			X	

Abréviations utilisées pour les noms d'état dans la table

in : init cp : carPresent tp : ticketProcessing
 wc : waitCard wba : waitBankAnswer pv : paymentValidated
 gr : gateRaised cg : carGone fl : failure
 pa : payment ci : carIn

TABLE 4.16 – Table des états composites

Phase 2 [Identification des transitions dans les états composites] Une fois les états composites déterminés, un problème qui se pose est la prise en compte éventuelle de l'introduction des états composites pour modifier la définition des transitions (état source et destination). Dans quel cas serait-il pertinent ou non de proposer l'état composite introduit comme état source ou cible d'une transition ?

Une première solution serait *de conserver la définition des transitions* en utilisant que les états simples (donc cela n'inclut pas les états composites). Dans la [Figure 4.21](#) l'état composite **payment** n'est ni source ni destination pour aucune des transitions allant ou sortant de ses sous-états. Il est possible d'ajouter un pseudo-état initial ou un état final dans l'état composite (même en absence de transitions allant ou sortant de cet état). Cette possibilité est un choix qui n'ajoute pas de sémantique ou de sens au diagramme mais permet d'indiquer l'entrée et la sortie de l'état composite. Dans la [Figure 4.18](#) l'état composite **S** comporte un état final sans qu'il y ait de transitions sortant de ce dernier.

La seconde solution consiste à introduire, dans la définition des transitions, les états composites en respectant certaines conditions.

- Dans le cas des *transitions en entrée* : si un état simple est un état lié à un pseudo-état initial (**S2** dans le bas de la [Figure 4.18](#)) dans un état composite alors les transitions entrantes vers cet état auront comme destination l'état composite qui le contient (**S** dans la [Figure 4.18](#)).
- Dans le cas des *transitions en sortie* :
 - Toute transition allant d'un sous-état (lié à un état final) d'un état composite aura comme source cet état composite. Par exemple, la transition allant de l'état **S2** vers l'état **S3** (dans le haut de la [Figure 4.19](#)) aura comme source l'état composite **S** (dans le bas de la [Figure 4.19](#)).
 - Toute transition avec événement allant d'un sous-état d'un état composite sera inchangé. Par exemple, la transition avec événement **a** entre l'état **S2** et **S4** en haut de la [Figure 4.19](#) gardera le même état source car si elle avait **S** pour état source, cela changerait la sémantique (avec de multiples possibilités d'interrompre l'activité de **S** dès l'occurrence de l'événement **a**).

Si l'état **S2** était remplacé par **S** alors à l'occurrence de l'événement **a** ou bien il y a la sortie de l'état **S** à partir de **S1** ou bien à partir de **S2** (sachant que la transition de base est entre l'état **S2** et l'état **S4**). Afin d'éviter cette situation la source de la transition sera inchangée et ainsi la sémantique du diagramme sera conservée.

1 Un dernier point dans cette phase est l'application de la règle 3 sur le diagramme états-
 2 transitions, en effet cette règle factorise les transitions avec le même événement partant de diffé-
 3 rents états vers un seul état destination. Le résultat de l'application de la règle est la suppression
 4 des transitions des états concernés, l'ajout d'un état composite qui englobe ces états, et l'ajout
 5 d'une transition de l'état composite vers l'état destination avec l'événement correspondant. Dans
 6 la Figure 4.17 il y a sept transitions, allant de différents états vers l'état **failure**, avec l'évé-
 7 nement **error**. L'application de la règle 3 permet d'introduire un nouvel état composite, de
 8 mettre à jour la table des états composites (comme montré dans le Tableau 4.17) et de modi-
 9 fier le diagramme états-transitions correspondant (comme montré dans la Figure 4.21). Dans
 10 la Figure 4.20 trois transitions avec l'événement **a** allant des états **S1**, **S2** et **S3** vers l'état **S4**
 11 (le diagramme états-transition d'en haut de la Figure 4.20). La partie basse de la Figure 4.20
 12 montre l'introduction de l'état composite **S** (contenant les états **S1**, **S2**, **S3**) avec la suppression
 13 des trois transitions avec l'événement **a** allant des états **S1**, **S2** et **S3** et l'ajout de la transition
 14 (avec l'événement **a**) allant de **S** vers **S4**.

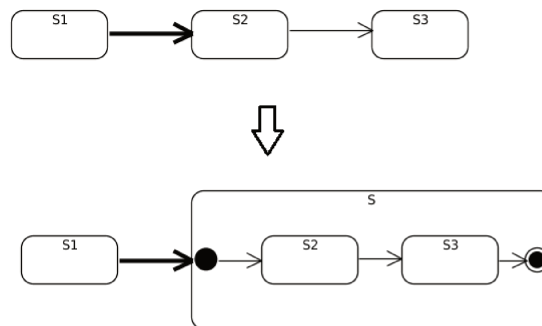


FIGURE 4.18 – Exemple de transition vers état simple dans un état composite

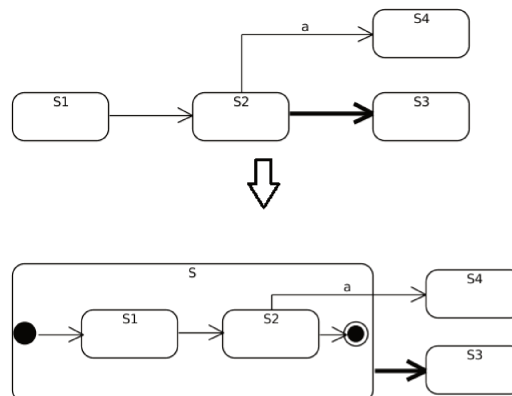


FIGURE 4.19 – Exemple de transition (avec ou sans événement) originaire d'un état simple dans un état composite

15 **Exemple** La Figure 4.21 montre le nouveau diagramme états-transitions avec introduction de
 16 la hiérarchie dans le cas du paiement et de deux autres états composites. Dans la Figure 4.21 il y a
 17 un état composite global **Success** qui modélisera la partie de notre système avec les cas de succès.
 18 Cet état permet d'éviter d'avoir des transitions avec l'événement **error** issues de tous les états.
 19 Ces transitions seront factorisées en une seule transition allant de l'état composite contenant tous

État composite	in	cp	tp	wc	wba	pv	gr	cg	fl	pa	ci	su
payment				X	X	X						
carIn		X	X				X			X		
success								X			X	

in : init cp : carPresent tp : ticketProcessing
 wc : waitCard wba : waitBankAnswer pv : paymentValidated
 gr : gateRaised cg : carGone fl : failure
 pa : payment ci : carIn su : success

TABLE 4.17 – Table des états composites : mise à jour

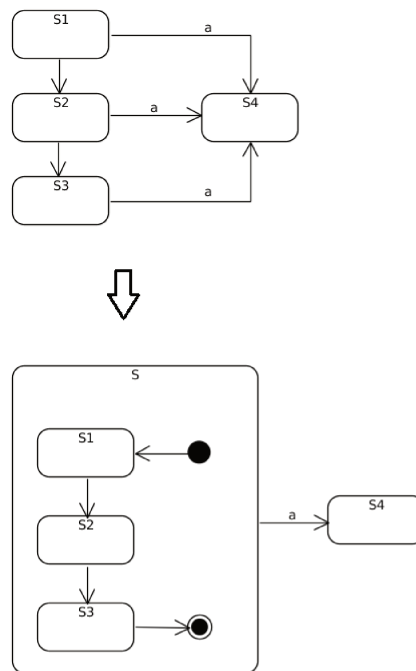


FIGURE 4.20 – Exemple d'application de la règle 3 sur un diagramme états-transitions

1 les autres états (résultat de l'application de la règle 3). Un état composite `payment` qui contient
 2 les états : `waitCard`, `waitBankAnswer` et `paymentValidated` (résultat de l'application de la
 3 règle 1). Enfin un état composite `carIn` qui contient `carPresent`, `ticketProcessing`, `payment`
 4 et `gateRaised` (résultat de l'application de la règle 2). Cela permet de mettre en évidence le
 5 fonctionnement du système dans le cas où la voiture est présente au niveau de la barrière de
 6 péage.

7 La Figure 4.22 est une autre version du diagramme états-transitions de la barrière de péage
 8 avec hiérarchie. Cette version est fautive à cause de la présence de la transition entre l'état `init`
 9 et l'état `carIn`. Étant donné que l'état `carIn` ne comporte pas un pseudo-état initial, franchir
 10 cette transition implique à l'entrée dans l'état `carIn` on sera dans l'état `carPresent` ou l'état
 11 `ticketProcessing`, etc. Cela entre en contradiction avec la version originale (voir Figure 4.21)
 12 où l'entrée dans l'état `carIn` implique l'entrée dans l'état `carPresent`.

13 La Figure 4.23 est une autre version du diagramme états-transitions de la barrière de péage
 14 avec hiérarchie. Cette version est également fautive à cause de la présence de la transition entre
 15 l'état `init` et l'état `Success`. Étant donné que l'état `Success` ne comporte pas un pseudo-état

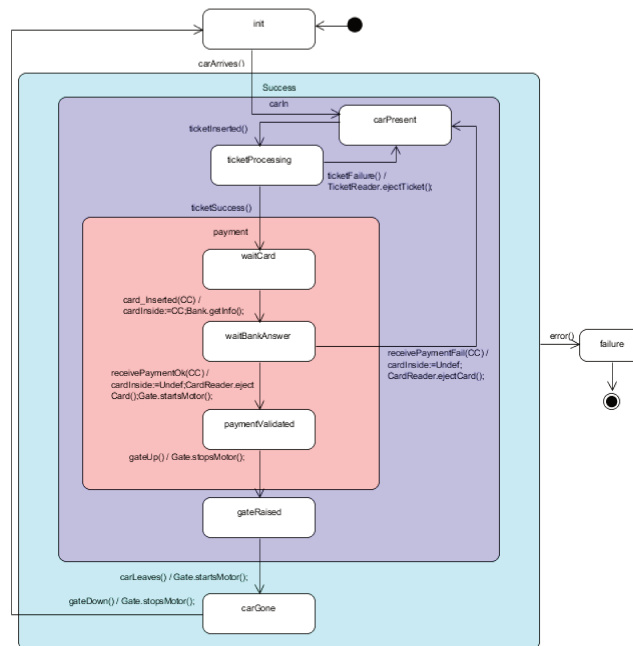


FIGURE 4.21 – Diagramme états-transitions de la barrière de péage avec hiérarchie

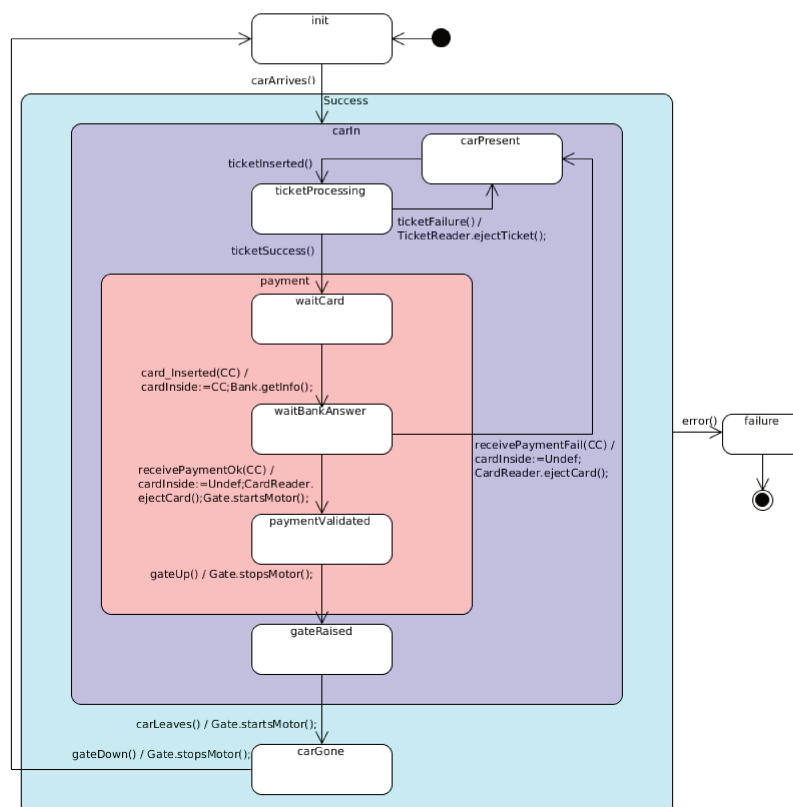


FIGURE 4.22 – Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)

- 1 initial, franchir cette transition implique à l'entrée dans l'état **Success** on sera dans l'état **carIn**
- 2 ou l'état **carGone**. Cela entre en contradiction avec la version originale (voir Figure 4.21) où

- 1 l'entrée dans l'état `Success` implique l'entrée dans l'état `carIn`.

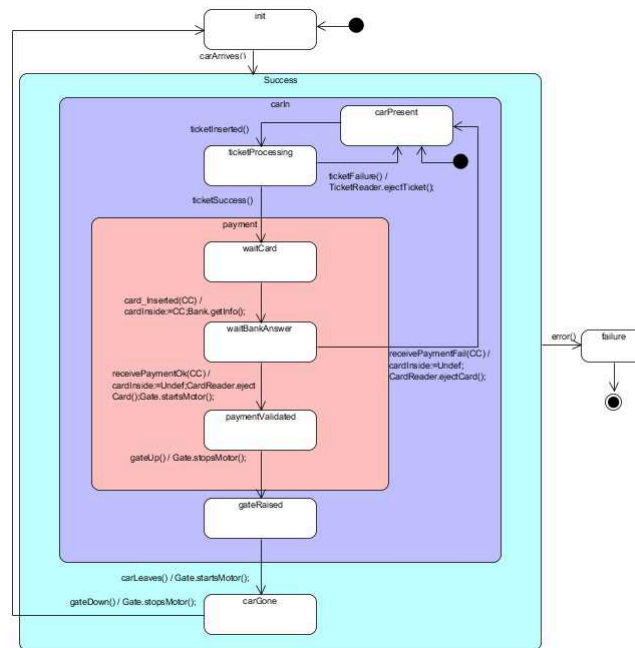


FIGURE 4.23 – Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)

2 Qualité du diagramme états-transitions

3 Afin d'évaluer la qualité d'un diagramme états-transitions l'utilisation de métriques et de
 4 facteur de qualité est nécessaire. Les métriques permettent l'évaluation quantitative, cependant
 5 les facteurs de qualité permettent l'évaluation qualitative d'un diagramme états-transitions.

6 **Notation** Pour chaque méthode de modélisation/spécification il faut préciser l'ensemble des
 7 éléments syntaxiques utilisés dans la modélisation. Cette précision permet à l'utilisateur de savoir
 8 à l'avance quels sont les éléments syntaxiques acceptés par la méthode.

9 Dans la méthode de spécification présentée dans la ??, les éléments syntaxiques pris en
 10 compte sont les suivants :

- 11 • Les états simples et composites simples (avec une seule région)
- 12 • Les transitions avec événements
- 13 • Les états finaux et les pseudo-états initiaux
- 14 • Les actions et les gardes sur les transitions
- 15 • La hiérarchie des états

1 Métriques

2 Nous avons montré dans les sections précédentes l'importance d'utiliser une méthode afin
 3 de modéliser/spécifier un système. Cette utilisation permet d'éviter les erreurs communes et
 4 d'avoir un diagramme correspondant à la description du système. Cependant, comment évaluer la
 5 qualité du diagramme, la facilité de la compréhension et l'absence de redondance ? Une solution
 6 dans le domaine de la modélisation est d'utiliser les métriques et les facteurs de qualité. Ces
 7 éléments permettent d'évaluer la qualité du diagramme états-transitions. En partant du résultat
 8 de l'évaluation nous proposons alors des solutions pour avoir une qualité meilleur du diagramme
 9 évaluer.

10 Il existe différents facteurs de qualités et métriques qui permettent l'évaluation d'un dia-
 11 gramme états-transitions. Le travail de [?] regroupe des métriques de la littérature le plus utilisées
 12 parmi eux :

13 *Métriques basiques*

- 14 • *États* : le nombre d'états d'un diagramme états-transitions
- 15 • *Actions (activités) des états* : le nombre d'activités définies pour les états dans un diagramme
16 états-transitions
- 17 • *Transitions* : nombre de transitions dans un diagramme états-transitions
- 18 • *Gardes des transitions* : nombre de transitions avec une garde dans un diagramme états-
19 transitions
- 20 • *Événements des transitions* : nombre des événements des transitions dans un diagramme
21 états-transitions

22 *Métriques complexes*

- 23 • *Complexité cyclomatique* : $M = E - N + 2P$, où E est le nombre d'arêtes du graphe, N : le
24 nombre de noeuds du graphe et P : le nombre de composantes connexes du graphe (plus
25 le nombre est grand plus la complexité est grande)

26 *Smells*

- 27 • *Pseudo-état initial et état final*
- 28 • *État sans nom*
- 29 • *Pas de transitions entrantes pour un état*
- 30 • *Pas de transitions sortantes pour un état*
- 31 • *Pas de gardes dans un pseudo-état choice*

32 En se référant aux métriques basiques ou complexes définies dans la littérature ([?]), nous
 33 définissons les métriques suivantes :

34 **Duplication** Un diagramme contient une duplication d'états s'il existe un ensemble d'états
 35 avec les mêmes transitions en sortie et allant vers les mêmes états (comme montré dans la
 36 [Figure 4.24](#) les deux diagrammes à gauche).

1 **Nombre d'états** Une autre métrique est le nombre d'états d'un diagramme états-transitions.
2 Cela permet d'évaluer ensuite dans les critères de qualités la complexité et la compréhension du
3 diagramme états-transitions à travers le nombre des états (selon si le nombre d'états est fini ou
4 non).

5 **Smell** La dernière métrique que nous considérons est une situation où le diagramme états-
6 transitions semble contenir une erreur possible, c'est-à-dire une possibilité d'incohérence ou d'er-
7 reur.

8 **Remarque** Les noms donnés aux états dans la table des états doivent être significatifs et
9 doivent représenter les conditions que les états modélisent. Notez également que le critère de
10 qualité le nombre d'états du diagramme (qui doit être fini mais également suffisant pour décrire
11 le système) est pris en compte dans le diagramme états-transitions résultat de la méthode de
12 spécification. En effet, les observateurs d'états utilisés dans la génération de la table des états
13 ont des valeurs finis.

14 Critères de qualité

15 En se basant sur les métriques (définies dans la [Section 4.7.1](#)), nous définissons dans cette sec-
16 tion des critères de qualité que nous utilisons ensuite pour évaluer la qualité des diagrammes états-
17 transitions et améliorer la compréhension de l'utilisateur vis-à-vis ces derniers ([Section 4.7.3](#)).

18 **Duplication des états** Nous définissons un premier critère de qualité qui est le nombre
19 d'états dupliqués (en se basant sur la métrique correspondante). Une hypothèse est que la qualité
20 ainsi que la compréhension du diagramme états-transitions sont meilleures en l'absence d'états
21 dupliqués. Il est possible que dans certains cas, il soit nécessaire de garder certains états dupliqués
22 (et cela dans le cas où il y a une influence sur la modélisation du système si les états sont
23 supprimés).

24 **Transitions avec le même événement** Le second critère que nous définissons dans cette
25 section est la présence de plusieurs transitions avec le même événement, issues de différents états
26 et allant vers un même état. La présence de plusieurs transitions de ce cas de figures pourrait
27 induire une difficulté à lire le diagramme ou à comprendre le sens des transitions. Donc le critère
28 de qualité serait l'absence de transitions avec le même événements et allant vers le même état
29 (dans le cas où cela n'influe pas sur la véracité de la modélisation du système).

30 **Transitions avec différents événements** Un autre critère de qualité que nous définissons
31 concerne la présence de différentes transitions avec différents événements et ayant le même état
32 source et le même état destination. Donc le critère de qualité serait un nombre minimal de ces
33 transitions qui permet une lisibilité meilleur et une compréhension plus facile.

34 **Pseudo-états initiaux et états finaux** Le dernier critère que nous définissons concerne
35 la présence de pseudo-états initiaux et d'états finaux dans un état composite. La présence de
36 pseudo-états initiaux dans les états composites est parfois importante pour la compréhension
37 du diagramme. Donc le critère de qualité serait la présence de pseudo-état initial dans un état
38 composite dans le cas où l'entrée à l'état composite n'est pas claire. Le même principe s'applique
39 pour les états finaux, car dans certains cas il est nécessaire d'avoir un état final dans un état

1 composite. Donc le critère de qualité serait la présence d'un état final dans un état composite si
2 la sortie de ce dernier n'est pas claire (avec l'absence de l'état final).

3 Améliorations

4 Nous proposons dans cette section des transformations dans le but d'améliorer la "qualité"
5 du diagramme états-transitions et en se basant sur les critères proposés dans la [Section 4.7.2](#).
6 Chaque transformation permet de respecter au mieux un critère de qualité et ainsi d'améliorer
7 la qualité du diagramme états-transitions. Les transformations sont présentées comme suit :

8 **Suppression de la duplication** La présence des mêmes¹ transitions en sortie allant vers
9 le même état permet de détecter les états dupliqués et ainsi de les supprimer. Nous distinguons
10 alors deux cas : le cas où les états dupliqués ont les mêmes transitions en entrée et le cas où les
11 états dupliqués ont des transitions en entrée différentes.

12 Pour chaque ensemble d'états dupliqués ayant les mêmes transitions en sortie (allant vers le
13 même état) et les mêmes transitions en entrée, nous supprimons l'ensemble de ces états pour le
14 remplacer par un seul état avec une seule transition en entrée et en sortie. Dans la [Figure 4.24a](#)
15 les deux états dupliqués sont **S3** et **S4** avec les mêmes transitions en sortie (allant vers le même
16 état **S5**) et en entrée (à partir de **S1**). Après transformation les deux états sont remplacés par
17 l'état **S3-4** et il y a une seule transition en entrée (car les deux transitions en entrée sont les
18 mêmes) et une seule transition en sortie vers l'état **S5**.

19 Pour chaque ensemble d'états dupliqués ayant les mêmes transitions en sortie (allant vers
20 le même état) et des transitions différentes en entrée, nous supprimons l'ensemble de ces états
21 pour le remplacer par un seul état et une seule transition en sortie. Les transitions en entrée sont
22 réorientées vers cet état. Dans la [Figure 4.24b](#) les deux états dupliqués sont **S3** et **S4** avec les
23 mêmes transitions en sortie (allant vers le même état **S5**) et des transitions différentes en entrée
24 (à partir de **S1**). Après transformation les deux états sont remplacés par l'état **S3-4** et les deux
25 transitions en entrée sont réorientées vers l'état **S3-4** (les transitions en entrée sont différentes).
26 Les deux transitions en sortie sont remplacées par une seule transition en sortie vers l'état **S5**
27 avec comme source l'état **S3-4**.

28 Suppression de la redondance des transitions par introduction de la hiérarchie

29 Pour chaque ensemble d'états ayant une transition avec le même événement vers le même état,
30 nous créons un état composite contenant ces états. Les transitions des états de départ seront
31 remplacées par une seule transition avec l'événement correspondant allant de l'état composite
32 qui contient ces états. Dans [Figure 4.25](#) il y a trois transitions avec le même événements **a** issues
33 de **S1**, **S2** et **S3**. Nous pouvons donc les factoriser en utilisant la hiérarchie par l'introduction
34 d'un état composite **S** qui contiendra les trois états (**S1**, **S2** et **S3**). Toutes les transitions avec
35 l'événement **a** seront remplacées par une transition issues de l'état composite **S** (avec l'événement
36 **a**) vers l'état **S4**.

37 Introduction des transitions avec plusieurs événements

38 Pour chaque ensemble de transitions ayant la même source et la même destination avec des événements, nous remplaçons
39 cet ensemble par une seule transition ayant plusieurs événements. Cette situation est envisageable
40 uniquement dans le cas où le regroupement des événements dans une seule transition ne complique
41 pas la lecture de la transition. Dans la [Figure 4.26](#) les transitions avec les événements **a**, **b**, **c**, **d** et

1. nous appellerons "mêmes transitions", les transitions qui sont étiquetées par le même événement, la même garde et la même action

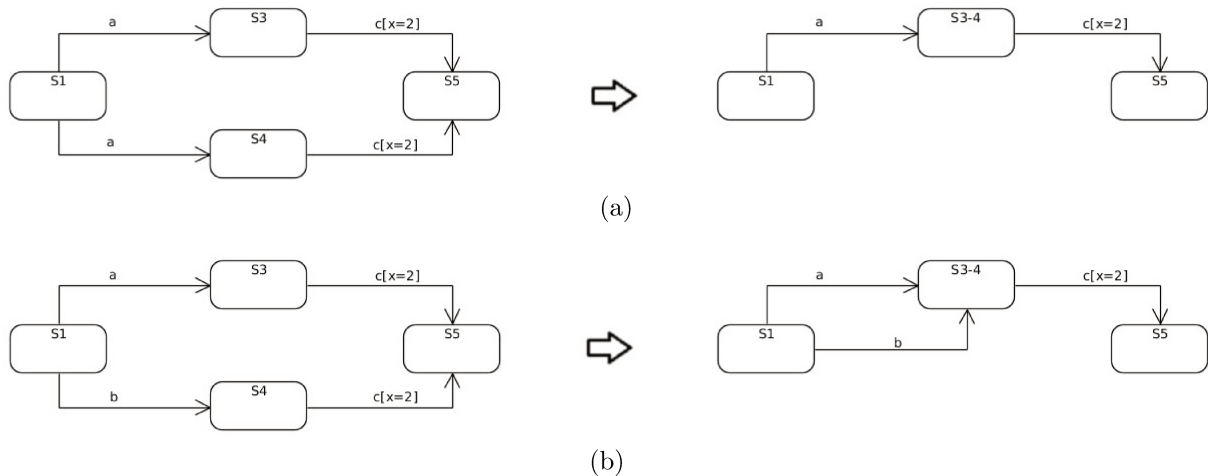


FIGURE 4.24 – Exemple de diagramme états-transitions avec et sans amélioration de la duplication

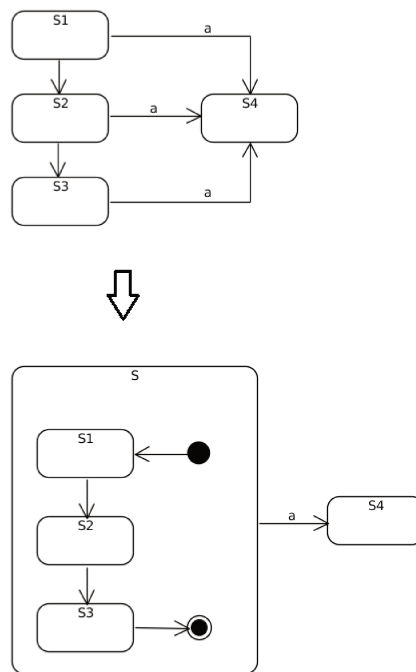


FIGURE 4.25 – Exemple d'introduction de la hiérarchie

- 1 e sont remplacées par la transition allant de l'état S1 vers S2 et ayant l'ensemble des événements
- 2 (c'est-à-dire a, b, c, d et e).



FIGURE 4.26 – Exemple d'introduction de transitions avec plusieurs événements

- 3 La situation inverse est possible, dans le cas où une transition avec un ensemble d'événements
- 4 est difficile à lire, nous la remplaçons par un ensemble de transitions avec chacune un événement

1 (ou plus dans le cas où cela ne complique pas la lecture de la transition).

2 **Pseudo-états initiaux et états finaux** Pour chaque état composite ayant un ensemble
 3 de transitions (avec ou sans événements) en entrée (l'ensemble doit contenir au moins une transi-
 4 tion), nous rajoutons un pseudo-état initial allant vers son état initial. Dans la [Figure 4.27](#) nous
 5 ajoutons un pseudo-état initial dans l'état composite S2 car il y a une transition avec comme
 destination cet état.



FIGURE 4.27 – Exemple d'ajout de pseudo-état initial

6
 7 Pour chaque état composite ayant un ensemble de transitions sans événements en sortie,
 8 nous rajoutons un état final lié au dernier état qui sera visité dans l'état composite. Dans la
 9 [Figure 4.28](#) nous ajoutons un état final dans l'état composite S2 car il y a une transition avec
 comme source cet état.



FIGURE 4.28 – Exemple d'ajout d'état final

10

11

12 Expérimentation

13 Dans le but de montrer l'efficacité de la prise en compte des critères de qualité (en appli-
 14 quant les améliorations présentées dans la [Section 4.7.2](#)), nous proposons une expérimentation
 15 avec plusieurs exemples. L'expérimentation consiste à définir plusieurs questionnaires contenant
 16 des exemples avec et sans améliorations. Le [Tableau 4.18](#) montre la liste des critères de qualités
 17 utilisés avec les exemples correspondants. Le symbole (-) signifie que l'exemple est sans amé-
 18 lioration et le symbole (+) signifie que l'exemple est avec amélioration. D'autres exemples sont
 19 donnés en [Annexe C](#).

Introduction hiérarchie	Multiples événements	Duplication d'états
barrière de péage (-)	threads [?] (-)	playlist (-)
barrière de péage (+)	threads [?] (+)	playlist (+)

TABLE 4.18 – Exemples utilisés dans l'expérimentation

Questionnaire Nous avons effectué une expérience en utilisant deux questionnaires (Annexe B) sur un groupe de 77 étudiants. Le premier questionnaire comporte 3 exemples (Figures 4.29a, 4.30 et 4.32) sans améliorations et pour chaque exemple 4 questions. Le second questionnaire comporte les mêmes exemples (Figures 4.29b, 4.31 et 4.33) avec améliorations et pour chaque exemple il y a 4 questions.

Le premier exemple (Figures 4.29a et 4.29b) est celui du contrôleur d'une barrière de péage avec deux versions : une sans amélioration et une avec amélioration (l'introduction de la hiérarchie et la suppression des transitions ayant le même événement). Dans la première version (Figure 4.29a) il y a plusieurs transitions avec le même événement (`blackout()`), qui sont issues de différents états et qui vont vers le même état. La seconde version (Figure 4.29b) les transitions avec l'événement `blackout()` sont remplacées par une seule transition avec le même événement et issue de l'état composite `carArrives()` (l'état composite qui va contenir tous les états sources des transitions en question).

Le second exemple (Figures 4.30 et 4.31) est celui du cycle de vie d'un thread avec deux versions : une sans amélioration et une avec amélioration (factorisations de transitions). Dans la première version (Figure 4.30) il y a plusieurs transitions avec différents événements issues du même état et qui vont vers le même état. Dans la seconde version (Figure 4.31) les transitions avec le même état source et le même état destination sont remplacées par une seule transition étiquetée avec tous les événements des transitions remplacées.

Le troisième exemple (Figures 4.32 et 4.33) est celui de la playlist avec deux versions : une sans amélioration et une avec amélioration (suppression des états dupliqués). Dans la première version (Figure 4.32) il y a plusieurs états dupliqués avec les mêmes transitions en sortie et des différentes transitions en entrée. Dans la seconde version (Figure 4.33) les états dupliqués sont remplacés par un seul état avec en sortie une seule transition et les transitions d'entrée (des états dupliqués supprimés) sont dirigées vers cet état.

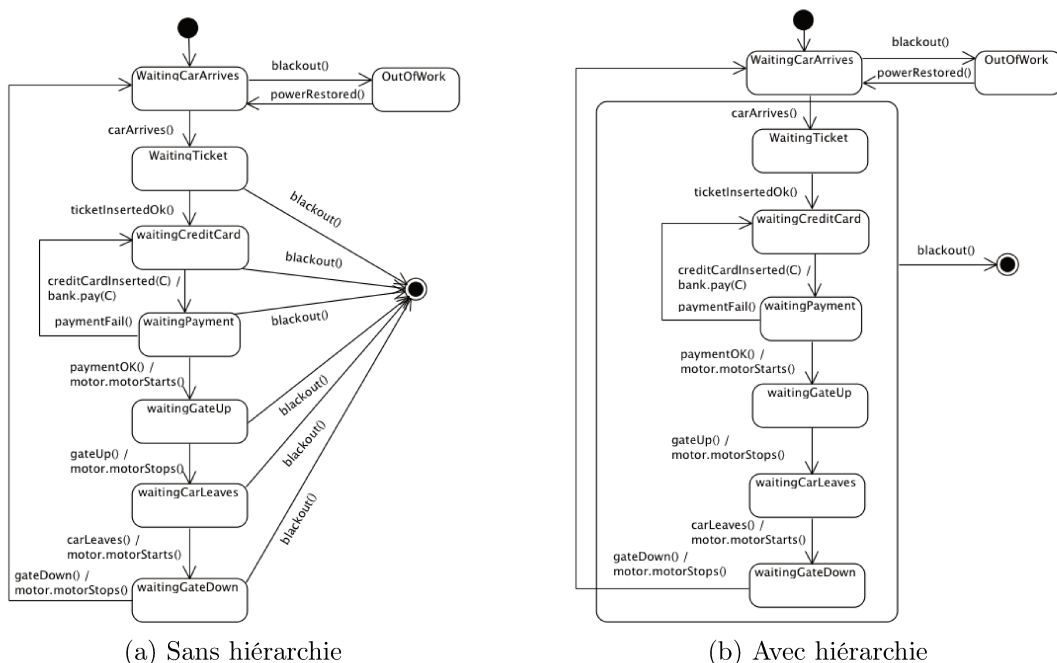


FIGURE 4.29 – Exemple de la barrière sans hiérarchie et avec hiérarchie

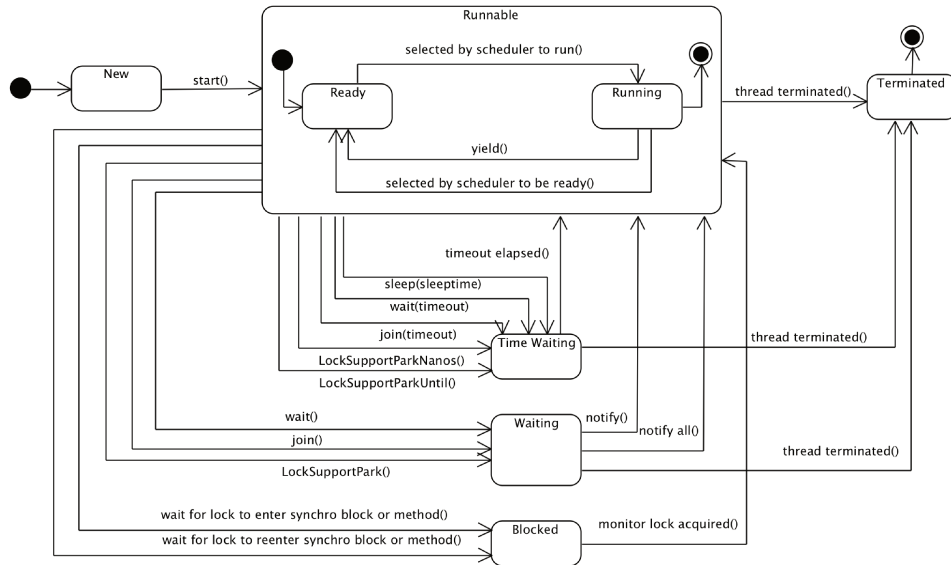


FIGURE 4.30 – Exemple du thread avec des événements dans des transitions différentes

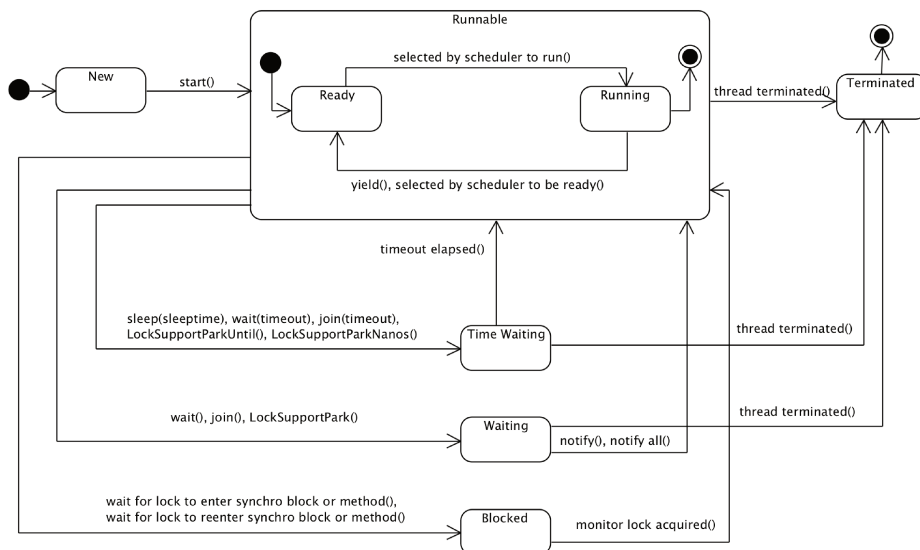


FIGURE 4.31 – Exemple du thread avec des événements dans une seule transition

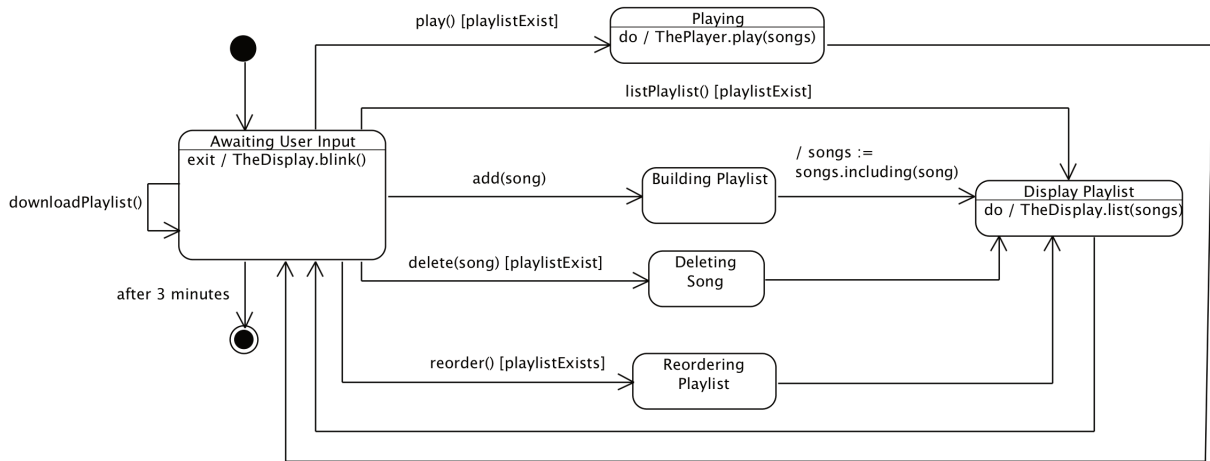


FIGURE 4.32 – Exemple de la playlist avec des états dupliqués

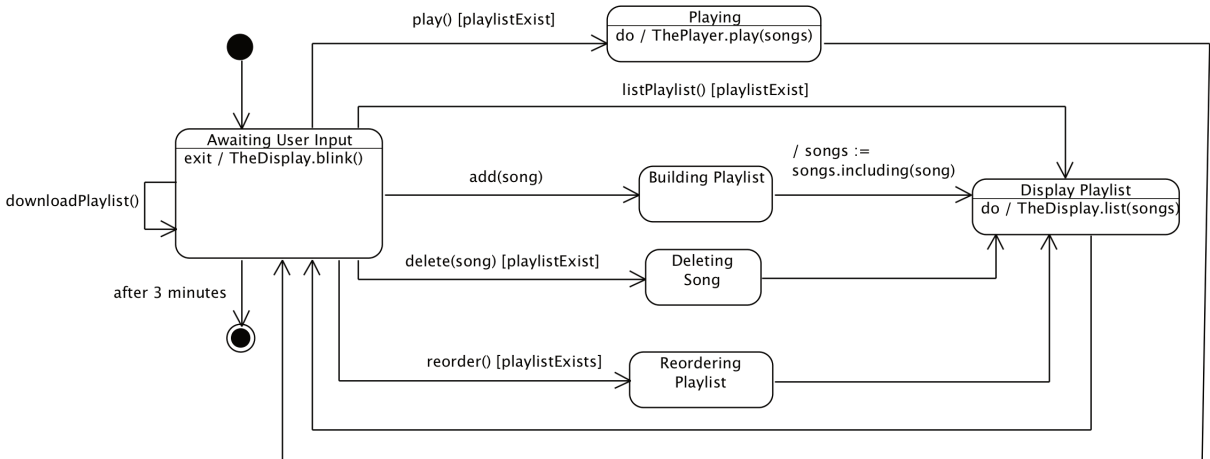


FIGURE 4.33 – Exemple de la playlist sans états dupliqués

1 Outil support de la méthode : *EasySM*

2 *EasySM* : première version

3 Nous avons présenté dans la [Section 4.3](#) une méthode de spécification permettant le passage
 4 entre la description textuelle du système vers son diagramme états-transitions correspondant.
 5 Gianna Reggio (Université de Gênes, Italie) et son étudiant Daniele De Gregori ont proposé un
 6 outil, nommé *EasySM*, qui met en œuvre cette méthode de spécification. *EasySM* offre à l'utili-
 7 sateur la possibilité de suivre les étapes de la méthode, de revenir en arrière en cas de besoin de
 8 modifications, et la mise à jour des données est faite automatiquement après chaque modification.
 9 La [Figure 4.34](#) montre l'interface de base de l'outil avec comme onglets les différentes étapes de
 10 la méthode. Il n'y a aucun projet visible et l'utilisateur doit créer un nouveau projet ou ouvrir
 11 un projet existant.

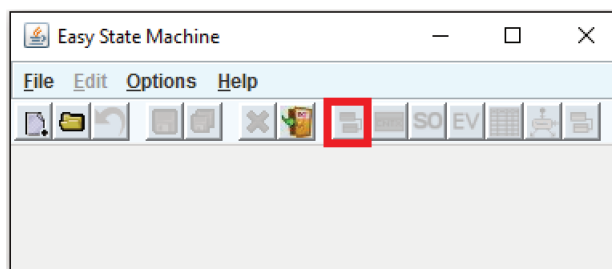


FIGURE 4.34 – Interface de base de *EasySM*

12 La méthode prend comme point de départ le diagramme de classes du système étudié dans
 13 lequel est sélectionnée une classe de contexte. L'utilisateur commence donc par créer le diagramme
 14 de classes (en cliquant sur l'onglet correspondant [Figure 4.34](#) qui devient actif après avoir créé
 15 ou ouvert un projet) à l'aide de l'outil et choisit la classe de contexte qui sera la classe principale
 16 pour le reste des étapes de la méthode.

17 La [Figure 4.35](#) montre le diagramme de classes du système de bibliothèque et le choix de la
 18 classe de contexte (l'exemple de la bibliothèque est présenté dans [?]). Chaque classe est présentée
 19 sous le format suivant dans l'outil : **CL Name** (où **Name** est le nom de la classe). Dans l'exemple de
 20 la bibliothèque il y a 4 classes : **Book**, **Copy**, **User** et **Date**. Chaque classe peut avoir des attributs
 21 et des opérations (par exemple, la classe **Book** a deux attributs : **title** et **author** qui sont de
 22 type chaîne de caractères). Il est possible dans cette phase d'ajouter des associations entre classes
 23 afin d'utiliser les différents attributs (ou opérations) dans les autres phases de la méthode. Dans
 24 l'exemple de la bibliothèque il y a une association **copyOf** entre la classe **Book** et **Copy**.

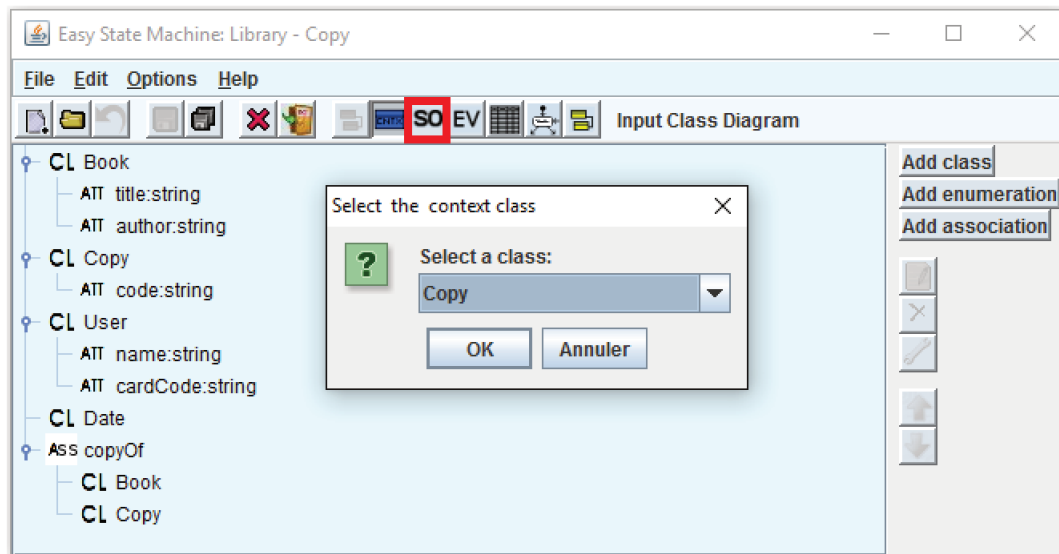
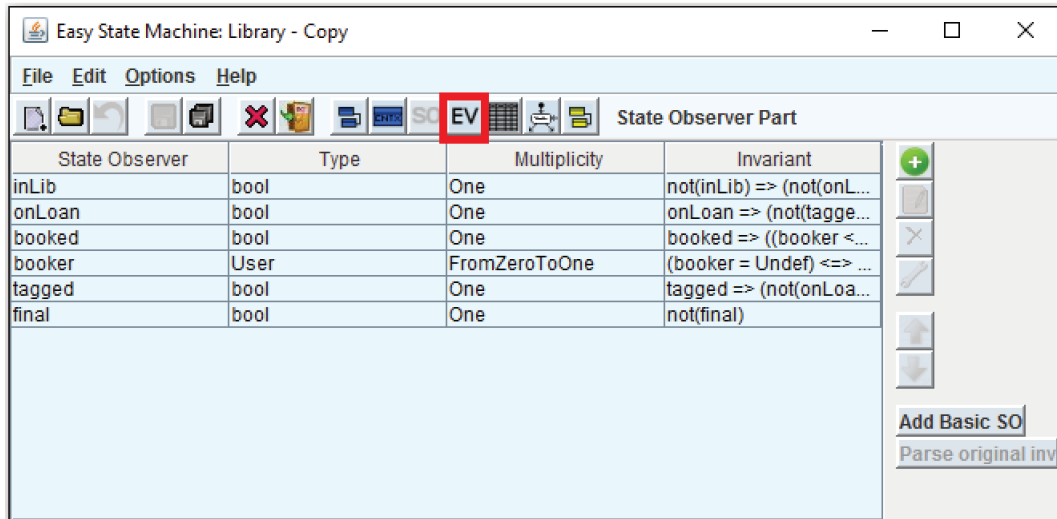


FIGURE 4.35 – Édition du diagramme de classes et de la classe de contexte dans *EasySM*

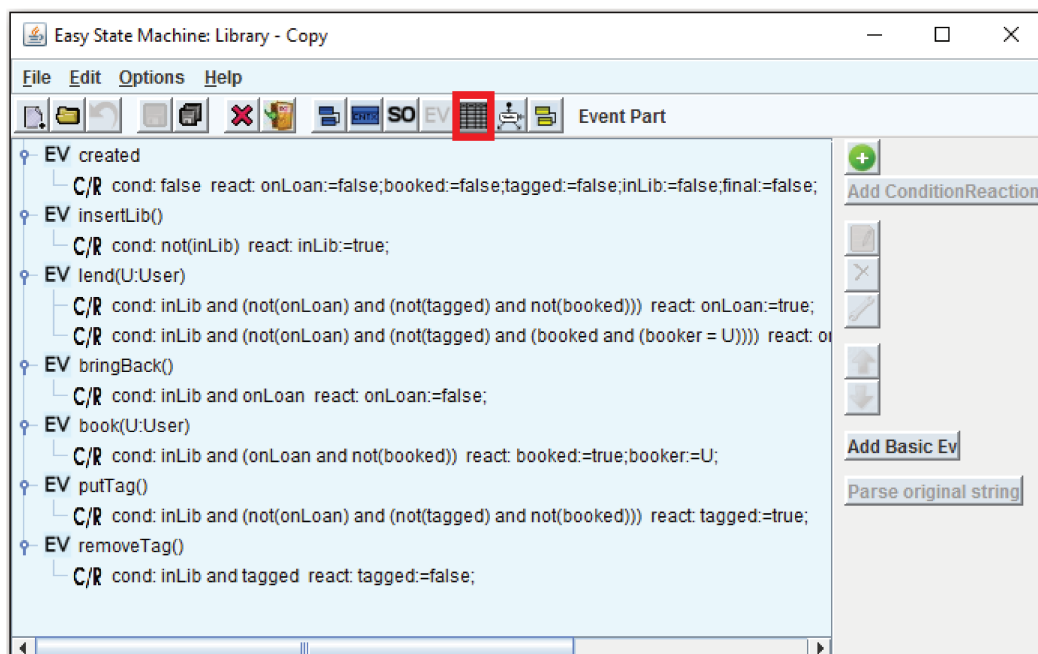
1 Une fois le diagramme de classes édité et la classe de contexte choisie, l'utilisateur engendre
 2 et édite les observateurs d'états (en cliquant sur l'onglet correspondant Figure 4.35). *EasySM*
 3 offre une fonctionnalité (représentée par le bouton *Add Basic SO*) de génération automatique
 4 d'une partie des observateurs d'états en se basant sur les attributs que les différentes classes
 5 contiennent. Cette fonctionnalité n'est pas obligatoire car les observateurs engendrés peuvent
 6 ne pas correspondre à ce que l'utilisateur souhaite ajouter. Dans la Figure 4.36 il n'y a aucun
 7 observateur d'états de base car l'utilisateur ne l'a pas demandé (car ce n'est pas pertinent). L'uti-
 8 lisateur pourra modifier les observateurs d'états engendrés mais aussi ajouter les observateurs
 9 d'états qui manquent.

10 La Figure 4.36 montre la phase d'édition des observateurs d'états dans l'outil *EasySM*.
 11 Chaque observateur d'état a un nom (dans la colonne **StateObserver**), un type (dans la co-
 12 lonne **Type**), une multiplicité (dans la colonne **Multiplicity**) et un invariant (dans la colonne
 13 **Invariant**). Le type de l'observateur d'état peut être un type prédéfini (tel qu'une chaîne de
 14 caractères **String**, un entier **Integer**, ...) ou un type défini par l'utilisateur (par exemple, l'ob-
 15 servateur d'état **booker** a comme type la classe **User**). La multiplicité d'un observateur d'état
 16 permet de savoir si un observateur d'état a toujours une valeur ou non. Dans le cas où l'ob-
 17 servateur a toujours une valeur alors la multiplicité sera **One** (par exemple, l'observateur d'état
 18 **onLoan** a toujours une valeur **true** ou **false** et donc sa multiplicité est **One**). Dans le cas où
 19 l'observateur d'état peut ne pas avoir de valeur alors sa multiplicité est ou bien **FromZeroToOne**
 20 ou bien **FromOneToStar** ou **Star**. Dans l'exemple, l'observateur d'état **booker** a comme type
 21 **User** et peut avoir une valeur, c'est-à-dire la présence d'un personne qui va emprunter un livre,
 22 ou non dans le cas où il n'y a pas d'emprunteur et c'est pour cette raison que sa multiplicité est
 23 **FromZeroToOne**. L'invariant est une expression booléenne qui permet de définir une condition
 24 sur l'observateur d'état.

FIGURE 4.36 – Édition des observateurs d'états dans *EasySM*

1 L'utilisateur peut revenir en arrière (c'est à dire à la phase d'édition du diagramme de classes
 2 [Figure 4.35](#)) afin de rajouter ou mettre à jour des éléments du diagramme de classes. L'étape
 3 suivante sera de définir les événements (en cliquant sur l'onglet correspondant [Figure 4.36](#)) et
 4 là aussi il est possible d'engendrer les événements de base automatiquement. *EasySM* offre une
 5 fonctionnalité (représentée par le bouton *Add Basic Ev*) de génération automatique d'une partie
 6 des événements (les événements évidents) en se basant sur les fonctions contenues dans les classes.

7 La [Figure 4.37](#) montre la phase d'édition des événements dans l'outil *EasySM*. Chaque évé-
 8 nement a un nom (représenté par `EVname()` où `Name` est le nom de l'événement), une précondition
 9 (représentée par `cond :`) et une réaction (représentée par `react :`). L'utilisateur peut donc défi-
 10 nir la condition qui doit être vérifiée quand l'événement se produit ainsi que la réaction après
 11 occurrences de l'événement.

FIGURE 4.37 – Édition des événements dans *EasySM*

1 L'étape suivante pour l'utilisateur pendant la manipulation de l'outil est la génération de la
 2 table des états (en cliquant sur l'onglet correspondant Figure 4.37). *EasySM* génère automati-
 3 quement la table des états (sachant que les états auront des noms aléatoires et c'est à l'utilisateur
 4 de les changer). L'utilisateur peut aussi décider de supprimer ou d'ajouter des observateurs de
 5 la table.

6 La Figure 4.38 montre la phase d'édition de la table des états dans l'outil *EasySM*. Les co-
 7 lonnes de la table correspondent aux observateurs d'états utilisés (par exemple, *inLib*, *onLoan*,
 8 ...) avec pour chaque colonne les valeurs possibles de l'observateur d'état. La colonne *name* re-
 9 présente les états possibles et impossibles. Un état impossible est celui dont la ligne (composée
 10 des valeurs des observateurs d'états) est en contradiction avec des invariants. Les noms états
 11 possibles seront générés automatiquement avec la forme *Statei* et l'utilisateur peut les éditer.

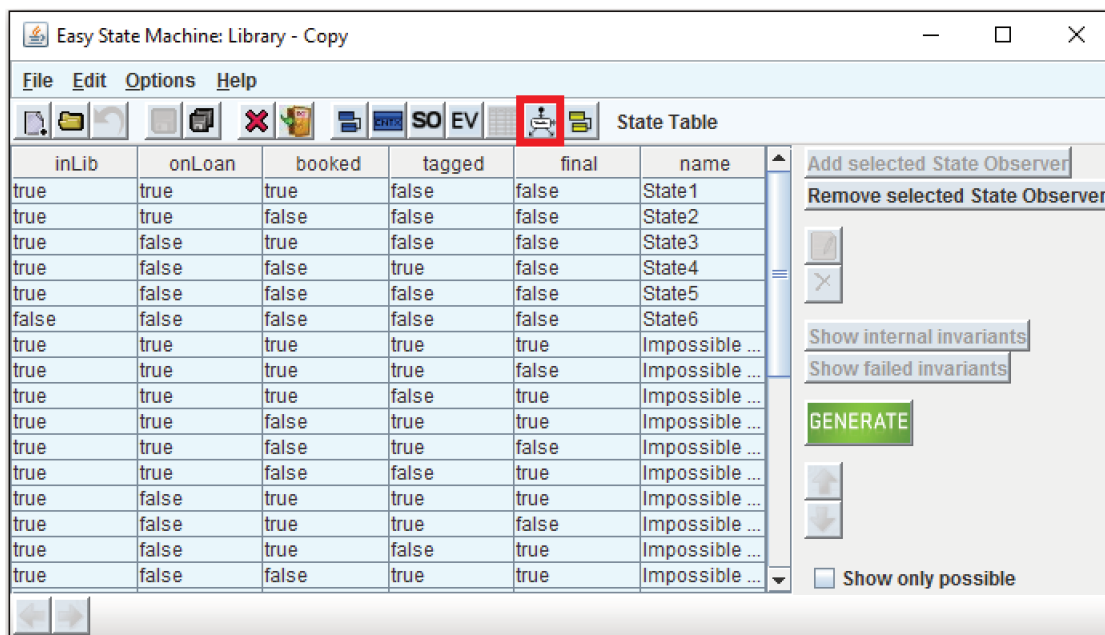


FIGURE 4.38 – Édition de la table des états dans *EasySM*

12 Une fois la table des états engendrée, l'utilisateur peut engendrer le diagramme états-transitions
 13 en cliquant sur l'onglet correspondant Figure 4.38 puis sur le bouton *Generate*. Le résultat de la
 14 génération est un tableau des états source et destination de chaque transition (chaque transition
 15 a une garde, un événement et une action). *EasySM* permet à cette phase d'engendrer aussi le rap-
 16 port contenant les différentes informations du diagramme. La Figure 4.39 montre le diagramme
 17 états-transitions dans l'outil *EasySM*.

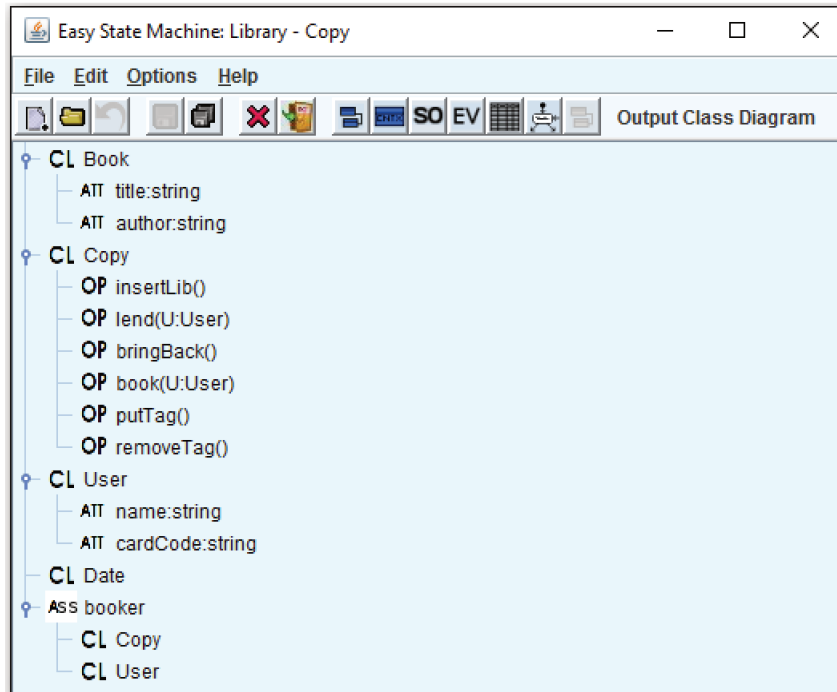
State Machine of Copy
 Built on: Thu Sep 08 19:49:07 CEST 2011
 Building time: 15 milliseconds

State Machine:

Source	Guard	Event	Effect	Target
Initial				NotInLibrary
NotInLibrary		insertLib()		Available
Available		lend(U:User)		OnLoanNotBooked
Booked	booker = U	lend(U:User)	booker:=Undef;	OnLoanNotBooked
OnLoanBooked		bringBack()		Booked
OnLoanNotBooked		bringBack()		Available
OnLoanNotBooked		book(U:User)	booker:=U;	OnLoanBooked
Available		putTag()		Tagged
Tagged		removeTag()		Available

FIGURE 4.39 – Diagramme états-transitions dans *EasySM*

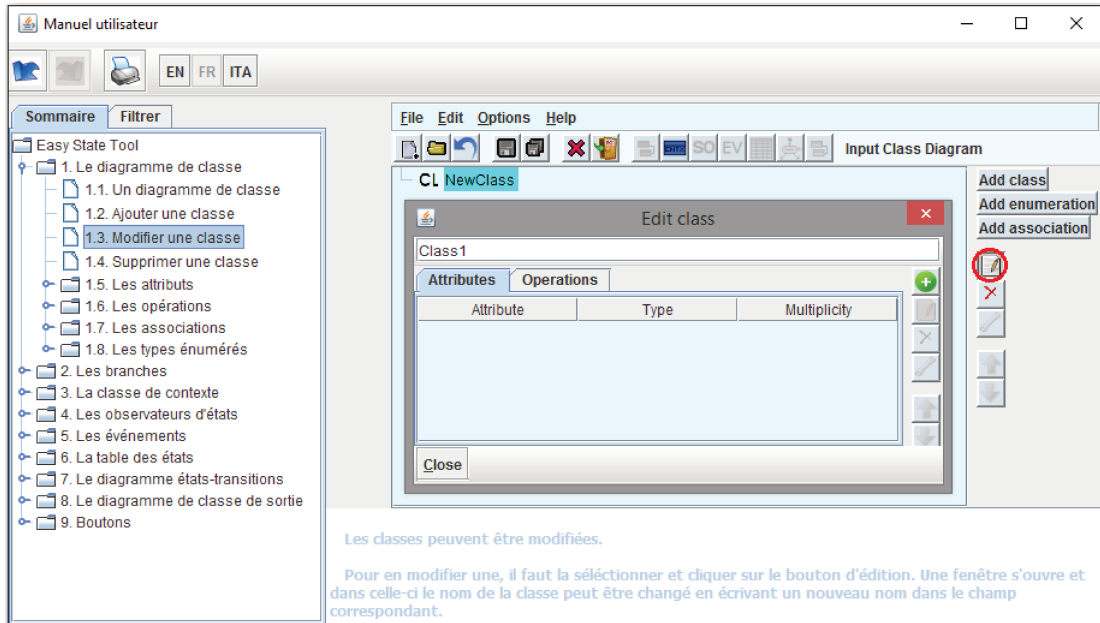
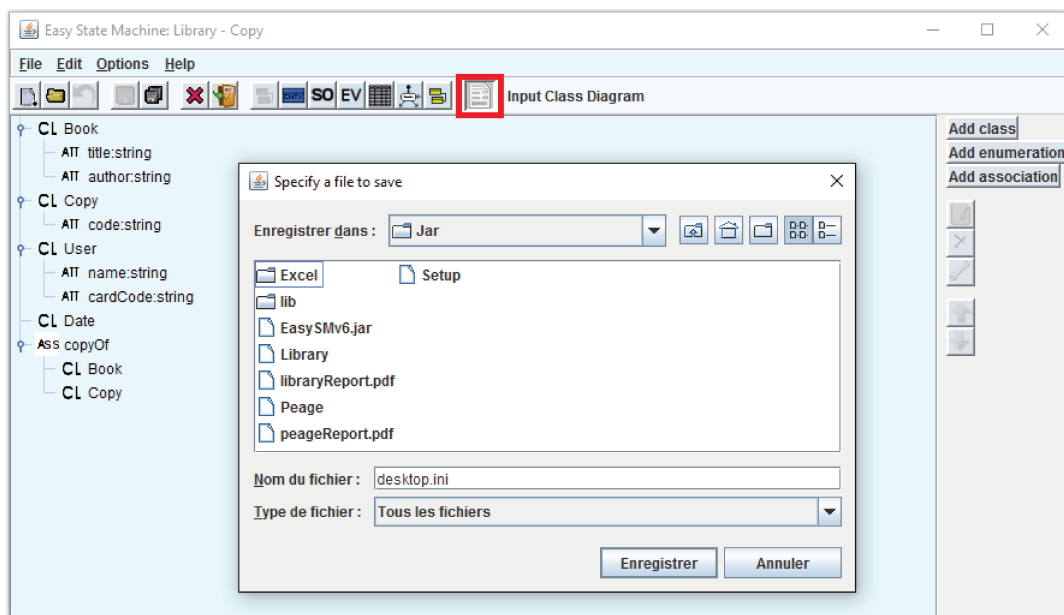
1 Le dernier onglet dans *EasySM* est la mise à jour du diagramme de classes (atteignable
 2 en cliquant sur l'onglet correspondant Figure 4.39) après la génération du diagramme états-
 3 transitions. Le diagramme de classes contiendra les événements sous forme de fonctions dans
 4 les classes, et les observateurs qui ne sont pas utilisés dans la table des états sont rajoutés (aux
 5 classes correspondantes) comme des attributs. La Figure 4.40 montre la mise à jour du diagramme
 6 de classes dans *EasySM*. Les événements, définis dans l'onglet d'édition des événements, sont
 7 rajoutés comme opérations dans la classe de contexte (par exemple, la classe de contexte *Copy*
 8 qui au départ n'avait pas de d'opérations (Figure 4.35) contient tous les événements comme des
 9 opérations). Chaque attribut ou association non utilisé dans les observateurs d'états ou invariants
 10 sera supprimé de sa classe lors de la mise à jour du diagramme de classes. Dans Figure 4.40
 11 l'attribut `code` de la classe *Copy* (Figure 4.35) est supprimé dans la mise à jour du diagramme
 12 de classes.

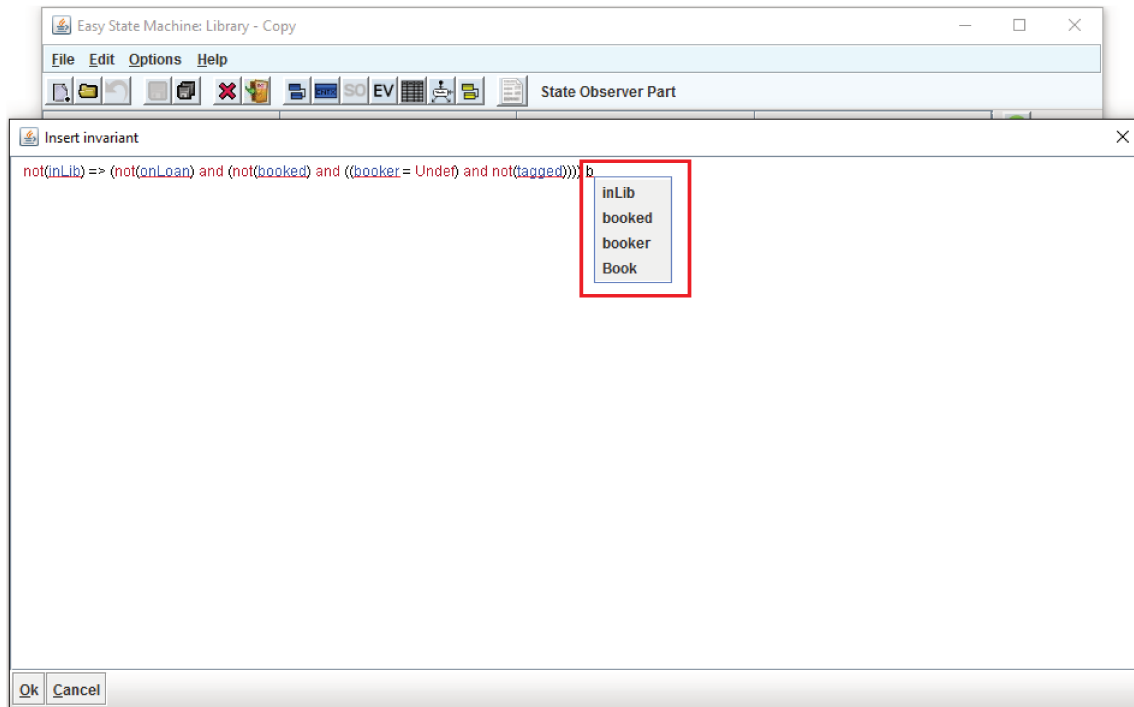
FIGURE 4.40 – Mise à jour du diagramme de classes dans *EasySM*

1 EasySM : améliorations

Nous avons encadré un projet de Master 1 afin d’améliorer l’outil *EasySM* (présenté dans la [Section 4.9.1](#)) et d’ajouter d’autres fonctionnalités. En plus des corrections de bugs dans l’édition des observateurs d’états et événements ainsi que la suppression des boutons inutiles, l’amélioration apportée à l’outil de base est la suivante :

- Une aide en ligne (en Anglais et en Français) afin d’aider toute personne utilisant *EasySM* (voir la [Figure 4.41](#)). L’aide en ligne est accessible à partir du menu **Help** ([Figure 4.40](#)). La partie gauche de la [Figure 4.41](#) représente le sommaire avec les différents éléments de l’outil. La partie droite représente l’explication de l’élément sélectionné dans le sommaire (image avec l’explication en dessous). L’outil prend en charge l’anglais, le français et l’italien (*EN*, *FR* et *ITA* [Figure 4.41](#)) et il y a la possibilité d’impression.
- Un bouton *Export Rapport* (voir [Figure 4.42](#)) qui permet d’exporter un fichier pdf contenant toutes les informations à propos du projet en cours (et même des projets associés). L’outil propose un emplacement par défaut où il est possible de stocker le fichier mais l’utilisateur peut choisir un autre emplacement. Un exemple de rapport engendré par cette fonctionnalité est en [Annexe A](#).
- Un menu déroulant pour faciliter la saisie des types pour les invariants et conditions/réactions et qui indique à l’utilisateur les types existants dans le cas où il tape une lettre (voir la [Figure 4.43](#)). Dans la [Figure 4.43](#) l’utilisateur tape la lettre *b* et l’outil lui propose les différents contenant cette lettre (tels que *booked*, *booker*, ...).
- Rédaction d’un manuel utilisateur contenant des explications détaillées sur l’outil, son fonctionnement, comment l’utiliser et d’un manuel d’installation.

FIGURE 4.41 – Manuel utilisateur dans *EasySM*FIGURE 4.42 – Le bouton *Export Rapport* dans *EasySM*

FIGURE 4.43 – Menu déroulant des propositions de types dans *EasySM*

1 Conclusion

2 Nous avons présenté dans ce chapitre une méthode de spécification en utilisant les diagrammes
3 états-transitions. Afin d'améliorer cette méthode, nous avons apporté une extension en termes
4 de structures mais aussi en terme de syntaxe; et pour montrer comment l'utiliser nous l'avons
5 appliquée sur l'exemple du paiement de parking. Enfin, nous avons présenté l'outil *EasySM* qui
6 implémente cette méthode de spécification ainsi que les améliorations que nous avons apportées
7 à ce dernier.

8 Une fois que l'utilisateur engendre le diagramme états-transitions en utilisant la méthode de
9 spécification, il pourra appliquer notre traduction vers les réseaux de Petri colorés pour faire de
10 la vérification. Cette traduction est présentée dans le [Chapitre 5](#) qui est le chapitre suivant de
11 ce document.

Chapitre 5

Contribution : Transformation des diagrammes états-transitions vers les réseaux de Petri colorés

Contents

5.1	Introduction	121
5.2	Formalisation et hypothèses	122
5.3	Schéma général	125
5.4	Places et transitions : nommage	128
5.5	Algorithme de traduction	128
5.6	L'ajout des comportements au code des transitions	132
5.7	Application de l'algorithme à des cas simples	134
5.8	Au-delà de notre traduction	140
5.9	Sélection des transitions	142
5.10	Conclusion	154

Introduction

Nous présentons dans cette partie notre traduction des diagrammes états-transitions vers les réseaux de Petri colorés. Nous commençons par présenter ce que nous considérons comme éléments syntaxiques dans UML (Section 5.2) ensuite le schéma général de la traduction (Section 5.3). Nous décrivons ensuite une méthode pour nommer les places et les transitions du réseau de Petri coloré résultat de la traduction (Section 5.4). Nous explicitons notre traduction dans la (Section 5.5), ainsi que la gestion de la hiérarchie des comportements d'entrée et de sortie (Section 5.6). Nous appliquons notre traduction sur des exemples (Section 5.7). Enfin nous montrons comment étendre l'ensemble des éléments syntaxiques considérés dans cette traduction (Section 5.8). Nous présentons l'extension de notre traduction afin de prendre en compte les priorités de franchissement des transitions ainsi que la notion de conflit entre transitions dans la Section 5.9.

1 Formalisation et hypothèses

2 Nous présentons dans cette sous-section les éléments des diagrammes états-transitions que
 3 nous prenons en compte dans notre traduction avec les hypothèses appliquées sur la définition
 4 d'UML. L'ensemble des éléments des diagrammes états-transitions que nous prenons en compte
 5 est le suivant :

- 6 • États simples, états finaux, états composites simples, états composites orthogonaux.
- 7 • Les comportements d'entrée, de sortie et do, impliquant des variables.
- 8 • Pseudos-états initiaux, histoires plats, fork et join.
- 9 • La hiérarchie des comportements et des états.
- 10 • Transitions : simples, avec événement, avec garde et comportement, locaux, externes, inter-
 11 niveau et haut niveau.

12 Notez que nous discuterons de comment supprimer les hypothèses ainsi que comment prendre
 13 en compte le reste des éléments dans la section [Section 5.8](#).

14 **États** Nous prenons en compte deux types d'états : simples et composites. Les états subma-
 15 chine ne sont pas considérés dans notre traduction.

16 **Comportements** Les comportements seront représentés par b (la même représentation dans [?,
 17 ?]) qui correspond à l'expression du comportement en cours et une fonction f pour exprimer les
 18 changements appliqués sur les variables du système (l'ensemble des variables sera noté par \mathcal{V}).
 19 Cette représentation prend en compte à la fois les comportements exprimés avec des expressions
 20 (par exemple, `fadein` dans le comportement d'entrée de l'état `PLAYING` dans la [Figure 6.1](#) et les
 21 comportements exprimés avec les modifications des variables (par exemple, `track ++` dans le
 22 comportements associé à la transition allant de `BUSY` vers lui même, [Figure 6.1](#)). L'ensemble de
 23 tout les comportements sera noté par \mathcal{B} ; *none* représente l'absence d'expressions de comporte-
 24 ments.

25 Le comportement sera noté alors par $(b, f) \in ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$, c'est-à-dire une
 26 expression de comportement, et une liste ordonnée de fonction dans \mathcal{F} , telle que \mathcal{F} est un
 27 ensemble de fonctions assignées à une variable de \mathcal{V} une valeur qui dépend de la valeur des
 28 autres variables. Nous assumons qu'un comportement do est un comportement atomique qui peut
 29 s'exécuter selon le nombre de fois désiré. Nous discutons cette hypothèse dans la [Section 5.8.1](#).
 30 Seul les états simples peuvent avoir un comportement do, une autre hypothèse que que nous
 31 discutons dans la [Section 5.8.1](#).

32 **Pseudo-état initial** Nous supposons que pour chaque région, il est nécessaire d'avoir un seul
 33 pseudo-état initial (direct) ayant une seule transition sortante. En accord avec la spécification
 34 [?], nous considérons que l'état actif du système ne peut être un pseudo-état initial. Notez que
 35 cela est un choix de modélisation : s'il y a le besoin de modéliser la situation où le système
 36 peut rester dans un pseudo-état initial, il suffit d'ajouter un état entre le pseudo-état initial et
 37 son successeur direct. Rappelons que la transition sortante d'un pseudo-état initial peut avoir
 38 un comportement, cependant afin garder notre algorithme simple nous ne prenons pas cela en
 39 compte.

40 Soit la fonction $init(\mathbf{s})$ qui retourne l'ensemble des sous-états directes destination d'une
 41 transition allant d'un pseudo-état initial, où \mathbf{s} est un état simple. De la même façon, $init^*(\mathbf{s})$

1 retourne l'ensemble de tous les sous-états simples (directs ou indirects) qui sont une destination
 2 pour une transition allant d'un pseudo-état initial. La fonction renvoie $\{\mathbf{s}\}$ si \mathbf{s} est un état simple.
 3 Par extension, nous notons $init^*(SMD)$ l'ensemble de tous les états simples (directs ou indirectes)
 4 qui sont destination d'une transition allant d'un pseudo-état initial racine dans le diagramme
 5 états-transitions SMD .

6 **Pseudo-état histoire** Dans ce travail, nous considérons uniquement les pseudo-états histoire
 7 plats (qui sont présentés dans [Section 2.2.1](#)), les pseudo-états histoire profonds ne sont pas
 8 considérés.

9 De plus, nous prenons pas en compte la transition par défaut du pseudo-état histoire (la
 10 transition qui permet d'indiquer le comportement du système dans le cas où l'état composite
 11 n'aurait jamais été entré, ou dans le cas où l'état final est le dernier état visité). Soit le pseudo-état
 12 histoire H , nous notons $r(H)$ la région qui contient comme sous-état directe le pseudo-état histoire
 13 H . La transition dont H est la destination peut être n'importe quel type de transitions (originaire
 14 d'un état simple ou composite, une transition locale ou externe ou inter-niveau, pseudo-état join).
 15 Afin de conserver un algorithme de traduction simple, nous prenons pas en compte les transitions
 16 fork dont l'un des états destinations est le pseudo-état histoire. La suppression de l'hypothèse
 17 peut être faite facilement, cependant le prix sera la complexité niveau algorithmique.

18 **État final** "Each region $[\dots]$ may have its own initial Pseudostate as well as its own Final-
 19 State." [?, Section 14.2.3.2 p.304] : nous permettons zéro ou un seul état final dans chaque région
 20 d'un état composite. Nous définissons la fonction $final(\mathbf{s})$ qui renvoi l'ensemble des états finaux
 21 directs de toutes les régions de l'état composite \mathbf{s} . De la même manière, la fonction $final^*(\mathbf{s})$
 22 renvoi l'ensemble des états finaux (directs ou indirects) de toutes les régions de l'état composite
 23 \mathbf{s} , ou $\{\mathbf{s}\}$ si \mathbf{s} est un état simple.

24 **Variables** Nous prenons en compte tout type de variable dans les comportements et les gardes
 25 des transitions. De tel variables (Booléen, entiers, listes, etc.) sont souvent rencontrées dans la
 26 pratique (*e.g.* [?, ?, ?]).

27 **Pseudo-états fork et join** Concernant les pseudo-états fork et join, nous proposons une
 28 représentations qui respecte la sémantique de la spécification (sans avoir d'impact dessus). Consi-
 29 dérons le pseudo-état join dans la [Figure 2.1](#) qui relie les états $S21$ et $S23$ à l'état final racine.
 30 Dans cette situation, la spécification UML considère un seul pseudo-état (le pseudo-état join) et
 31 trois transitions, une originaire du pseudo-état join et deux allant vers le pseudo-état join. En
 32 revanche, nous considérons une seule transition avec plusieurs états sources (pour le pseudo-état
 33 join) et plusieurs états destinations (pour le pseudo-état fork). Ensuite, les pseudo-états fork et
 34 join ne seront pas formalisés dans la [Définition 2](#).

35 **Transitions** Nous prenons en compte les transitions locales et externes ; nous prenons en
 36 compte également les transitions inter-niveaux avec une restriction concernant la concurrence :
 37 si la transition traverse la bordure de l'état source ou destination, alors cet état ne doit pas être
 38 un état composite orthogonal.

39 Nous réutilisons le concept de [?] d'état niveau (ou *level state* noté par $sLevel$). L'état niveau
 40 d'une transition allant de \mathbf{s}_1 à \mathbf{s}_2 est l'état le plus intérieur dans la structure hiérarchique du
 41 diagramme états-transitions qui contient la transition. Par exemple, l'état niveau de la transition
 42 allant de l'état `PLAYING` vers `PAUSED` dans la [Figure 6.1](#) est l'état `BUSY`.

L'état niveau est le SMD dans le cas où l'état source et l'état destination d'une transition sont des états racines. Par exemple, considérons la transition de BUY vers NONPLAYING étiquetée avec l'événement `stop` dans la Figure 6.1 : l'état niveau de cette transition est le diagramme états-transitions lui-même. Le concept d'état niveau nous permet de différencier entre une transition locale et une transition externe. Par exemple, dans la Figure 2.1, l'état niveau de la transition étiquetée avec l'événement `b` vers le pseudo-état histoire de `S1` est `S1` (car cette transition est locale), tandis que l'état niveau de la transition étiquetée avec l'événement `c` est `S`.

Formalisation d'un SMD Nous formalisons la syntaxe des diagrammes états-transitions comme ci-dessous. Notre définition de la syntaxe des diagrammes états-transitions ne diffère pas de celle d'UML, à l'exception de certaines constructions syntaxiques qui ne sont pas prises en compte, et quelques hypothèses que nous avons faite. Cependant, notre représentation de la syntaxe diffère afin de faciliter la traduction. Par exemple, les pseudo-états `fork` et `join` dans notre définition sont des transitions complexes (avec plusieurs états sources et destinations), alors que dans la spécification d'UML les pseudo-états `fork` et `join` sont juste des pseudo-états, c'est-à-dire des nœuds.

Definition 2 (Diagramme états-transitions). *Un diagramme états-transitions est un tuple*

$$SMD = (\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \text{SubStates}, \mathcal{T}), \text{ où}$$

1. \mathcal{S} est l'ensemble des états (incluant les pseudo-états),
2. \mathcal{B} est l'ensemble des expressions des comportements,
3. \mathcal{E} est l'ensemble des événements,
4. $\mathcal{V} = \{V_1, \dots, V_{N_V}\}$ est l'ensemble des variables N_V (pour certaines valeurs $N_V \in \mathbb{N}$),
5. $\mathcal{P} : \mathcal{S} \rightarrow Pr$ associe à chaque état une seule propriété dans $Pr = \{isSimpleNotFinal, isComposite, isFinal, isHistory\}$,
6. $\mathcal{N} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ associe pour chaque état un comportement d'entrée, c'est-à-dire une liste ordonnée de fonctions dans \mathcal{F} , où \mathcal{F} est l'ensemble des fonctions qui assignent à une variable de \mathcal{V} une valeur qui dépend de la valeur des autres variables,
7. $\mathcal{X} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ associe pour chaque état un comportement de sortie,
8. $\mathcal{D} : \mathcal{S} \rightarrow ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ associe pour chaque état un comportement `do`,
9. $\text{SubStates} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ associe pour chaque état un ensemble de sous-états directs,
10. \mathcal{T} est l'ensemble des transitions de la forme $\mathbf{t} = (\mathbf{S}_1, e, g, (b, f), sLevel, \mathbf{S}_2)$, où
 - $\mathbf{S}_1, \mathbf{S}_2 \subseteq \mathcal{S}$ sont, respectivement, les états sources et destinations avec les contraintes suivantes :
 - \mathbf{S}_1 ou \mathbf{S}_2 peuvent contenir exactement un état – dans le cas de transitions simples entre deux états – ou plus d'un seul état – dans le cas d'une transition `fork` ou `join` ;
 - Si \mathbf{S}_1 (respectivement \mathbf{S}_2) contient plus d'un état, chacun de ces états doit être un sous-état direct d'une région différente du même état composite, et que cet état composite ne doit pas avoir plus de régions que le nombre d'états dans \mathbf{S}_1 (respectivement \mathbf{S}_2) ;

- 1 • $e \in \mathcal{E} \cup \{noEvent\}$, où *noEvent* est une valeur spéciale qui dénote l'absence d'événement dans la transition,
- 2
- 3 • g est la garde (c'est-à-dire une expression booléenne sur les variables de \mathcal{V}),
- 4 • $(b, f) \in ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ est le comportement à exécuter pendant le franchissement de la transition, et
- 5
- 6 • $sLevel \in \mathcal{S}$ est l'état niveau qui contient la transition, tel que chaque état contenu
- 7 dans $\mathbf{S}_1 \cup \mathbf{S}_2$ est un sous état (avec une possibilité qu'il soit direct) de $sLevel$.

8 Notez que, dans la plus part des cas, $sLevel$ de τ est $LCA(\mathbf{S}_1, \mathbf{S}_2)$, tel que “The operation
 9 $LCA(s1, s2)$ returns the Region that is the least common ancestor of Vertices $s1$ and $s2$, based on
 10 the StateMachine containment hierarchy.” ([?, p.356]). Un exemple où la fonction LCA ne couvre
 11 pas tous les cas est la transition entre l'état \mathbf{S}_1 et son pseudo-état histoire \mathbf{H} dans la [Figure 2.1](#).
 12 En effet, la fonction $LCA(\mathbf{S}_1, \mathbf{H})$ renvoie \mathbf{S}_1 , tandis que le résultat doit être la machine à états
 13 \mathbf{S} . La fonction $sLevel$ pour la même transition (dans notre traduction) renvoie \mathbf{S} (la machine à
 14 états), car $sLevel$ permet de capturer le cas d'une transition externe.

15 Notez que les fonctions présentées précédemment dans cette section (par exemple, *init**,
 16 *final**, etc.) peuvent être définies en utilisant les éléments du tuple donnés dans [Définition 2](#).

17 Schéma général

18 Nous présentons dans cette section une vue générale de notre traduction des diagrammes
 19 états-transitions vers les réseaux de Petri colorés. Nous notons *transition SMD* pour désigner
 20 les transitions d'un diagramme états-transitions, et *transition CPN* pour désigner les transitions
 21 d'un réseau de Petri coloré.

22 **Idée générale** Nous définissons un schéma général de traduction où les états simples (incluant
 23 les états finaux) sont traduits vers des places, et où les transitions *SMD* sont traduites vers des
 24 transitions *CPN*. En effet, chaque étape d'exécution d'une transition *SMD* (*run-to-completion*
 25 *step*) est encodée par une seule transition *CPN* dans notre traduction, ce qui encode la totalité
 26 de la phase d'exécution d'une transition *SMD*. Nous pensons que c'est une solution élégante,
 27 et qui diffère des solutions précédentes (e.g. [?, ?, ?]), où certaines situations d'*entrelacement*
 28 entre deux transitions indépendantes peuvent se produire. Notez que notre solution prend en
 29 compte l'*entrelacement* entre comportements d'entrée ou de sortie des états courants d'une même
 30 transition.

31 Notre traduction est divisée en deux parties : la première partie (voir la [Section 5.5](#)) concerne
 32 la traduction des états et de leurs comportements ainsi que les transitions sans prendre en
 33 compte l'algorithme de sélection des transitions (fourni par l'OMG dans la spécification). Cette
 34 première partie est implémentée en utilisant l'[Algorithme 1](#). La seconde partie concerne la prise
 35 en compte de l'algorithme de sélection des transitions. Cette partie est implémentée en utilisant
 36 l'[Algorithme 2](#) qui implémente les priorités d'exécution des transitions ainsi que la gestion des
 37 conflits.

38 Enfin, bien que nous prenions en compte la gestion des comportements induite par la hiérarchie
 39 et l'utilisation des états composites, nous ne les traduisons pas sous une forme hiérarchique.
 40 En effet, dans notre traduction, les états composites ne sont pas codés par des places (seulement
 41 les états simples le sont). Ceci est en contraste avec d'autres approches qui utilisent les réseaux
 42 de Petri (colorés) hiérarchiques (e.g. [?]). La préservation de la hiérarchie (la représentation
 43 graphique) dans le formalisme de destination est l'objet de travaux futurs.

Comportements d'entrée et de sortie Rappelons que nous utilisons une abstraction des comportements exprimée par des expressions (b, f) , où b est l'expression du comportement et f est la liste des fonctions qui modifient les valeurs des variables globales (voir dans la [Section 5.2](#), page 122). La traduction des fonctions ne crée pas de difficulté particulière (comme expliqué dans la section [Section 5.6](#)) : chaque modification des variables *SMD* à travers une transition *SMD* sera traduite par la même modification des variables globales *CPN* correspondantes dans la partie code de la transition *CPN* encodant la transition *SMD*. À chaque entrée ou sortie d'un état à travers une transition *SMD*, nous ajoutons le comportement d'entrée ou de sortie correspondant (s'il existe) dans le code associé à la transition *CPN*. Cependant, par souci de simplicité, nous n'utilisons que la fonction f qui modifie les valeurs des variables (l'expression du comportement b ne sera pas utilisée). Les travaux tels que [?, ?] prennent en compte l'expression du comportement b . Nous discutons comment les réintroduire dans notre traduction dans la [Section 5.8](#).

Comportement do Nous traduisons les comportements *do* par une transition *CPN* (plus de détails dans la [Section 5.6](#)), où le code associé à cette transition comporte la fonction f encodant le comportement. Nous rajoutons ensuite deux arcs l'un de la place de l'état vers la transition du comportement, et l'autre de la transition du comportement vers la place de l'état. Cela est conforme à notre hypothèse que le comportement *do* peut être exécuté autant de fois que souhaité ([Section 5.2](#)). Nous discutons comment considérer une variante de cette hypothèse dans la [Section 5.8](#). Comme pour les comportements d'entrée et de sortie nous n'utilisons pas l'expression du comportement b .

États composites et pseudo-états Contrairement aux états simples, les états composites et les pseudo-états (tels que *fork*, *join*, *initial* ou pseudo-état histoire) ne sont pas traduits vers des places *CPN* (plus de détails dans la [Section 5.7](#)). Rappelons que la différence entre un état simple et un pseudo-état est le fait que l'état courant du système peut être un état mais non pas un pseudo-état. Par conséquent, le fait que les pseudo-états ne sont pas explicitement traduits vers des places *CPN* correspond au fait qu'un état actif du système d'un *SMD* est un état simple (ou un ensemble d'états simples). Cependant, leurs comportements et leurs sous-états (dans le cas des états composites) sont traduits vers des places ou des transitions.

D'une manière similaire, les pseudo-états initiaux ne sont pas traduits explicitement.

Comme les pseudo-états *fork* et *join* sont formalisés sous forme de transitions avec un ensemble d'états sources et un ensemble d'état cible (voir [Définition 2](#), [Section 5.2](#)), ils ne nécessitent pas un traitement particulier : chaque pseudo-état *fork* (ou *join*) sera représenté par une transition *CPN* qui reliera les places encodant les états sources et les états cibles.

Pseudo-état histoire Rappelons que lorsqu'une transition a comme cible un pseudo-état histoire plat (*shallow history pseudostate*) dans un état composite, l'état actif est le dernier état visité dans l'état composite (plus de détails à propos de la traduction du pseudo-état histoire avec exemple dans la [Section 5.7.6](#)). Cela est effectué en utilisant une variable globale (une par pseudo-état histoire) qui est mise à jour avec l'état actif courant à chaque fois que la place encodant cet état est entrée (ou l'un de ses sous-états dans le cas d'un état composite). La mise à jour est faite dans le code de n'importe quelle transition allant vers cette place.

Variation Nous utilisons le concept de variables globales que nous définissons dans la formalisation des réseaux de Petri colorés (voir [Section 2.3](#)) afin d'encoder la valeur des variables *SMD*. Les variables globales dans les réseaux de Petri colorés peuvent être lues dans les gardes et mises à jour à travers les transitions, avec le même principe que pour les variables *SMD*.

1 **Gestion des transitions** Le but principal de notre traduction est d'encoder les transitions
 2 entre les états composites avec différentes régions (les états orthogonaux), en particulier les
 3 transitions de haut niveau, les pseudo-états fork/join et les transitions avec événements (dans
 4 la [Section 5.7](#) nous expliquons la traduction des transitions avec événements et sans événements
 5 avec plusieurs états simples ou composites). Chaque transition *SMD* peut en effet correspondre
 6 à un nombre de transitions *CPN*. Par exemple, dans la [Figure 2.1](#) la transition *SMD* étiquetée
 7 avec l'événement *a* correspond à 18 possibilités pour sortir de *S1* (3 états actifs possibles dans la
 8 région supérieure de *S1*, multiplié par 4 dans *S13* plus 2 autres dans la région inférieure de *S1*),
 9 par conséquent à 18 transitions *CPN* :

- 10 • *S1_a_S2_1* pour la combinaison $\langle S11, S131 \rangle$
- 11 • *S1_a_S2_2* pour la combinaison $\langle S11, S132 \rangle$
- 12 • *S1_a_S2_3* pour la combinaison $\langle S11, S13_{1F} \rangle$
- 13 • *S1_a_S2_4* pour la combinaison $\langle S11, S13_{2F} \rangle$
- 14 • *S1_a_S2_5* pour la combinaison $\langle S11, S14 \rangle$
- 15 • *S1_a_S2_6* pour la combinaison $\langle S11, S1_{2F} \rangle$
- 16 • *S1_a_S2_7* pour la combinaison $\langle S12, S131 \rangle$
- 17 • *S1_a_S2_8* pour la combinaison $\langle S12, S132 \rangle$
- 18 • *S1_a_S2_9* pour la combinaison $\langle S12, S13_{1F} \rangle$
- 19 • *S1_a_S2_{10}* pour la combinaison $\langle S12, S13_{2F} \rangle$
- 20 • *S1_a_S2_{11}* pour la combinaison $\langle S12, S14 \rangle$
- 21 • *S1_a_S2_{12}* pour la combinaison $\langle S12, S1_{2F} \rangle$
- 22 • *S1_a_S2_{13}* pour la combinaison $\langle S1_{1F}, S131 \rangle$
- 23 • *S1_a_S2_{14}* pour la combinaison $\langle S1_{1F}, S132 \rangle$
- 24 • *S1_a_S2_{15}* pour la combinaison $\langle S1_{1F}, S13_{1F} \rangle$
- 25 • *S1_a_S2_{16}* pour la combinaison $\langle S1_{1F}, S13_{2F} \rangle$
- 26 • *S1_a_S2_{17}* pour la combinaison $\langle S1_{1F}, S14 \rangle$
- 27 • *S1_a_S2_{18}* pour la combinaison $\langle S1_{1F}, S1_{2F} \rangle$

28 Soit t une transition *SMD*, notons par *combinations*(t) la fonction qui calcule toutes les
 29 combinaisons possibles des états sources. Chaque combinaison est traduite par une transition
 30 *CPN*. Par exemple, considérons la transition de *S1* vers *S2* étiquetée par l'événement *b* dans
 31 la [Figure 5.6a](#). Les états *S11* et *S12* forment une combinaison ; par conséquent, une transition
 32 *CPN* est créée et sera notée *S1_b_S2_1* dans la [Figure 5.6b](#) (1 dénote que c'est la première
 33 combinaison). Dans le cas d'une transition *SMD* sans événement, nous la traduisons en une
 34 seule transition *CPN*, ce qui permet d'éviter l'explosion combinatoire dans ce cas : en effet,
 35 afin de sortir d'un état composite (avec une ou plusieurs régions) en utilisant une transition de
 36 terminaison, il est nécessaire que chaque région soit dans son état final, ce qui implique une seule
 37 transition *CPN*.

Places et transitions : nommage

Nous présentons dans le [Tableau 5.1](#) la convention de nommage pour les variables, places et transitions en *CPN* qui correspondent aux différents éléments *SMD* pris en compte dans la traduction. Notez que i dénote ou bien la i^e région d'un état composite orthogonal, ou bien la i^e transition entre $S1$ et $S2$; j dénote la j^e combinaison considérée pour cette transition (nous supposons que les régions, les transitions et les combinaisons sont ordonnées en utilisant un ordre lexicographique).

SMD		CPN	
Élément	Situation	Élément	Nommage
Variable v	-	variable	v
État final	Dans un état orthogonal s Dans un état composite s État final racine	place place place	s_iF sF F
État simple s	-	place	s
Pseudo-état histoire	Dans une seule région Dans plusieurs régions	place place	S_h S_ih
Comportement <i>do</i> d'un état s	-	transition	s_do
Transition sans événement	Une seule transition entre $S1$ & $S2$ Plusieurs transitions entre $S1$ & $S2$	transition transition	$S1_S2$ $S1_S2_i$
Transition avec événement e	Une seule transition entre $S1$ & $S2$ Plusieurs transitions entre $S1$ & $S2$	transition transition	$S1_e_S2_j$ $S1_e_S2_i_j$
Transition vers un pseudo-état histoire	Sans événement Avec événement e	transition transition	$S1_H_S2$ $S1_He_S2$

TABLE 5.1 – Conventions du nommage pour les éléments *CPN*

Algorithme de traduction

Nous décrivons dans [Algorithme 1](#) la procédure qui permet de traduire les diagrammes états-transitions vers les réseaux de Petri colorés. Tout au long de l'algorithme, les éléments à trait plein (places et transitions du résultat) dénotent les éléments ajoutés par les instructions en cours, alors que les éléments à trait pointillé dénotent les éléments ajoutés par les instructions précédentes. Nous divisons notre algorithme en trois étapes.

Première étape La première étape ([ligne 1](#) – [ligne 4](#)) traite la traduction des états ainsi que leurs comportements *do*, s'ils existent. Pour chaque état simple dans le diagramme états-transitions, nous ajoutons une nouvelle place avec le même nom que l'état (comme défini dans la [Section 5.4](#)). Ensuite, nous ajoutons une transition pour encoder le comportement *do* correspondant, s'il existe ([ligne 4](#)).

Seconde étape La seconde étape de l'algorithme ([ligne 5](#)–[ligne 18](#)) traite la traduction des transitions pour lesquelles aucun état cible n'est un pseudo-état histoire. Dans cette étape, il y a deux parties : la première partie ([ligne 6](#)–[ligne 12](#)) traite les transitions avec événement. Soit une transition t , nous utilisons la fonction *combinations*(t) afin d'engendrer les combinaisons possibles, et nous ajoutons les transitions correspondantes (voir [Section 5.3](#)). Rappelons que la fonction *combinations*(t) retourne comme résultat une liste d'ensembles d'états; chaque ensemble d'états représente une configuration de sortie de l'état composite, autrement dit un état actif dans chaque région de l'état composite avec ordre hiérarchique. La première boucle **foreach** ([ligne 9](#)–[ligne 10](#)) permet de lier la place de chaque état simple d'une combinaison avec la transition *CPN*

1 qui encode la transition *SMD* correspondante; la seconde boucle **foreach** (ligne 11–ligne 12)
 2 permet de lier la transition *CPN* encodant la transition *SMD* correspondante aux places qui
 3 encodent les états simples liés à des pseudo-états initiaux dans l’ensemble des états cibles de la
 4 transition *SMD*. Dans la seconde partie de la seconde étape (ligne 13–ligne 18), nous traitons les
 5 transitions UML sans événement. Dans un premier temps, nous ajoutons une transition *CPN* qui
 6 correspondra à la transition UML (ligne 13), et la transition portera le nom selon la convention
 7 de la Section 5.4. Nous lions les places correspondent aux états finaux de la source de la transition
 8 UML à la transition *CPN* (ligne 15–ligne 16). Enfin, tout comme les transitions avec événement
 9 (voir au dessus), nous lions la transition *CPN* aux places qui encodent les états simples liés avec
 10 les pseudo-états initiaux de l’état cible de la transition *SMD* (ligne 17–ligne 18).

11 **Troisième étape** La dernière étape (ligne 19–ligne 26) traite les transitions ayant comme
 12 cible un pseudo-état histoire. Rappelons que nous excluons la possibilité d’avoir une transition
 13 fork qui cible un pseudo-état histoire, par conséquent l’état cible est nécessairement unique. En
 14 revanche, la source de la transition peut être multiple, et un ou plusieurs état(s) source(s) peut
 15 être un état composite, ce qui nécessite d’énumérer les différentes combinaisons des états sources
 16 (ligne 20). Ensuite, pour chaque pseudo-état histoire cible (“*s₂*”), c’est-à-dire que pour chaque état
 17 non final dans la région du pseudo-état histoire, nous ajoutons une transition dédiée (ligne 22).
 18 Cette transition *CPN* aura une garde (avec la fonction *DecorateH()*), cela implique qu’elle peut
 19 être franchie uniquement si *s₂* est le dernier état visité dans cette région. Cette transition a
 20 comme source les places encodant tous les états simples de la combinaison considérée (ligne 24),
 21 et comme cible les places encodant tous les états initiaux de la cible du pseudo-état histoire
 22 (ligne 26); en effet, il est possible d’avoir plusieurs places dans le cas où la cible du pseudo-état
 23 histoire est un état composite.

24 **Place initiale** Nous ajoutons une place spéciale qui va contenir un jeton dans le marquage
 25 initial du *CPN* (ligne 27) et qui sera liée avec la place de l’état pointé par le pseudo-état initial
 26 du *SMD* (ou les places dans le cas où le *SMD* possède plusieurs états pointés par des pseudo-états
 27 initiaux). Notons que toutes les places associées aux variables (utilisées dans les actions ou dans
 28 les pseudo-états histoire) vont également contenir des jetons dans le marquage initial. Cette place
 29 est connectée via une transition à tous les états simples initiaux du *SMD* (ligne 28–ligne 29). Elle
 30 sera également décorée (en utilisant la fonction *AddBehaviours()*) avec tous les comportements
 31 d’entrée nécessaires qui doivent être effectués lorsque le *SMD* sera initialisé. Par exemple, dans
 32 le *SMD* de la Figure 5.4a, l’unique état simple initial est *S1*. Dans le *CPN* de la Figure 5.4b,
 33 la place *init* est connectée à *S1* via une transition où les comportements d’entrée (de *S* et *S1*)
 34 appropriés sont effectués.

35 **Gestion des gardes et comportements** Nous utilisons les fonctions *AddGuards()*, *DecorateH()*
 36 et *AddBehaviours()* (ligne 30) afin de gérer les comportements et les gardes dans la partie code
 37 des transitions. La fonction *AddGuards()* ajoute à n’importe quelle transition *CPN* la garde de
 38 la transition *SMD* correspondante. Par exemple, soit la garde *i = 0* de la transition venant de
 39 *S0* à *S1* dans la Figure 5.5a, *AddGuards()* permet d’ajouter une garde la transition *CPN* corres-
 40 pondante (Figure 5.5b), c’est-à-dire que [*!i = 0*] (où *!i* est une notation ML qui dénote l’accès à
 41 la valeur de *i*).

42 La fonction *DecorateH()* ajoute les gardes et les mises à jour des variables nécessaires aux
 43 transitions (représentant les pseudo-états histoire) comme suit. En premier, chaque transition
 44 *S1_He_s2_i_j* ajoutée par Algorithme 1 (ligne 22) est décorée avec une garde “[*Sh = s2*]”, où
 45 *S* est la région qui contient le pseudo-état histoire (rappelons que *Sh* est la variable *CPN* qui
 46 encode le pseudo-état histoire). En effet, le pseudo-état histoire a comme cible *s2* si et seulement



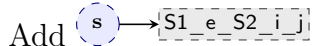

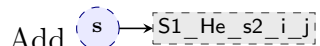

1 si s_2 est le dernier état visité dans la région qui contient le pseudo-état histoire. En second, pour
 2 chaque région S qui contient un pseudo-état histoire, pour chaque sous-état direct s_1 de S , pour
 3 chaque sous-état simple indirect de s_1 , nous ajoutons le code suivant à la transition CPN qui
 4 conduit à la place encodant cet état simple (le sous-état indirect de s_1) : $Sh := s_1$. Cela permet
 5 d'enregistrer le dernier état visité dans la région S qui est s_1 . Une exception est quand s_1 est un
 6 état final, dans cette situation, le code de la transition n'est pas $Sh := s_1$, mais plutôt $Sh := si$,
 7 où si est l'état destination du pseudo-état initial de la région. Cela est en accord avec le fait que,
 8 si nous quittons un état composite à travers son état final, et nous rentrons à travers son pseudo-
 9 état histoire, alors l'entrée sera par son entrée par défaut, c'est à dire le pseudo-état initial. Nous
 10 donnerons des exemples de la gestion des pseudo-états histoire dans la [Section 5.7.6](#).

11 La fonction *AddBehaviours()* est plus complexe que les autres fonctions et sera détaillée dans
 12 la [Section 5.6](#).

13 **Définition de l'état initial du réseau de Petri coloré** Rappelons que l'état initial
 14 résulte du marquage initial, et des valeurs initiales de toutes les variables globales. En ce qui
 15 concerne le marquage initial, nous ajoutons un jeton dans la place initiale *init*. Concernant les
 16 variables globales CPN qui encodent les variables SMD , leur valeur initiale est celle des variables
 17 SMD . Dans le cas où cette valeur n'est pas définie alors le modèle est considéré comme mal défini
 18 ("ill defined") ou bien une valeur initiale par défaut est affectée (par exemple, 0 pour les entiers,
 19 faux pour les Booléens, etc.). Concernant la variable globale CPN encodant la valeur du pseudo-
 20 état histoire, nous rappelons que nous prenons pas en compte la transition par défaut d'un
 21 pseudo-état histoire. Cependant, nous l'initialisons avec l'état pointé par le pseudo-état initial de
 22 la région contenant le pseudo-état histoire. Le pseudo-état histoire aura la valeur de l'état pointé
 23 par le pseudo-état initial dans le cas où la région (contenant le pseudo-état histoire) est visitée
 24 pour la première fois. Cela est conforme avec la spécification ("If no default history Transition is
 25 defined, then standard default entry of the Region is performed" [?, p.307]).

Algorithm 1: Translating an SMD $(\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, \text{SubStates}, \mathcal{T})$ into a CPN

```

// Step 1: Simple states and their do behaviours
1 foreach simple state  $s \in \mathcal{S}$  do
2   Add 
3   if  $s$  has a do behaviour then
4     Add 
// Step 2: Transitions
5 foreach transition  $t = (\mathbf{S}_1, e, g, (b, f), sLevel, \mathbf{S}_2) \in \mathcal{T}$  with no history pseudostate
   within  $\mathbf{S}_2$  do
6   if  $e \neq noEvent$  then
7     foreach  $c \in combinations(t)$  do
8       Add  $\boxed{S1\_e\_S2\_i\_j}$  //  $j$ th combination of the  $i$ th transition from
         $\mathbf{S}_1$  to  $\mathbf{S}_2$  with  $e$ 
9       foreach simple state  $s \in c$  do
10        Add 
11        foreach simple state  $s \in init^*(\mathbf{S}_2)$  do
12        Add  $\boxed{S1\_e\_S2\_i\_j} \rightarrow s$ 
13      else
14        Add  $\boxed{S1\_S2\_i}$  //  $i$ th transition without event from  $\mathbf{S}_1$  to  $\mathbf{S}_2$ 
15        foreach simple state  $s \in final^*(\mathbf{S}_1)$  do
16        Add 
17        foreach simple state  $s \in init^*(\mathbf{S}_2)$  do
18        Add  $\boxed{S1\_S2\_i} \rightarrow s$ 
// Step 3: History states
19 foreach transition  $t = (\mathbf{S}_1, e, g, (b, f), sLevel, H) \in \mathcal{T}$  do
20   foreach  $c \in combinations(t)$  do
21     foreach state  $s_2$  such that  $s_2$  is a non-final direct substate of  $r(H)$  do
22       Add  $\boxed{S1\_He\_s2\_i\_j}$  //  $j$ th combination of the  $i$ th transition from
         $\mathbf{S}_1$  to  $H$  with  $e$ 
23       foreach state  $s \in c$  do
24       Add 
25       foreach simple state  $s \in init^*(s_2)$  do
26       Add  $\boxed{S1\_He\_s2\_i\_j} \rightarrow s$ 
27 Add  //  $S$  is pointed by the initial pseudostate of  $SMD$ 
28 foreach  $s \in init^*(SMD)$  do
29   Add  $\boxed{init\_S} \rightarrow s$ 
30  $AddGuards()$  ;  $DecorateH()$  ;  $AddBehaviours()$ ;

```

1 L'ajout des comportements au code des transitions

2 Nous présentons dans cette section, en détails, la fonction *AddBehaviours()*. Rappelons qu'un
3 comportement UML est abstrait en utilisant le couple (b, f) , où b est l'expression du compor-
4 tement et f est la liste des fonctions exprimant les changements appliqués sur les valeurs des
5 variables du système. Rappelons aussi que nous ne traduisons que les fonctions f .

6 **Comportement Do** Chaque comportement est encodé avec une transition *CPN* liée avec
7 la place qui encode l'état correspondant (ligne 4 de l'Algorithme 1). Par conséquent, la fonction
8 *AddBehaviours()* ajoutera à la partie code de cette transition l'équivalent de la fonction f . Par
9 exemple, le comportement do de l'état **S2** dans Figure 5.4a (c'est-à-dire que $j = j + 1$) est traduit
10 par le code associé à la transition **S2_do** dans la Figure 5.4b (c'est-à-dire que $j := j + 1$).

11 **Comportement d'entrée et de sortie** Une difficulté majeure dans la formalisation des
12 *SMD* est de traiter correctement les comportements d'entrée et de sortie lors du franchissement
13 des transitions *SMD*. La fonction *AddBehaviours()* est responsable de la décoration de la partie
14 code des transitions *CPN* afin de prendre en compte tous les comportements d'entrée et de sortie
15 pendant le franchissement d'une transition *SMD*. Rappelons que, pendant le franchissement
16 d'une transition *SMD*, les comportements de sortie doivent être exécutés en premier, venant des
17 états les plus intérieurs de l'état source (ou des états source si la transition est un "fork") jusqu'à
18 l'état qui contient la transition (nommé ici par "état niveau" ou "level state" *sLevel* dans la Définition 2),
19 le comportement de la transition sera ensuite exécuté. Enfin, les comportements d'entrée
20 sont exécutés venant de l'état niveau jusqu'à l'état le plus interne ("innermost") de l'état(s) desti-
21 nation de la transition. Le mécanisme est relativement simple en l'absence de concurrence, c'est-à-
22 dire que dans le cas où un état composite n'est pas orthogonal. Soit la transition de l'état **S0** vers
23 l'état **S1** dans le *SMD* de la Figure 5.1a, pendant le franchissement de cette transition, le compor-
24 tement de sortie de **S0** sera exécuté en premier (c'est-à-dire que $i = 0$ doit être exécuté). Le
25 comportement associé à la transition est ensuite exécuté ($j = 3$), enfin le comportement d'entrée
26 de l'état **S1** est exécuté (c'est-à-dire que $j = 1$), suivi par le comportement de **S11** et **S12** (c'est-à-
27 dire que $i = i + 1$ et $i = 3$). En correspondance à l'exécution des comportements en franchissant
28 la transition entre **S0** et **S1** le code suivant est ajouté à la transition **S0_S1** dans la Figure 5.1b
(qui encode la transition *SMD*) : $i := 0; i := 3; j := 1; interleave(< i := i + 1 >, < i := 3 >)$.

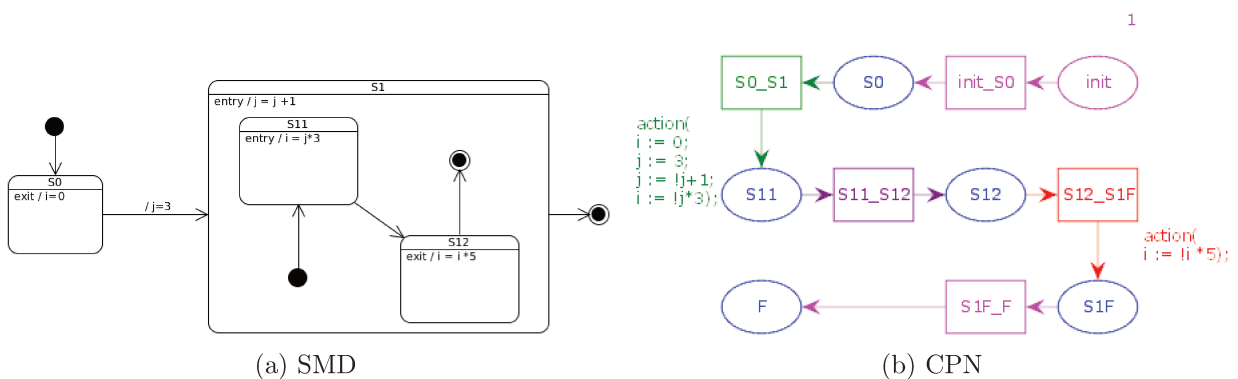


FIGURE 5.1 – Exemple d'un diagramme états-transitions sans concurrence et avec des comportements

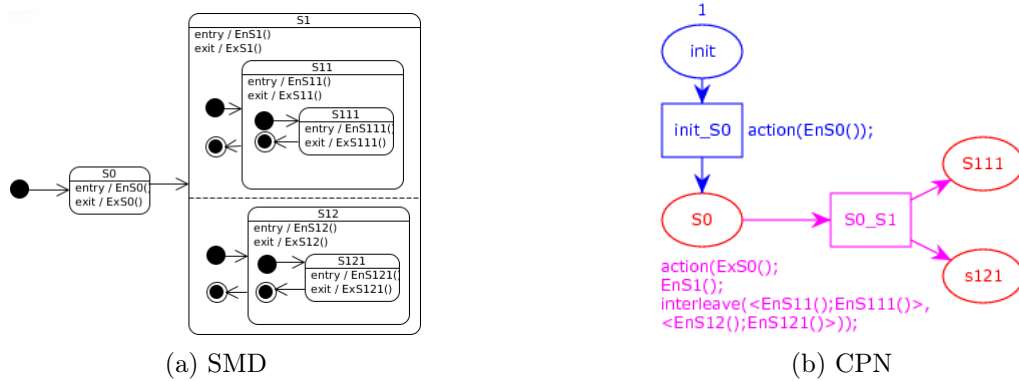


FIGURE 5.2 – Exemple d’un diagramme états-transitions avec de la concurrence et des comportements

1 Cependant, l’exécution des comportement est plus compliquée dans les états composites or-
 2 thogonaux. Soit la transition de **S0** vers **S1** dans le diagramme états-transitions de la [Figure 5.2a](#).
 3 À la sortie de l’état **S0** et une fois que le comportement de la transition est exécuté, le comporte-
 4 ment d’entrée de l’état **S1** sera exécuté. Ensuite, les comportements des états **S11** et **S12** seront
 5 exécutés d’une manière concurrente, ce qui est représenté par :

- 6 1. `exS0(); enS1(); enS11(); enS111(); enS12(); enS121()`
- 7 2. `exS0(); enS1(); enS11(); enS12(); enS111(); enS121()`
- 8 3. `exS0(); enS1(); enS11(); enS12(); enS121(); enS111()`
- 9 4. `exS0(); enS1(); enS12(); enS11(); enS111(); enS121()`
- 10 5. `exS0(); enS1(); enS12(); enS11(); enS121(); enS111()`
- 11 6. `exS0(); enS1(); enS12(); enS121(); enS11(); enS111()`

12 En effet, la seule exigence est que le comportement d’entrée d’un état doit apparaître après
 13 le comportement d’entrée de son état contenant (ou de ses états contenant), c’est-à-dire que
 14 les comportements d’entrée de l’état niveau de chaque pseudo-état initial le plus interne doivent
 15 être exécutés de façon séquentielle (et inversement pour les comportements de sortie) ; cependant,
 16 toutes les séquences possibles peuvent être exécutées de manière entrelacée.

17 Une solution simple est d’énumérer toutes ces possibilités et de créer une transition *CPN* pour
 18 chaque séquence de comportements. Au lieu de cela, nous proposons une fonction *interleave()*
 19 qui est utilisée afin de modéliser tous les entrelacements, et cela en gardant notre représenta-
 20 tion compacte. Soit une liste de comportements (d’entrée ou de sortie) `exS1, ..., exSn`, nous
 21 notons `< exS1, ..., exSn >` la séquence des comportements exécutée dans cet ordre. Soit la liste
 22 de plusieurs séquences `seq1, ..., seqn`, notre fonction *interleave(seq1, ..., seqn)* considère tous les
 23 entrelacements entre ces séquences de comportements. Autrement dit, dans une séquence donnée
 24 `seqi`, les comportements sont exécutés d’une façon séquentielle ; mais l’ordre entre les comporte-
 25 ments de `seqi` et celui des autres séquences peut être arbitraire. En fait, cette fonction *interleave()*
 26 est du “sucre syntaxique” pour une liste de transitions d’un *CPN*, chaque transition contenant un
 27 entrelacement particulier. L’utilisation de cette fonction permet une vision plus claire et propre
 28 de nos figures¹.

1. Bien que nous utilisions cette fonction dans les captures d’écrans de CPNTools, cette fonction

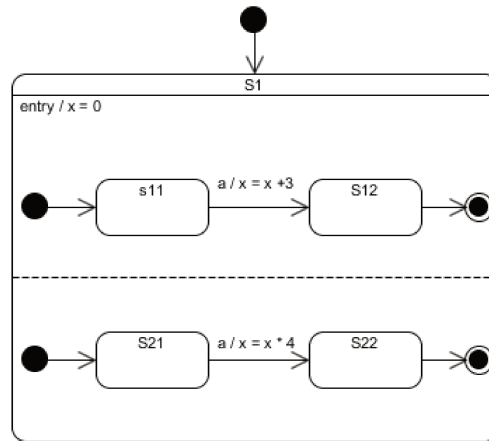


FIGURE 5.3 – Exemple avec modification concurrente des variables

1 Par exemple, l’entrelacement des comportements au-dessus peuvent être représentés comme
 2 suit : `exS0(); enS1(); interleave(< enS11(); enS111() >, < enS12(); enS121() >)`

3 Enfin, rappelons qu’il y a un certain non-déterminisme dans UML dans le cas d’exécution des
 4 actions en concurrence des transitions ce qui provoque un accès concurrent aux variables globales.
 5 Par exemple, le diagramme états-transitions dans la Figure 5.3 comporte deux transitions qui
 6 modifient la variable x . Ces deux transitions (la transition de $S11$ vers $S12$ et la transition de
 7 $S21$ vers $S22$) sont synchronisées par l’occurrence de l’événement a . Selon l’ordre d’exécution
 8 des transitions, nous obtenons des valeurs différentes, c’est-à-dire qu’en exécutant la transition
 9 de $S11$ vers $S12$ ensuite la transition de $S21$ vers $S22$ la variable x contiendra la valeur “12”.
 10 En revanche, en exécutant la transition de $S21$ vers $S22$ ensuite la transition de $S11$ vers $S12$ la
 11 variable x contiendra la valeur “3”.

12 L’exécution des *CPN* est également non-déterministe, c’est-à-dire que le choix de la transi-
 13 tion qui sera franchie est non-déterministe. Cependant, dans la phase de vérification, le modèle
 14 checker doit explorer toutes les combinaisons possibles, c’est-à-dire qu’il va considérer toutes les
 15 exécutions possibles du diagramme états-transitions. Ceci dit, le résultat de la phase du model
 16 checking est bien naturellement déterministe. Bien que l’exécution des actions en concurrence
 17 d’une transition cause un accès concurrent aux variables globales, ces actions sont exécutées d’une
 18 manière instantanée. À l’ajout du temps dans notre transformation (Chapitre 8), ces actions se-
 19 ront certainement exécutées en temps zéro. Le résultat est en effet équivalent : après l’exécution
 20 des comportements dans un certain ordre dans le *CPN*, les différentes configurations possibles
 21 résultantes (en fonction du non-déterminisme) seront équivalentes à celles du *SMD* après la fin
 22 de l’étape de run-to-completion.

23 Application de l’algorithme à des cas simples

24 Nous illustrons notre traduction (formalisée par l’Algorithme 1) par diverses situations.

n’est en réalité pas supportée par CPN Tools. Par conséquent, dans notre modèle du lecteur de CD, nous devons dupliquer nos transitions afin de modéliser tous les entrelacements possibles, ce qui rend notre modèle difficile à lire. Cela a été une motivation afin de proposer une telle représentation syntaxique. L’intégration d’une telle fonction dans CPN Tools pourrait être envisagée, soit en collaborant avec les développeurs de CPN Tools, soit en utilisant un script en prétraitement.

1 États simples et comportements “Do”

2 *Situation considérée* La Figure 5.4a montre un exemple d’un état composite **S** avec deux sous-états **S1** et **S2**.

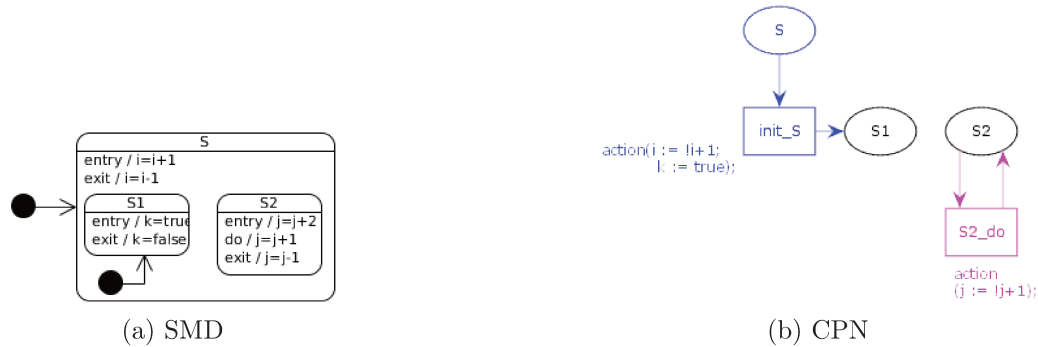


FIGURE 5.4 – Exemple d’états simples et de comportement “Do”

3

4 *Traduction* Les deux états simples dans Figure 5.4a sont traduits en deux places corres-
 5 pondantes (**S1** et **S2**) dans la Figure 5.4b. La troisième place (**S**) est une place d’initialisation
 6 avec sa transition (**init_S**) correspondante. Nous ajoutons également une transition (**S2_do** dans
 7 Figure 5.4b) qui correspond au comportement **do** de l’état **S2**.

8 Transitions sans événements avec régions orthogonales

9 *Situation considérée* Nous considérons dans la Figure 5.5a les transitions sans événements
 10 entre états simples/composites et orthogonaux. Dans le cas de la sortie d’un état orthogonal,
 11 il est nécessaire de sortir de chaque région (à travers son état final). Dans le cas de l’entrée
 12 dans un état orthogonal, il est nécessaire d’entrer dans chaque état (simple) initial de chaque
 13 région. Notez que l’entrée dans les sous-états de l’état **S1** est faite d’une manière concurrente,
 14 c’est-à-dire qu’à l’entrée dans l’état **S1** nous exécutons son comportement d’entrée, ensuite nous
 15 exécutons le comportement d’entrée du sous-état **S11** ensuite le comportement d’entrée de **S12**,
 16 ou le comportement d’entrée de **S12** ensuite le comportement de **S11**.

17 *Traduction* La transition de l’état **S0** à l’état **S1** est traduite en une transition CPN (**S0_S1**
 18 dans la Figure 5.5b). À l’entrée des états **S11** et **S12**, les comportements d’entrée de **S1**, **S11**
 19 et **S12** doivent être exécutés. La fonction *interleave()* est utilisée pour modéliser le fait que
 20 les comportements d’entrée des états **S11** et **S12** sont exécutés d’une manière concurrente (en
 21 parallèle). La transition entre l’état **S1** et **S2** est traduite en une transition CPN : **S1_S2** dans
 22 la Figure 5.5b. Nous sortons de l’état **S1** à travers ses états finaux (c’est-à-dire que **S1_1F** et
 23 **S1_2F**) des différentes régions, car la transition est une transition de terminaison.

24 Transitions avec événement et régions orthogonales

25 *Situation considérée* Nous considérons dans la Figure 5.6a les transitions étiquetées avec
 26 des événements entre des états simples et composites. Rappelons que sortir d’un état orthogonal à
 27 travers une transition étiquetée avec un événement résulte en un certain nombre de combinaisons.

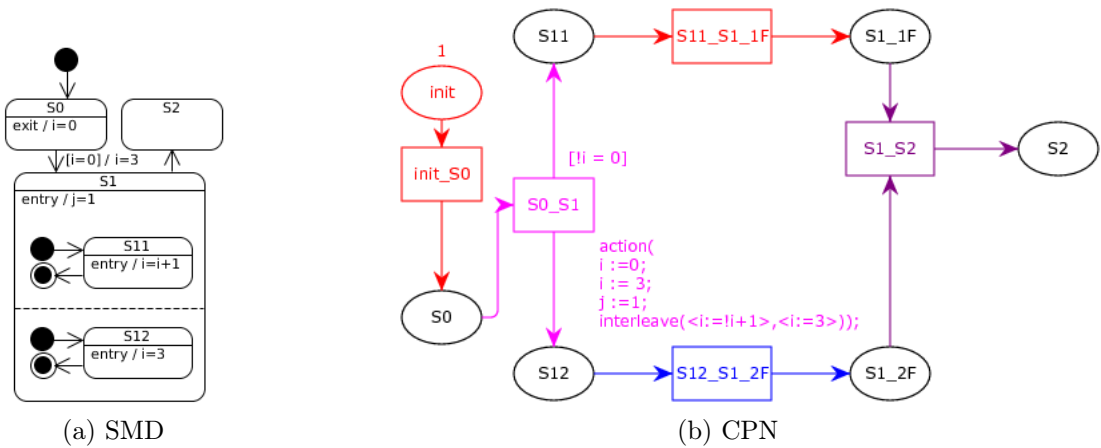


FIGURE 5.5 – Exemple : transitions sans événements avec des régions orthogonales

- 1 Sortir de l'état **S1** à travers la transition étiquetée avec l'événement **b**, résulte aux combinaisons
- 2 suivantes (le système peut être dans l'une de ces combinaisons) :
- 3 $\{(S11, S12), (S11, S13), (S11, S1_2F), (S1_1F, S12), (S1_1F, S13), (S1_1F, S1_2F)\}$.

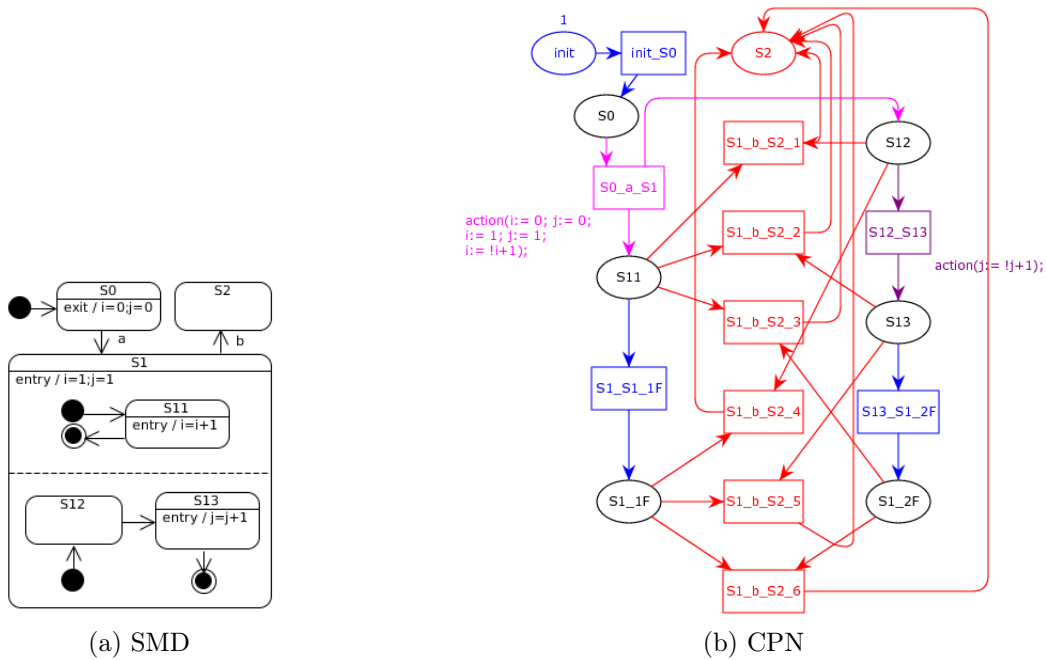


FIGURE 5.6 – Exemple : transitions avec événements et régions orthogonales

- 4 **Traduction** La traduction de l'exemple dans la Figure 5.6a est présentée dans la Figure 5.6b.
- 5 La transition étiquetée par l'événement **b** entre l'état **S1** et **S2** est modélisée par six transitions
- 6 CPN (une transition CPN pour chaque combinaison d'états simples dans chaque région ortho-
- 7 gonale). Par exemple, pour la combinaison entre **S11** et **S13**, nous ajoutons une transition CPN :
- 8 **S1_b_S2_2**.

1 Hiérarchie des régions orthogonales

2 **Situation considérée** Nous considérons la hiérarchie des régions orthogonales dans l'exemple
 3 de la Figure 5.7a (avec une profondeur de trois). L'état orthogonal **S1** contient dans ses régions
 4 autres états orthogonaux (**S11** et **S12**), qui eux mêmes contiennent deux régions chacun. Pour
 5 entrer dans l'état **S1** à partir de l'état **S0**, nous avons besoin d'exécuter en premier le compor-
 6 tement d'entrée de **S1**, ensuite nous exécutons en parallèle les comportements d'entrée des états
 7 **S11** et **S12** pour respecter la hiérarchie des comportements.

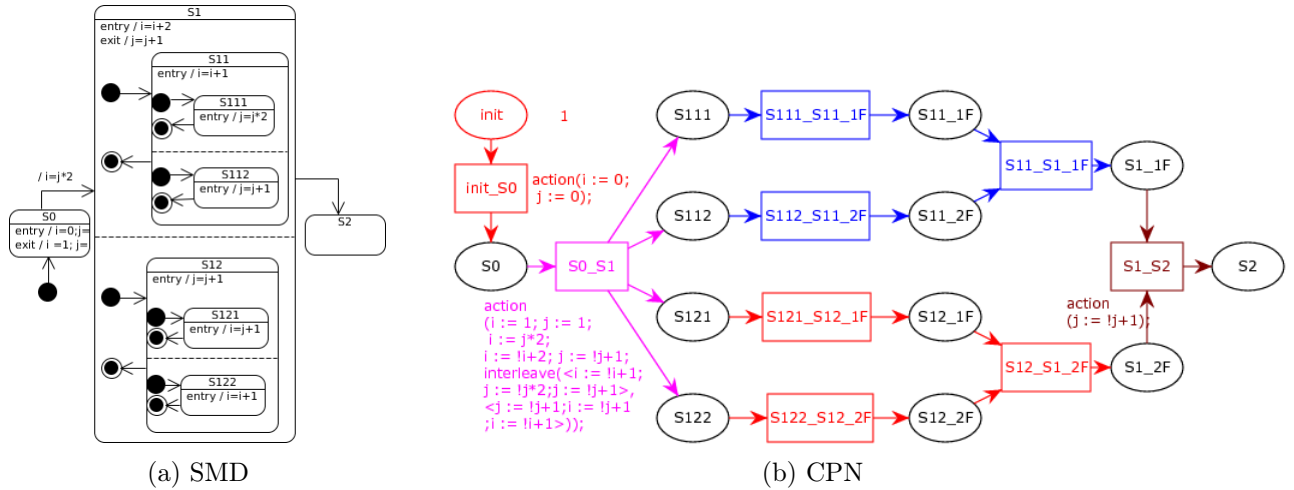


FIGURE 5.7 – Exemple : hiérarchie des régions orthogonales

8 **Traduction** La traduction de l'exemple de la Figure 5.7a est donnée dans la Figure 5.7b.
 9 Rappelons que, afin d'avoir un CPN compact, nous utilisons la fonction *interleave()* qui exécute
 10 les comportements d'entrée ou de sortie d'une manière concurrente (en parallèle). Cela permet-
 11 trait d'avoir un CPN compact quelque soit la profondeur de la hiérarchie. Une ancienne version
 12 de notre travail [?] donne un CPN plus large et cela à cause des énumérations exhaustives de
 13 toutes les combinaisons des comportements d'entrée (ou de sortie).

14 Pseudo-état fork

15 **Situation considérée** Nous considérons le pseudo-état fork dans la Figure 5.8a, qui a un
 16 seul état source **S0**, et trois états cibles **S11**, **S12** et **S13**.

17 **Traduction** Nous modélisons le pseudo-état fork avec une seule transition **S0_S11S12S13**
 18 qui a comme source une place représentant l'état **S0**, et comme cible les places modélisant les
 19 états **S11**, **S12** et **S13**, comme le montre la Figure 5.8b. Le segment de code de la transition
 20 **S0_S11S12S13** contient le comportement de sortie de l'état **S0** et les comportements d'entrée
 21 (exécutés en parallèle) de **S11**, **S12** et **S13**. Nous observons que dû à l'ordre relatif des compor-
 22 tements $i = i + 1$ et $i = i * 2$, la valeur de i peut être 1 ou 2 après l'exécution de la transition.

23 Pseudo-état histoire

24 **Situation considérée** Nous considérons un pseudo-état histoire dans la Figure 5.9a. Rap-
 25 pelons que seulement les états composites peuvent avoir un pseudo-état histoire. La transition du

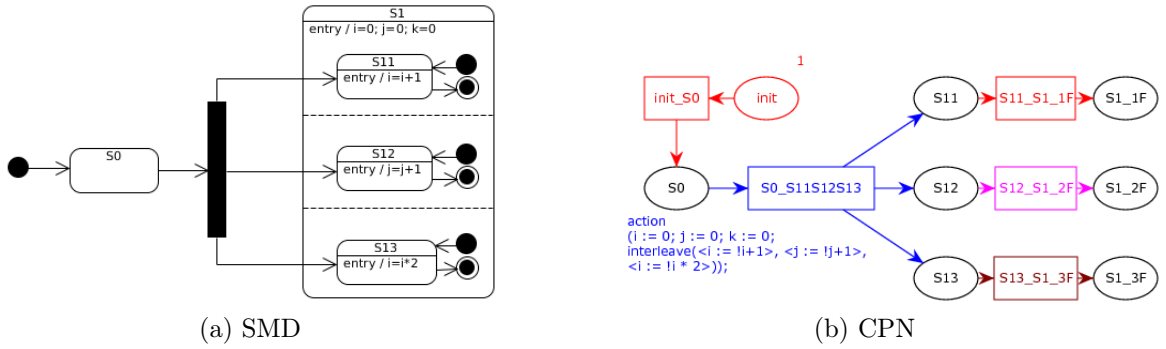


FIGURE 5.8 – Exemple : pseudo-état fork

- 1 pseudo-état histoire (c'est-à-dire que la transition étiquetée avec l'événement a dans Figure 5.9a)
- 2 peut être exécutée à partir de n'importe quel état (c'est-à-dire que S11, S12 et l'état final de
- 3 l'état S1) contenu dans la région qui contient le pseudo-état

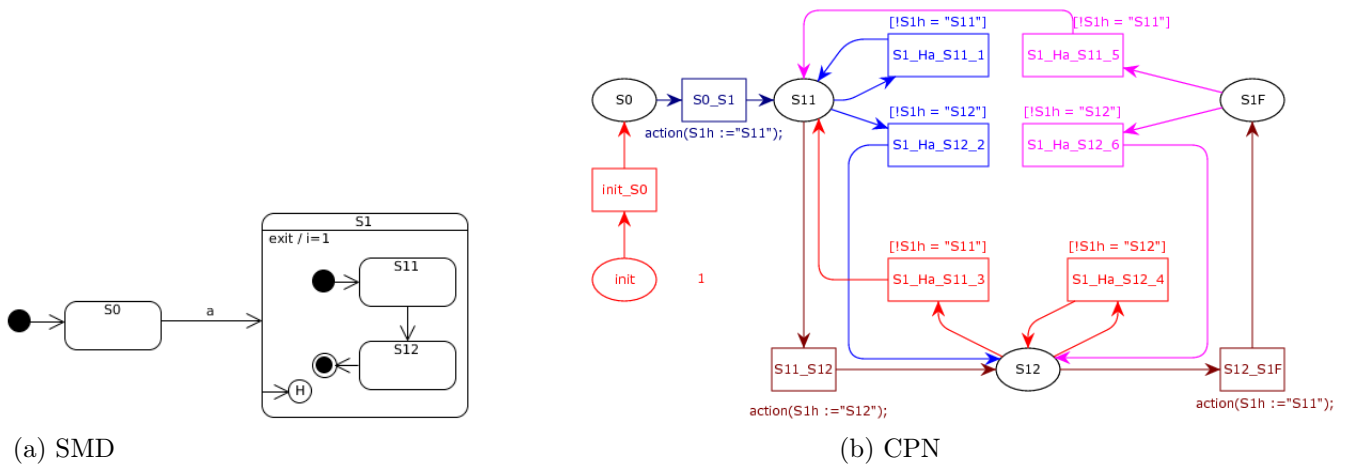


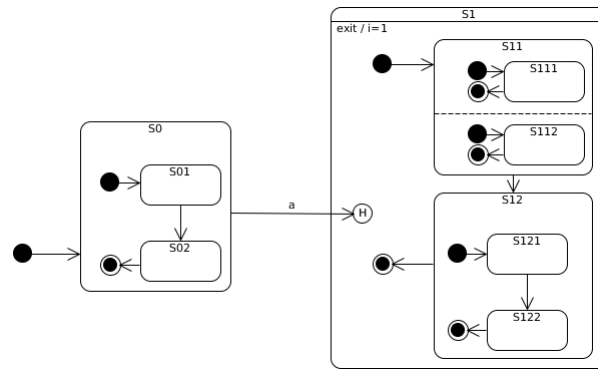
FIGURE 5.9 – Exemple : pseudo-état histoire

- 4 **Traduction** Nous associons pour chaque pseudo-état histoire une variable qui sauvegarde
- 5 l'état courant à chaque fois qu'une transition est exécutée dans la région qui contient le pseudo-
- 6 état histoire.

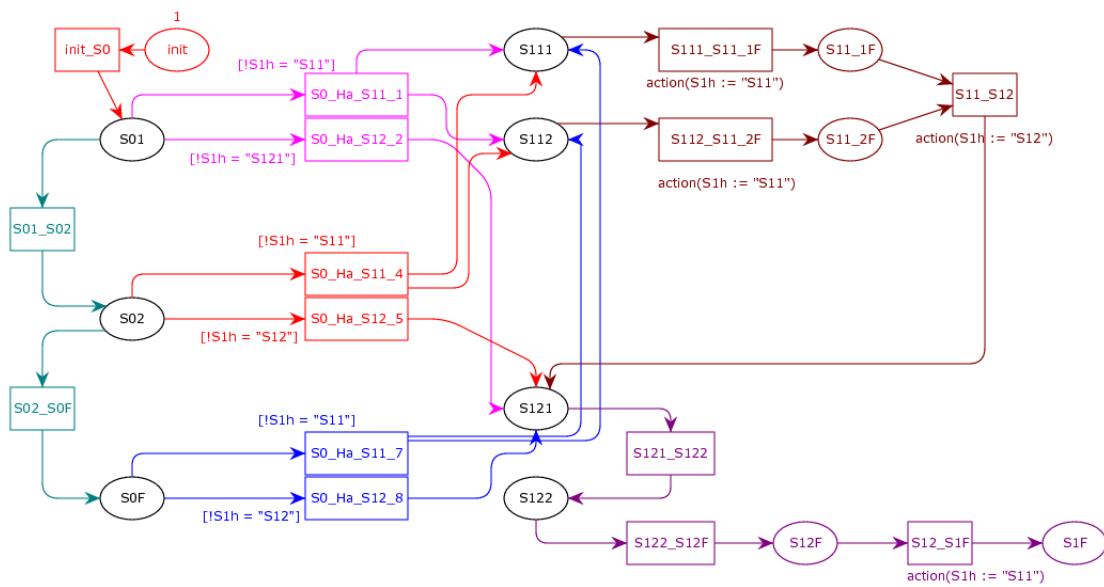
- 7 Dans la Figure 5.9b, la variable S1h est mise à jour dans chaque transition qui conduit à la
- 8 place CPN modélisant un sous-état de l'état S1 (par exemple, le segment de code S1h := "S11"
- 9 de la transition S0_S1).

- 10 Ensuite, pour chaque sous-état direct de l'état S1, deux transitions CPN correspondantes à la
- 11 transition UML étiquetée par l'événement a sont créées, une dirigée vers l'état S11 (si l'état S11
- 12 est le dernier état visité), et l'autre dirigée vers l'état S12 (si l'état S12 est le dernier état visité).
- 13 Donc à chaque fois que le pseudo-état histoire est atteint par une transition, alors le CPN atteint
- 14 la place modélisation le dernier état visité, et cela grâce à la variable du pseudo-état histoire qui
- 15 est testée dans la garde de la transition (par exemple, la transition S1_Ha_S11_1 avec la garde
- 16 "[S1h = "S11"]"). Enfin, nous présentons un autre exemple d'un pseudo-état histoire, transition
- 17 inter-niveau, et des régions orthogonales. L SMD est donné dans la Figure 5.10a et le CPN dans
- 18 la Figure 5.10b. Nous observons que la variable S1h est mise à jour à S11 dans dans la place

- 1 encodant l'état S11, mais aussi à l'entrée des places encodant ses sous-états (S111 and S112).



(a) SMD



(b) CPN

FIGURE 5.10 – Exemple : pseudo-état histoire avec la concurrence

1 Au-delà de notre traduction

2 Supprimer les hypothèses

3 La plupart des hypothèses que nous avons effectuées ont pour but de simplifier notre algo-
4 rithme. Nous discutons, dans ce qui suit, comment les supprimer.

5 ***Pseudo-état initial et comportements*** Rappelons qu’une transition originaire d’un
6 pseudo-état initial peut avoir un comportement associé, et afin de garder notre algorithme simple
7 nous excluons cet aspect. Cependant, nous pouvons prendre en compte cela simplement en ajou-
8 tant à la liste des comportement à exécuter le comportement du pseudo-état histoire. Ce com-
9 portement sera exécuté après le comportement d’entrée de l’état parent (dans le cas où l’état
10 destination n’est pas un état racine) et avant le comportement d’entrée de l’état considéré.

11 ***Modélisation des expressions des comportements*** Rappelons que, bien que nous
12 modélisons la modification des variables dans les comportements d’entrée et de sortie, nous ne
13 prenons pas en compte dans notre traduction les expressions des comportements (par exemple,
14 le comportement d’entrée de l’expression “fade in” dans l’état PLAYING dans la Figure 6.1) Cela
15 nous empêche de vérifier par exemple des propriétés telles que “quand une piste est jouée (le
16 comportement do play track), alors “fade in“ (le comportement d’entrée fade in) est nécessai-
17 rement apparu”. Afin de considérer de telles propriétés, une variable globale booléenne fade in
18 peut être ajoutée, qui sera mise à vrai lors de l’entrée dans la place correspondante à l’état
19 PLAYING, et à faux lors de la sortie de cette la place. Par conséquent, vérifier cette propriété
20 consiste à vérifier si la variable booléenne fade in est égale à vrai avant que la variable encodant
21 le comportement do play track ne soit modifiée.

22 ***Comportement Do*** Une hypothèse que nous avons effectuée est que seuls les états simples
23 ont des comportements do. Afin d’étendre cela à des états composites, il suffit d’ajouter une
24 transition avec le même état comme source et destination (“self-loop”) introduite dans l’étape-1
25 (step-1) dans l’Algorithme 1 pour tout sous-état simple (direct ou indirect) de l’état composite.
26 En outre, nous supposons que le comportement do peut être exécuté autant de fois que nous
27 voulions (cette hypothèse est forte). En effet, le CPN est un modèle discret qui ne capture pas la
28 nature continue de certains comportement tels que play track. L’adaptation de notre traduction
29 afin de prendre en compte l’aspect continu des comportements “do” n’est pas évidente (d’autres
30 formalismes tels que CSP ne le peuvent pas non plus). Cependant, nous pouvons exiger que le
31 comportement do soit exécuté au plus une fois. Cela est possible en exigeant que la transition avec
32 le même état comme source et destination (“self-loop”) introduite dans l’étape 1 de l’Algorithme 1
33 ne puisse être exécutée qu’une seule fois, ce qui peut être effectué en utilisant une variable globale
34 additionnelle (ou un jeton système), qui permet à cette transition d’être exécutée au plus une
35 fois à chaque fois que la place associée est marquée. Il sera peut-être plus simple de capturer la
36 nature continue des comportements “do” une fois que notre travail sera étendu au temps continu.

37 ***Pseudo-états fork et join implicites*** Une transition modélisant un pseudo-état fork
38 ou join n’entre pas (ou ne sort pas dans le cas d’un join) dans toutes les régions d’un état
39 orthogonal (ou ne sort pas de toutes les régions d’un état orthogonal dans le cas d’un join).
40 Prendre en compte cette situation n’est pas difficile dans notre travail, seulement il faut ajouter
41 un cas particulier dans l’étape 2 de l’Algorithme 1. Pour un join implicite, toutes les régions
42 qui n’ont pas de transitions sortantes vers ce pseudo-état peuvent sortir de n’importe quel état.

1 Pour un fork implicite, toutes les régions qui n'ont pas de transitions entrantes originaires de ce
2 pseudo-état, elles seront entrées via leurs pseudo-états initiaux.

3 ***Pseudo-état histoire et fork*** Nous exigeons qu'un pseudo-état fork ne doive pas avoir
4 comme destination un pseudo-état histoire. Cette hypothèse peut être supprimée, en fusionnant
5 simplement les étapes 2 et 3 de l'Algorithme 1 en prenant en compte ce cas (par exemple, une
6 condition **if** qui permettra de vérifier si l'une des destinations du fork est un pseudo-état histoire,
7 ensuite la transition sera ajoutée dans l'étape 3).

8 ***Transition entre pseudo-états*** Nous ne considérons pas dans notre travail les transitions
9 entre pseudo-états (par exemple, une transition originale d'un pseudo-état initial vers un pseudo-
10 état fork) afin de garder notre algorithme plus simple. Cependant, nous pouvons prendre en
11 compte cette situation, simplement en remplaçant le pseudo-état cible par ses états cibles.

12 ***Transition avec plusieurs événements*** Dans la spécification, chaque transition peut
13 être étiquetée par un ou plusieurs événements, dans notre travail nous prenons en compte uni-
14 quement le cas d'un seul événement. Afin de supprimer cette restriction, il suffit de remplacer la
15 transition avec plusieurs événements par plusieurs transitions (ayant les mêmes états source/desti-
16 nation) où chaque transition est étiquetée par l'un des événements de la transition de départ.
17 Par conséquent, à chaque apparition d'un événement, l'une des transitions avec l'événement
18 correspondant sera exécutée.

19 Extension de la syntaxe

20 ***Transition par défaut : pseudo-état histoire*** La transition par défaut du pseudo-
21 état histoire peut être ajoutée facilement à notre algorithme (elle n'est pas prise en compte
22 afin de limiter le nombre de cas dans l'Algorithme 1). Deux opérations devraient être ajoutées :
23 premièrement, il est nécessaire d'initialiser la variable associée au pseudo-état histoire avec comme
24 valeur l'état cible de la transition par défaut du pseudo-état histoire. Deuxièmement, une fois
25 dans l'état final d'une région ayant un pseudo-état histoire, la variable (du pseudo-état histoire)
26 doit être mise à jour non pas avec l'état initial de la région mais avec l'état cible de la transition
27 par défaut du pseudo-état histoire.

28 ***Pseudo-état histoire profond*** Nous prenons en compte dans notre travail uniquement le
29 pseudo-état histoire plat ("shallow history pseudostate") mais pas le pseudo-état histoire profond
30 ("deep history pseudostate). Cet élément syntaxique peut être ajouté à notre traduction, et
31 cela en enregistrant à chaque fois tous les états de la hiérarchie jusqu'au dernier état visité (et
32 non pas que le dernier état visité). Donc, tous les états et leurs sous-états directs et indirects
33 seront enregistrés. Par conséquent, le type de la variable utilisée pour modéliser les pseudo-états
34 histoire n'est plus un état mais une liste d'états. En outre, cette variable doit être mise à jour
35 non seulement à l'entrée d'un sous-état direct, mais aussi à l'entrée des sous-états indirects de
36 l'état composite qui contient le pseudo-état histoire. Enfin, si une transition a comme destination
37 un pseudo-état histoire profond, alors elle doit avoir comme destination toute la hiérarchie des
38 derniers états visités, et non pas seulement les états du niveau supérieur.

39 ***Transitions internes*** Les transitions internes peuvent être exécutées dans le cas où leurs
40 événements sont dispatchés et leurs gardes qui doivent être satisfaites. L'exécution des transi-
41 tions internes n'engendre pas l'exécution des comportements d'entrée et de sortie. Par conséquent,

elles sont considérées comme des transitions locales dans notre traduction (ce qui est conforme à la spécification [?, Section 12.2.3.8.1, p.312]) avec la particularité que l'état source et l'état cible de ces transitions sont les mêmes et qu'il n'y a aucune entrée ou sortie d'un état lors de leurs exécutions.

Pseudo-états jonction et choix Les pseudo-états jonction peuvent être ajoutés à notre traduction facilement : la destination d'un pseudo-état jonction dépend de l'évaluation de la garde respective. Le même mécanisme est appliqué dans les CPNs, à l'exception que nous avons besoin de dupliquer la transition CPN (une transition CPN avec une garde pour chaque cas). La modélisation de la fusion est triviale (simplement en ajoutant une transition CPN pour chaque transition SMD menant à la fusion).

La modélisation du pseudo-état choix est un peu plus difficile : rappelons qu'un pseudo-état choix est similaire à un pseudo-état jonction, à l'exception que les gardes sont évaluées *dynamiquement*, c'est-à-dire que quand la transition atteint ce pseudo-état. Ceci dit, l'évaluation des gardes peut dépendre des comportements de sortie exécutés à la sortie de l'état actif vers ce pseudo-état. La modélisation du pseudo-état avec les CPNs en soit n'est pas un problème, cependant dans le cas où aucune garde n'est évaluée à vrai, le CPN doit tout de même produire un jeton, ce qui peut conduire à des problèmes dans le model checking. Notez cependant que la spécification de l'OMG considère cette situation comme un modèle mal formé. En outre, le pseudo-état histoire rend l'algorithme de sélection des transitions plus lourd. Cela dit, afin d'évaluer si une transition contenant un pseudo-état histoire est active, il suffit de vérifier s'il y a un chemin à partir de l'état source vers le pseudo-histoire de choix où toutes les gardes sont satisfaites, par conséquent si l'une des gardes est évaluée à vrai il n'y a aucun problème (sinon, le modèle est mal formé).

Points d'entrée, de sortie, pseudo-état de terminaison et états submachines Les points d'entrée et de sortie ne posent aucun problème particulier. Le pseudo-état de terminaison (qui implique que l'exécution du SMD s'arrête immédiatement) peut être modélisé simplement en ajoutant une variable globale booléenne qui aura comme valeur faux dans le cas où le pseudo-état est atteint (vrai dans les autres cas). Ensuite, chaque transition CPN est gardée en vérifiant la valeur de cette variable si elle est à vrai. Cela sera similaire à la traduction du nœud d'activité final des diagrammes d'activités UML vers les réseaux de Petri colorés dans [?]. Les états submachines sont "semantically equivalent to a composite state" [?, p.309], et par conséquent sont en fait actuellement considérés dans notre traduction.

Temps et événements différés Ces deux points ne sont pas considérés dans notre travail, et il n'est pas clair si notre schéma général peut être facilement étendu pour prendre en compte ces deux aspects, par conséquent l'intégration de ces deux aspects est notre perspective majeure.

Sélection des transitions

Jusqu'à maintenant, nous avons présenté une théorie afin de formaliser les diagrammes états-transitions en utilisant les réseaux de Petri colorés. Cette théorie est une extension de travaux précédents (par exemple [?]) en considérant la plupart des aspects syntaxiques des diagrammes états-transitions (ou en discutant comment les prendre en compte d'une manière simple). Cependant, notre traduction ne prend pas en compte le mécanisme de priorité d'UML.

Dans cette section, nous proposons une extension de notre travail en prenant en compte à la fois la notion d'envoi d'événements mais également la gestion des priorités d'UML.

Élément	pris en compte ?
états simple et composite	Oui
Région orthogonale	Oui
Pseudo-états initial / états finaux	Oui
Pseudo-état de terminaison	Non (mais trivialement extensible)
Pseudo-état histoire (plat)	Oui
Pseudo-état histoire (profond)	Non (mais trivialement extensible)
Submachine	Non (mais trivialement extensible)
Points d'entrée / de sortie	Non (mais semble faisable)
Comportements d'entrée / de sortie	Oui
Variables	Oui
Transitions : externe / locale / haut niveau / inter-niveau	Oui
Transition interne	Non (mais trivialement extensible)
Pseudo-état fork / join (basique)	Oui
Pseudo-état fork / join (implicite)	Non (mais trivialement extensible)
Pseudo-état jonction	Non (mais probablement facile)
Pseudo-état choix	Non (mais semble faisable)
Événement différé	Non
Aspects temporels	Non

TABLE 5.2 – Résumé des aspects syntaxiques considérés

1 Au-delà des raisons historiques (la version originale de ce travail [?] ne prend pas en compte
 2 les priorités), il y a plusieurs raisons de séparer la gestion des priorités du reste de la traduction.
 3 Un premier avantage de considérer une stratégie à deux étapes est que notre traduction (la
 4 partie qui ne prend pas en compte les priorités), formalisée à l'aide de [Algorithme 1](#), est un
 5 travail autonome qui prend en compte plusieurs éléments syntaxiques d'UML à l'exception des
 6 priorités d'exécutions des transitions et la gestion des conflits ; comme la plupart des traductions
 7 de la littérature qui ne considèrent pas les priorités d'exécutions des transitions et la gestion des
 8 conflits. L'[Algorithme 1](#) est un bon moyen de comparer notre travail avec les travaux précédents.
 9 Le second avantage est que le résultat de l'application de l'[Algorithme 1](#) représente une version
 10 graphique élégante et similaire à celle des diagrammes états-transitions UML ; ce qui ne sera pas
 11 le cas lors de la prise en compte des priorités. Enfin, nous pensons qu'il est intéressant de séparer
 12 la gestion des priorités du reste de la traduction, car c'est un aspect spécifique des diagrammes
 13 états-transitions qui est formalisé dans une partie dédiée dans la spécification de l'OMG ([?,
 14 p.315]).

15 Gestion des événements

16 Nous réutilisons l'idée proposée dans [?] et cela en représentant les événements donnés dans
 17 une place *CPN*. Premièrement, nous supposons dans le *CPN* un type énuméré "event" (dénote
 18 par \mathbb{E}), qui représente les différents événements utilisés dans le diagramme états-transitions source
 19 (par exemple, **a**, **b**, **c**), et également une valeur spéciale \bullet qui dénote l'absence d'événements.
 20 Ensuite, nous ajoutons une place au *CPN* avec le type "event". Cette place représentera les
 21 événements en attente de traitement, c'est-à-dire qu'elle contient un ensemble d'événements (ces
 22 événements sont sous forme de jetons *CPN*) à expédier ; par conséquent, ça modélise le groupe
 23 d'événements dans la spécification de l'OMG.

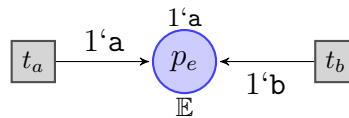


FIGURE 5.11 – Modélisation de la place des événements et de l’arrivée aléatoire des événements

1 Nous n’imposons pas la façon dont les événements apparaissent, ni l’ordre dans lequel ils
 2 sont envoyés (“The order of event dispatching is left undefined, allowing for varied scheduling
 3 algorithms”, [?, Section 14.2.3.9.1, p.314]). Cependant, nous pensons que, si besoin, le mécanisme
 4 de l’arrivée des événements ainsi que celui de l’expédition peut être facilement encodé en utilisant
 5 des fragments *CPN* supplémentaires afin de les connecter avec notre traduction. Par exemple,
 6 Figure 5.11 représente la situation où un événement peut se produire à tout moment : la place
 7 p_e (avec le type \mathbb{E}) est la place des événements qui contient (dans cet exemple) uniquement un
 8 seul événement a (“ $1'a$ ” est la notation *CPN* pour modéliser un jeton de valeur a). La transition
 9 *CPN* t_a (resp. t_b) modélise l’arrivée, dans n’importe quel ordre, à tout moment, de l’événement a
 10 (resp. b). Un schéma d’arrivée plus complexe peut aisément être modélisé par l’utilisateur si
 11 besoin.

12 Un petit contraste avec l’esprit de la spécification d’UML est le fait que les événements de
 13 terminaison ne sont pas représentés par des jetons, c’est-à-dire qu’ils n’apparaissent pas dans
 14 la place d’attente des événements. En effet, ils seront traités directement lors de la gestion des
 15 priorités : toute transition avec événement sera bloquée dans le cas où il y a la possibilité de
 16 franchir une transition avec un événement de terminaison.

17 Gestion des priorités

18 Aperçu

19 Afin de gérer les priorités, nous présentons dans cette section les différentes modifications
 20 que nous avons effectuées sur le schéma général de la traduction (vu dans la Section 5.3), mais
 21 aussi quelques ajouts. Notre solution est la suivante : premièrement, l’arrivée des événements
 22 (également l’attente qu’ils soient distribués) est maintenant représentée par des jetons *CPN*.
 23 Deuxièmement, pour chaque événement possible (par exemple, a) nous ajoutons une transition
 24 *CPN* unique qui sera présente dans le *CPN* résultat ; cette transition sera la fusion de toutes les
 25 transitions (créées par l’Algorithme 1) qui modélisent l’occurrence de l’événement a . Les priorités
 26 de franchissement des transitions seront gérées en utilisant un code segment *CPN ML* (statique)
 27 associé à cette unique transition *CPN*.

28 Afin de savoir si certaines transitions peuvent être franchies (ce qui est nécessaire afin de
 29 gérer les priorités), nous aurons besoin de tester la présence du jeton dans certaines places du
 30 *CPN*. Une solution aurait pu être la lecture de l’information sur les arcs de lecture. Cependant,
 31 afin de rendre notre schéma simple, nous proposons ce qui suit : toutes les places créées dans
 32 Section 5.3 (qui peuvent contenir des jetons de type \bullet) contiennent maintenant un jeton unique,
 33 de type \mathbb{B} . Essentiellement, si l’état simple UML est actif, alors la place correspondante en *CPN*
 34 contiendra la valeur *vrai*, sinon *faux*.

35 Nous décrivons dans ce qui suit notre solution, étape par étape.

1 **Fusion des transitions**

2 Toute transition *CPN* dans notre traduction de la [Section 5.3](#) représentant le même évé-
 3 nement (par exemple, **a**) est maintenant fusionnée : cela donne résultat à une transition représentant
 4 l'événement **a** et ayant plusieurs arcs en entrée et en sortie (ces arcs sont liés à toutes les places
 5 qui étaient liées avec les transitions fusionnées). Nous supprimons également les arcs redondants,
 6 c'est-à-dire qu'au maximum un seul arc sort d'une place vers la même transition et de même pour
 7 l'arc entrant. Tous les comportements et gardes (qui étaient attachés à la partie code segment
 8 des transitions dans la [Section 5.3](#)) sont maintenant supprimés. Nous ajoutons à la place un code
 9 CPNML que nous attachons à chaque transition *CPN* fusionnée pour implémenter le mécanisme
 10 de priorité, mais aussi afin de déterminer quelle transition UML est censée s'exécuter (notons
 11 qu'il peut y avoir aucune ou plusieurs transition qui peuvent s'exécuter); les comportements de
 12 ces transitions seront également ajoutés au code CPNML.

13 En outre, nous ajoutons également aux transitions fusionnées les arcs entrants et sortants
 14 de n'importe quelle place en relation avec une transition *CPN* représentant un événement de
 15 terminaison (dans tous le *CPN* et non pas que la région courante). Cela permet au code segment
 16 CPNML de tester si des transitions avec événements de terminaison sont activées, auquel cas.

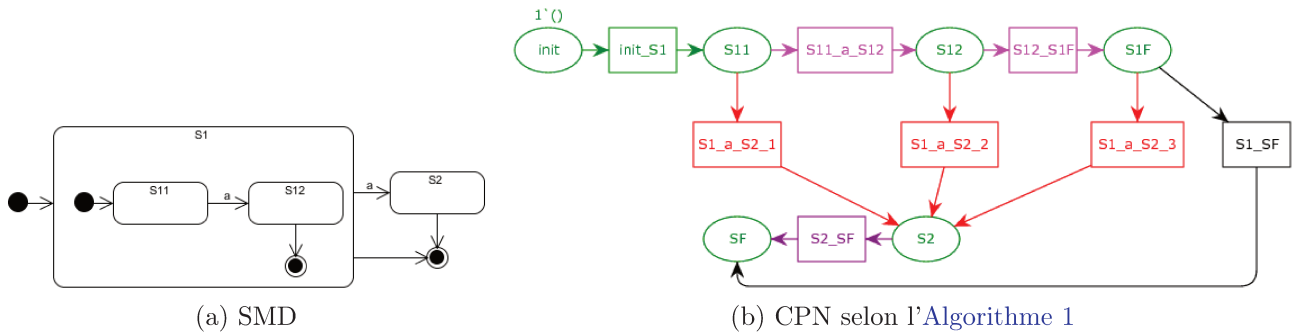


FIGURE 5.12 – Exemple avec priorités

17 Afin de tester la présence des jetons (en d'autres termes, si leurs valeurs est à *true*) dans
 18 le code CPNML, nous avons besoin de nommer les variables sur les arcs entrants et sortants :
 19 pour chaque place *p*, nous notons *pi* la valeur allant de la place vers la transition, et *po* la
 20 valeur allant de la transition vers la place. La seule exception est pour la place des événements :

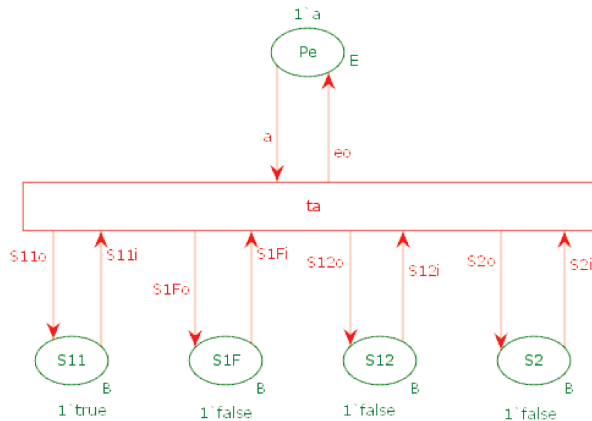


FIGURE 5.13 – Un exemple d'une transition fusionnée pour l'événement **a**

1 par exemple l'arc allant de la place des événements vers une transition avec un événement a
 2 n'acceptera comme valeur que celle de l'événement, c'est-à-dire que a . Nous dénotons la valeur
 3 de l'arc sortant allant de la transition vers la place par eo , car le jeton contiendra ou bien le
 4 même événement (dans le cas où il y a une transition avec événement de terminaison et donc
 5 l'événement ne sera pas consommé), ou bien l'événement sera consommé et donc la valeur de
 6 retour sera \bullet .

7 **Exemple.** Soit le diagramme états-transitions dans la Figure 5.12a et sa traduction corres-
 8 pondante (en utilisant l'Algorithme 1) dans la Figure 5.12b. Chaque état simple est représenté
 9 par sa place correspondante (par exemple, l'état $S11$ est représenté par la place $S11$) et les places
 10 $init, S1F, SF$ correspondent respectivement à la place contenant un jeton au marquage initial, à
 11 l'état final de $S1$ et enfin à l'état final racine. Chaque transition venant d'un état simple (avec
 12 ou sans événement) est représentée par sa transition CPN correspondante (par exemple, la tran-
 13 sition entre les états $S11$ et $S12$ est représentée par la transition $CPN S11_a_S12$). Chaque
 14 transition avec événement venant d'un état composite est représentée par un ensemble de tran-
 15 sitions CPN encodant la combinaison (par exemple, la transition avec l'événement a allant de
 16 $S1$ vers $S2$ est représentée par les transitions $CPN S1_a_S2_1, S1_a_S2_2$ et $S1_a_S2_3$).
 17 Notons qu'aucune transition sans événement venant d'un état composite a le même mécanisme
 18 qu'une transition venant d'un état simple (et est donc traduite par une seule transition).

19 Le modèle CPN résultat de l'application de l'Algorithme 1 ne prend pas en compte les
 20 priorités de franchissement des transitions. Par exemple, la transition allant de $S1$ vers l'état
 21 final racine a une priorité plus élevée que la transition entre $S1$ et $S2$ (selon l'algorithme de
 22 sélection des transitions de l'OMG). Afin de prendre en compte ce genre de situations (mais
 23 également l'algorithme de sélection de transitions de l'OMG), nous proposons un mécanisme de
 24 fusion des transitions qui permet de gérer les priorités de franchissement. Dans Figure 5.12a il
 25 y a uniquement l'événement a qui étiquette la transition de $S1$ vers $S2$ qui peut être en conflit
 26 de priorité avec l'événement de terminaison. Dans la Figure 5.13, nous donnons la transition qui
 27 fusionne toutes les transitions étiquetées par a ; cette transition est liée à la place des événements
 28 p_e , ainsi qu'à toutes les places sources ou cibles des transitions étiquetées par l'événement a dans
 29 Figure 5.12b.

30 La transition fusionnée ta remplacera toutes les transitions avec l'événement a . Les états
 31 sources/cibles de la transition ta sont les états sources/cibles des transitions précédentes repré-
 32 sentant l'événement a . Notons que les états sources ou cibles des transitions avec événements de
 33 terminaison (qui sont en conflit avec la transition ayant l'événement a) seront également liées à
 34 la transition ta . Par exemple, les états $S1F$ et $S2$ sont connectés à t_a , pas uniquement car ils
 35 sont connectés à une transition avec l'événement a dans la Figure 5.12b, mais également car ils
 36 sont les états sources d'une transition de terminaison (et qui a une priorité plus élevée que celle
 37 de la transition étiquetée avec l'événement a).

38 Algorithme de gestion des priorités

39 Nous décrivons dans cette section l'Algorithme 2 qui implémente l'algorithme de sélection
 40 des transitions mais également les priorités de franchissements des transitions (concept expliqué
 41 dans les Section 5.9.2.1 et Section 5.9.2.2). Nous réutilisons les places et les transitions obtenus
 42 après l'application de l'Algorithme 1.

43 Nous supposons dans l'Algorithme 2 une fonction $AddCodeSegments()$ qui permet de mettre
 44 à jour le code segment de chaque transition CPN , ce code gère les priorités et les conflits (ce
 45 code est décrit dans les Sections 5.9.2.4 et 5.9.2.5).

Algorithm 2: Gestion des priorités de franchissement

```

1 Add  $\textcircled{\text{Pe}}$  // Create the events pool place
2 foreach event  $e \in \mathcal{E}$  do
    // Step 1
    // Add the event to the events pool
3 Add an “e” token to  $\textcircled{\text{Pe}}$ 
    // Add the transition corresponding to the fusion, and link  $\text{Pe}$  and
     $t_e$ 
4 Add  $\textcircled{\text{Pe}} \xrightarrow{ei} t_e \xrightarrow{eo} \textcircled{\text{Pe}}$ 
    // Step 2
5 foreach transition  $t = (\mathbf{S}_1, e_1, g, (b, f), sLevel, \mathbf{S}_2) \in \mathcal{T}$  such that  $e_1 = e$  do
6 Delete  $\boxed{t}$ 
7 foreach simple state  $s_1 \in \mathbf{S}_1$  do
8     Add  $s_1 \xrightarrow{s_1^i} t_e \xrightarrow{s_1^o} s_1$ 
9 foreach simple state  $s_2 \in \mathbf{S}_2$  do
10     Add  $s_2 \xrightarrow{s_2^i} t_e \xrightarrow{s_2^o} s_2$ 
11 foreach completion transition  $(\mathbf{S}'_1, noEvent, g', (b', f'), sLevel', \mathbf{S}'_2) \in \mathcal{T}$  do
12     foreach simple state  $s \in \mathbf{S}'_1$  do
13         Add  $s \xrightarrow{s^i} t_e \xrightarrow{s^o} s$ 
14 AddCodeSegments();

```

1 Par exemple, la Figure 5.15 montre l’application de l’Algorithme 2 au modèle *CPN* (résultat
2 de l’application de l’Algorithme 1) du diagramme états-transitions de la Figure 5.14.

3 Dans un souci de lisibilité, les codes segments ne sont pas présentés.

4 Détection et gestion des completion events

5 Les événements de terminaisons sont plus prioritaires que les autres événements. Afin de
6 prendre en compte cette notion de priorité, nous traitons différemment les transitions avec évé-
7 nements de terminaisons et les transitions avec les autres événements.

8 D’une part, nous conservons les transitions *CPN* modélisons les événements de terminaisons
9 sans les changer. Cependant nous ajoutons le système de jetons *vrai/faux* où la transition *CPN*
10 qui modélise la transition de terminaison consommera tous les jetons dans les places source (ou
11 engendre des jetons dans les places destination). Cela permet de changer la valeur *vrai* des jetons
12 dans les places source en *faux* et la valeur *faux* des places destination en *vrai*. Par exemple,
13 la transition de terminaison entre *S2* et *SF* dans la Figure 5.12a est traduite par la transition
14 *CPN S1_SF* dans la Figure 5.16. En conservant la traduction des transitions de terminaisons
15 sans les changer (mise à part la gestion des jetons dans les places), nous conservons le principe
16 que chaque transition de ce type peut être franchie à n’importe quel moment (dans le cas où les
17 états source de cette transitions sont actifs).

18 D’autre part, il est nécessaire d’empêcher les transitions avec événements d’être franchies dans
19 le cas où une transition de terminaison est active. Afin de savoir si une transition de terminaison

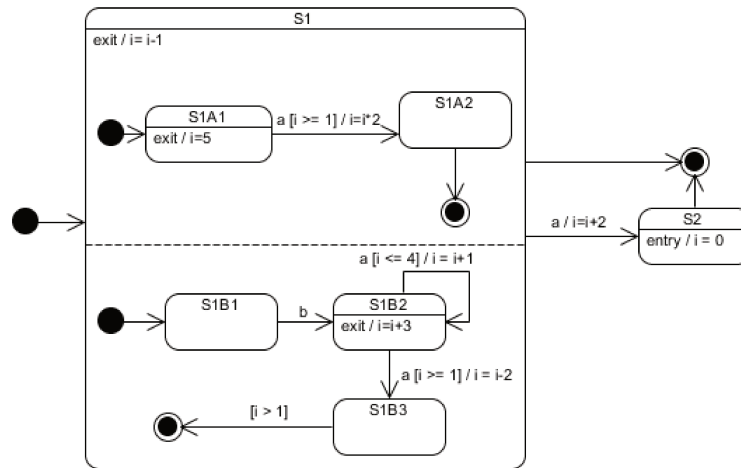


FIGURE 5.14 – Exemple avec des priorités et de la concurrence

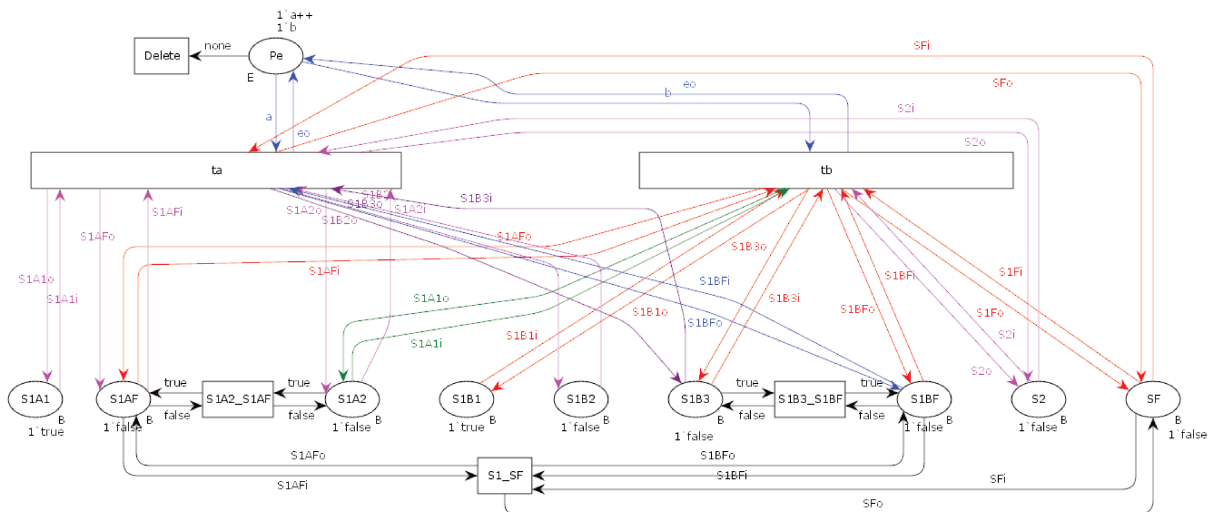


FIGURE 5.15 – Exemple avec des priorités et de la concurrence : traduction

1 est active il suffit uniquement de savoir si l'ensemble des places source de la transition *CPN*
 2 correspondante comportent des jetons avec valeur *vrai* et que la garde de cette transition est
 3 vérifiée. La fonction permettant de vérifier cela peut être calculée d'une manière statique et une
 4 fois pour toutes (par contre son exécution sera dynamique). En effet, elle peut être calculée pour
 5 toutes les transitions fusionnées en utilisant une fonction booléenne.

6 Cette fonction utilise les valeurs des jetons (*vrai* ou *faux*) et vérifie les gardes. Par exemple,
 7 la fonction qui vérifie si une transition de terminaison est active pour le diagramme dans la
 8 Figure 5.12a est la suivante (il n'y a pas de gardes dans cet exemple).

```

    9
    10 1 fonction completionEventEnabled()
    11 2     S12i or S1Fi or S2i or SFi
    
```

13 Cette fonction peut faire intervenir plus de choses que juste des disjonctions quand les transitions
 14 de terminaisons sont avec des gardes; alors que dans le cas de concurrence et pour un état
 15 orthogonal pour franchir une transition de terminaison il faut que toutes les régions soient dans
 16 leurs états finaux. Par exemple, la fonction qui vérifie si une transition de terminaison est active

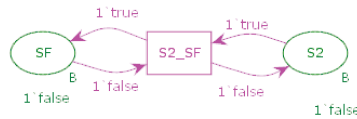


FIGURE 5.16 – Modélisation d'une transition de terminaison

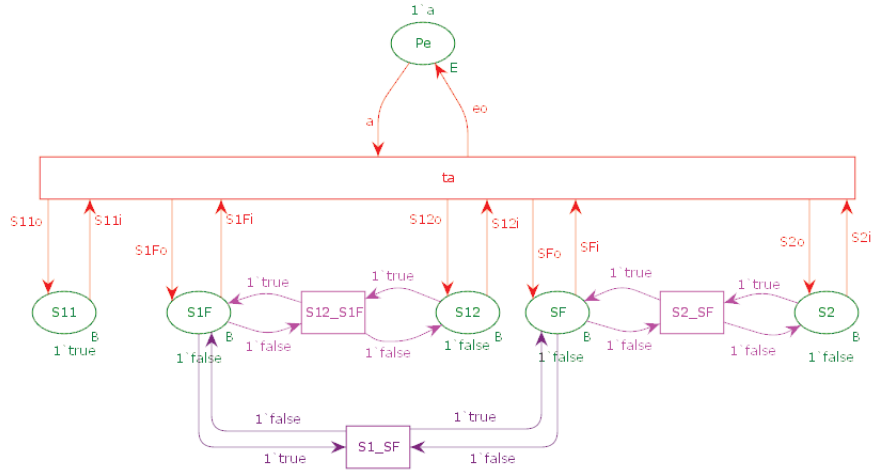


FIGURE 5.17 – Exemple avec les priorités : traduction complète sans code

```

input(S11,S12i, S1Fi, S2i, SFi);
output(eo,S11o,S12o,S1Fo,S2o, SFo);
action
  (if S12i=true or else S1Fi=true or else S2i=true or else SFi=true then (a,S11i, S12i,S1Fi,S2i, SFi) else
  let val flagS1 = true val flagR = true in
  if flagS1=true then (a,false,true,S1Fi,S2i, SFi)
  else
  if flagR=true then
  if S11i=true then (none,false,S12i,S1Fi,true, SFi)
  else
  if S12i=true then (none,S11i,false,S1Fi,true, SFi)
  else
  if S1Fi=true then (none,S11i, S12i,false,true, SFi)
  else
  (none,S11i, S12i,S1Fi,S2i, SFi)
  else
  (none,S11i, S12i,S1Fi,S2i, SFi)
  end);
  
```

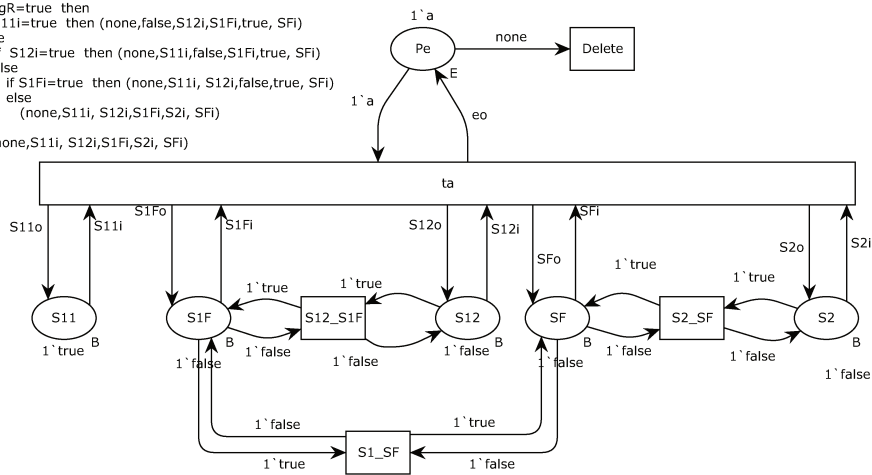


FIGURE 5.18 – Exemple avec les priorités : traduction complète avec code

1 pour l'exemple de la Figure 5.14 est la suivante :

```

2
31 function completionEventEnabled()
42   S1A2i or (S1B3i and i>1) or (S1AFi and S1BFi) or S2i or SFi
  
```

1 Détection et gestion des conflits

Après la présentation de la gestion des priorités entre événements, nous présentons dans cette section comment les transitions doivent être exécutées. À chaque apparition d'un événement a (c'est-à-dire un jeton de valeur a arrive dans la place des événements), la transition correspondante peut être franchie. Les transitions (0, une ou plus) étiquetées par cet événement doivent être franchies selon le mécanisme de priorités d'UML. Le mécanisme de priorités peut être implémenté en utilisant l'algorithme de sélection de transitions décrit dans la spécification d'UML comme suit. "States in the active state configuration are traversed starting with the innermost nested simple States and working outwards. For each State at a given level, all originating Transitions are evaluated to determine if they are enabled" [?, Section 14.2.3.9.5, p.316]. Tandis que cet algorithme doit être appliqué dynamiquement, notre code CPNML qui implémente cet algorithme peut être calculé statiquement une fois pour toutes et pour chaque événement possible. Soit un événement a , nous calculons statiquement toutes les transitions possibles qui sont étiquetées par cet événement en vérifiant leurs états source ainsi que leurs gardes, et en utilisant l'algorithme décrit au-dessus.

En plus simple, une transition CPN ou plus encodant des transitions UML étiquetées par un événement a peuvent être franchies si et seulement si : 1) un événement a peut être distribué (c'est-à-dire l'événement est présent dans la place des événements et n'est pas encore traité), et 2) aucune transition de terminaison n'est active. Cela peut être fait en vérifiant que l'événement est effectivement présent dans la place des événements, mais également en vérifiant qu'aucune transition de terminaison n'est active (en utilisant le code CPNML décrit au-dessus).

Ensuite, il est nécessaire de déterminer quelle transition va être franchie. Nous effectuons cela en utilisant du code CPNML dans la transition fusionnée comme suit :

1. Si la transition de terminaison est active, alors aucune transition étiquetée avec l'événement a peut être franchie, et le jeton de l'événement a est remis dans la place des événements (en attendant qu'aucune transition de terminaison ne soit active).
2. Sinon, nous créons un ensemble de drapeaux de type booléen (initialement avec la valeur *vrai*), un pour chaque région dans laquelle un sous-état direct est la source de la transition étiquetée avec l'événement a . Ces drapeaux indiquent si les transitions venant d'un sous-état direct dans cette région doivent encore être explorées.
3. Nous copions également toutes les variables du SMD (encodées dans des variables globales CPN) ; en effet nous appliquons quelques comportements (qui peuvent modifier ces variables) avant de tester les gardes des autres transitions.
4. Ensuite, en partant des états les plus internes source de la transition étiquetée par l'événement a , nous vérifions si toutes les places correspondantes aux états source de la transition étiquetée avec l'événement a comportent un jeton (avec la valeur *vrai*). Nous vérifions la garde de la transition en utilisant les valeurs des variables. Enfin, nous désactivons les drapeaux des régions contenant les états source de la transition (dans le cas où la transition a plusieurs états source) et cela afin d'empêcher d'autres transitions d'être franchies. Notons que les transitions dans des régions qui ne sont pas en conflit avec la région de cette transition ne sont pas désactivées.
5. Dans le cas où il n'y a pas de transition active, nous parcourons la hiérarchie.

Exemple 1. Nous expliquons d'abord notre approche sur un exemple non concurrent (*Figure 5.12a*), le code CPNML attaché à la transition fusionnée et étiquetée par l'événement a est le suivant :

```

1
21 if completionEventEnabled() then
32   (* No firing of "a" for now: return all tokens as they are, including the event
4     itself *)
53   S11i, S12i, S1Fi, S2i, SFi, "a"
64
75 else
86   (* Prepare the resulting values, initially equal to the input values *)
97   let S11o, S12o, S1Fo, S2o, SFo =
108     S11i, S12i, S1Fi, S2i, SFi
119   in
120
131   (* Define flags for all regions *)
142   let flagS1, flagR = true, true in
143
144   (* Copy variables *)
145   (* no variable to copy in this SMD *)
146
147   if flagS1 then
148     if S11i (* and no guard here *) then
149       S11o := false (* disable input places *)
150       S12o := true (* enable output places *)
151       flagR := false (* block parents regions *)
152       (* no behaviour to execute here *)
153
154     if flagR then
155       if S11i (* and no guard here *) then
156         S11o := false (* disable input places *)
157         S2o := true (* enable output places *)
158         (* no parents regions to block *)
159         (* no behaviour to execute here *)
160       else if S12i (* and no guard here *) then
161         S12o := false (* disable input places *)
162         S2o := true (* enable output places *)
163         (* no parents regions to block *)
164         (* no behaviour to execute here *)
165       else if S1Fi (* and no guard here *) then
166         S1Fo := false (* disable input places *)
167         S2o := true (* enable output places *)
168         (* no parents regions to block *)
169         (* no behaviour to execute here *)
170
171     (* Return all tokens values *)
172     S11o, S12o, S1Fo, S2o, SFo, "none"

```

46 La première partie du code CPNML permet de vérifier s'il y a des transitions de terminaison
47 active dans le diagramme états-transitions. Dans ce cas de figure tous les jetons sont remis à
48 leurs places sans les échanger, y compris le jeton de l'événement. Sinon, nous définissons des
49 valeurs de sortie (qui étiquettent les arcs allant de la transition fusionnée), et qui sont initialement
50 égales aux valeurs d'entrée². Ensuite, les drapeaux sont définis pour chaque région, par exemple
51 le drapeau de la région contenant S1 est flagR. Nous vérifions pour chaque région si une transition
52 étiquetée avec l'événement a peut être franchie, et cela en vérifions successivement que la région
53 contenant la transition est toujours active ("if flagS1 then"), que le jeton dans la place d'entrée
54 est présent ("if S11i"), et que la garde de la transition est vérifiée (dans cet exemple, il n'y a pas
55 de garde). Dans le cas où ces conditions sont vérifiées nous changeons la valeur des jetons des
56 places source (la valeur devient faux) ainsi que ceux des places destination (la valeur devient
57 true). Nous désactivons également les autres régions contenant cette transition ("flagR := false")
58 et nous exécutons les comportements en utilisant la fonction AddBehaviours(). Notons que les
59 combinaisons engendrées par la sortie des états composites (trois combinaisons en sortant de
60 l'état S1 à travers l'événement a) sont déjà calculées par l'Algorithme 1. Enfin, toutes les valeurs

2. "let ... in" est la syntaxe de CPNML pour définir les variables.

1 de sortie sont remises à leurs places respectives, et l'événement est consommé (la remise du jeton
2 • dénoté par "none" dans le code CPNML).

3 La traduction complète sans segment de code et avec segment de code du diagramme états-
4 transitions de la Figure 5.12a est présentée dans les Figures 5.17 et 5.18. La première figure
5 montre la traduction sans la présence de segment de code par rapport aux transitions et la seconde
6 figure montre la traduction complète avec la présence de segment de code dans les transitions.

7 **Exemple 2.** Considérons dans ce qui suit un exemple (Figure 5.14) comportant des comporte-
8 ments, des gardes et de la concurrence. Nous donnons le code CPNML correspondant.

```

9
101 if completionEventEnabled() then
112 (* No firing of "a" for now: return all tokens as they are, including the event
12   itself *)
133 S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi, "a"
144
155 else
166 (* Prepare the resulting values, initially equal to the input values *)
177 let S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo =
188   S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi
199 in
200
211 (* Define flags for all regions *)
222 let flagS1A, flagS1B, flagR = true, true, true in
233
244 (* Copy variables *)
255 let i' = i in
266
277 if flagS1A then
288   if S1A1i and i'>=1 then
299     S1A1o := false (* input places *)
300     S1A2o := true (* output places *)
311     flagR := false (* block parents regions *)
322     i:=5 ; i:=i*2 (* transition behaviours *)
333
344 if flagS1B then
355   if S1B2i and i'<=4 then
366     S1B2o := false (* input places *)
367     S1B2o := true (* output places *)
378     flagR := false (* block parents regions *)
389     i:=i+3 ; i:=i+1 (* transition behaviours *)
400   else if S1B2i and i'>=1 then
411     S1B2o := false (* input places *)
422     S1B3o := true (* output places *)
433     flagR := false (* block parents regions *)
444     i:=i+3 ; i:=i-2 (* transition behaviours *)
455
466 if flagR then
477   if S1A1i and S1B1i (* and no guard here *) then
488     S1A1o := false; S1B1o := false (* input places *)
499     S2o := true (* output places *)
510     (* no parents regions to block *)
521     i:=5 ; i:=i-1; i:=i+2 ; i:=0 (* transition behaviours *)
532   else if S1A1i and S1B2i (* and no guard here *) then
543     S1A1o := false; S1B2o := false (* input places *)
554     S2o := true (* output places *)
565     (* no parents regions to block *)
576     interleave(<i:=5>, <i:=i+3>) ; i:=i-1; i:=i+2 ; i:=0 (* transition behaviours *)
587   else if S1A1i and S1B3i (* and no guard here *) then
598     ... etc ...
609
610 (* Return all tokens values *)
621 S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo, "none"
632

```

63 L'ajout de la concurrence dans l'exemple a pour conséquences ce qui suit.

- 1 • Dans le cas où une transition peut être franchie dans la région d'en haut de S1, alors la
2 région d'en bas n'est pas bloquée, seulement la région racine est bloquée afin d'empêcher la
3 transition allant de S1 vers S2 d'être franchie. Par conséquent, une seule transition dans
4 la région d'en haut de S1 et/ou une seule transition dans la région d'en bas de S1 peuvent
5 être franchies.
- 6 • Dans le cas d'une région orthogonale, nous avons besoin d'énumérer toutes les combinaisons
7 des états sources, le code CPNML correspondant peut être engendré automatiquement en
8 utilisant la fonction combinations de l'Section 5.5.
- 9 • Dans le cas de la sortie de l'état S1 en étant dans des régions orthogonales, les comporte-
10 ments sont entrelacés (par exemple, en étant dans l'état S1A1 et l'état S1B2) en utilisant
11 la fonction interleave définie dans la Section 5.5.

12 Discussion

13 Nous discutons dans cette section certains aspects syntaxiques. Un premier point concerne
14 la gestion des événements, notre traduction respecte le schéma de priorité d'UML, et en accord
15 avec le fait qu'un événement qui ne correspond à aucune transition lors de son occurrence est
16 abandonné (“If no Transition is enabled and the corresponding Event type is not in any of
17 the deferrableTriggers lists of the active state configuration, the dispatched Event occurrence is
18 discarded and the run-to-completion step is completed trivially” [?, Section 14.2.3.9.1, p.314]).
19 En effet, lors de l'exécution du code CPNML associé à la transition a, si aucune transition de
20 terminaison n'est active, mais qu'aucune transition ne peut être franchie (à cause de l'absence
21 des d'états source actifs, ou qu'aucune garde n'est satisfaite), alors le jeton de l'événement est
22 abandonné. En revanche, si une transition de terminaison est active, alors l'événement n'est
23 pas abandonné mais uniquement remis dans la place des événements et donc l'occurrence de
24 l'événement est reportée jusqu'à ce qu'aucune transition de terminaison soit active.

25 Un second point est que notre code CPNML dans le code segment des transitions est re-
26 lativement complexe. Cependant, rappelons que la construction de ce code (ce qui nécessite le
27 parcours de la structure hiérarchique des états et des transitions) est fait seulement une fois
28 pour toutes (pour chaque événement) lors de la phase de traduction, et non pas durant la phase
29 du model checking. Durant la phase du model checking, seulement l'exécution de ce code (qui
30 consiste à vérifier des expressions booléennes et les exécutions des comportements) est effectuée.

31 Le troisième point concerne la résolution des conflits, bien que nous sommes conformes à
32 la résolution du conflit mentionnée par UML, notre travail a une limitation, qui apparaît dans
33 le cas de deux transitions en conflit (autrement dit, deux transitions venant du même état,
34 avec les deux gardes vérifiées). Cette situation est ambiguë dans la spécification de l'OMG (“If
35 that event occurs and both guard conditions are true, then at most one of those Transitions
36 can fire in a given run-to-completion step.” [?, Section 14.2.3.9.3, p.315]). Par exemple, dans la
37 Figure 5.14, si $1 \leq i \leq 4$ et l'événement a apparaît, alors les deux transitions vers S1B2 et S1B3
38 peut être franchies. Dans notre approche, nous résolvons cette situation d'une manière statique :
39 seulement une transition sera franchie (ce point est correcte), et toujours la même, en faite la
40 première transition qui apparaît dans le code CPNML sera franchie, ce qui dépend de l'ordre du
41 parcours des états hiérarchiques durant la traduction. Dans cet exemple (Figure 5.14), la première
42 transition dans le code est la transition “self-loop” vers l'état S1B2. Afin de rendre le choix plus
43 flexible, et bien que le choix statique d'une seule transition reste en accord avec la spécification,
44 nous proposons des solutions pour des travaux futurs : (i) interdire des gardes vérifiées en même
45 temps pour des transitions ayant le même état source (ii) nécessité d'un mécanisme de priorité

1 dans le diagramme UML que notre traduction implémentera (iii) permettre un choix aléatoire
2 de transitions à franchir.

3 Une remarque mineure que nous notons est que si un événement est consommé, une valeur
4 \bullet est renvoyée à la place des événements, et par conséquent il y restera pour toujours Ceci
5 est cohérent avec la sémantique d'UML, seulement cela conduira à une accumulation de jetons
6 inutiles dans la place des événements, ce qui pourra ralentir le model checking. Une possibilité
7 afin d'éviter cette situation est de supprimer les jetons inutiles en utilisant une transition *CPN*
8 additionnelle qui a comme source la place des événements mais ne possède pas d'état cible, et
9 accepte uniquement les jetons avec la valeur \bullet .

10 Enfin, rappelons que les événements différés ne sont pas considérés dans notre travail. La
11 considération de ces derniers (ce qui sera certainement pas simple) peut se faire en utilisant
12 notre place d'événements : dans le cas où un événement doit être différé alors il sera déplacé à
13 une place (la place qui comportera tous les événements différés); ensuite à chaque fois qu'une
14 transition accepte l'événement différé; elle aura une priorité plus élevée par rapport aux autres
15 transitions avec événements.

16 Conclusion

17 Nous avons présenté dans ce chapitre notre traduction des diagrammes états-transitions vers
18 les réseaux de Petri colorés. Cette traduction prend en compte un ensemble d'éléments syn-
19 taxiques (tels que les états composites, orthogonaux, la concurrence, la hiérarchie des comporte-
20 ments et des états, etc). Elle est basée sur deux algorithmes : le premier ([Algorithme 1](#)) traduit
21 les éléments syntaxiques des diagrammes états-transitions, le second ([Algorithme 2](#)) traduit le
22 mécanisme de sélection des transitions et le principe de priorités présent dans la spécification de
23 l'OMG.

24 Nous présentons dans les deux chapitre suivants (c'est-à-dire [Chapitres 6](#) et [7](#)) une application
25 de notre traduction sur une étude de cas ainsi que l'implémentation de cette dernière.

Chapitre 6

Étude de cas

Contents

6.1	Introduction	155
6.2	Étude de cas : lecteur de CD	155
6.3	Conclusion	165

Introduction

Étude de cas : lecteur de CD

Dans cette section, nous illustrons les diagrammes états-transitions en utilisant l'exemple du lecteur de CD. L'exemple est décrit dans la [Section 6.2.1](#), l'application de nos algorithmes dans la [Section 6.2.3](#), quelques limites concernant le comportement du dans la [Section 6.2.4](#), une modification du modèle dans la [Section 6.2.4](#) et enfin une phase de vérification dans la [Section 6.2.5](#).

Description

La description suivante représente la spécification du lecteur de CD :

Le dispositif est le lecteur de CD avec cinq boutons : load (pour charger un CD), play (pour lire un CD), stop (pour arrêter la lecture du CD), pause (pour stopper momentanément la lecture) et enfin off (pour éteindre le lecteur). L'utilisation standard du lecteur de CD est la suivante : avant que l'utilisateur puisse utiliser le lecteur pour la première fois, le tiroir est fermé, il n'y a aucun CD à l'intérieur du lecteur et aucun bouton n'est pressé. Quand l'utilisateur presse le bouton load, le tiroir s'ouvre et si l'utilisateur presse le bouton encore un fois alors le tiroir se ferme. Si le tiroir est fermé et il y a un CD à l'intérieur, alors le lecteur est prêt à le lire, et la piste courante est fixée à la première piste, sinon rien ne se produit. Quand l'utilisateur presse le bouton play, s'il n'y a aucun CD dans le tiroir rien ne se produit, sinon le lecteur commence à lire la piste courante du CD. Si l'utilisateur presse le bouton pause pendant la lecture du CD, alors le lecteur arrête la lecture et le tiroir reste fermé. Si l'utilisateur presse le bouton pause encore une fois, alors le lecteur continue à lire le CD. Le lecteur dispose d'une lumière qui en étant allumée permet d'indiquer quand le lecteur lit le CD, et indique que le lecteur est éteint ou qu'il n'y a pas de CD dedans en étant éteinte. Enfin, si l'utilisateur presse le bouton stop, alors le lecteur arrête la lecture. L'utilisateur peut ensuite récupérer le CD (en pressant le bouton load), ou peut recommencer la lecture (en pressant le bouton play). À tout moment le lecteur peut être éteint en pressant le bouton off.

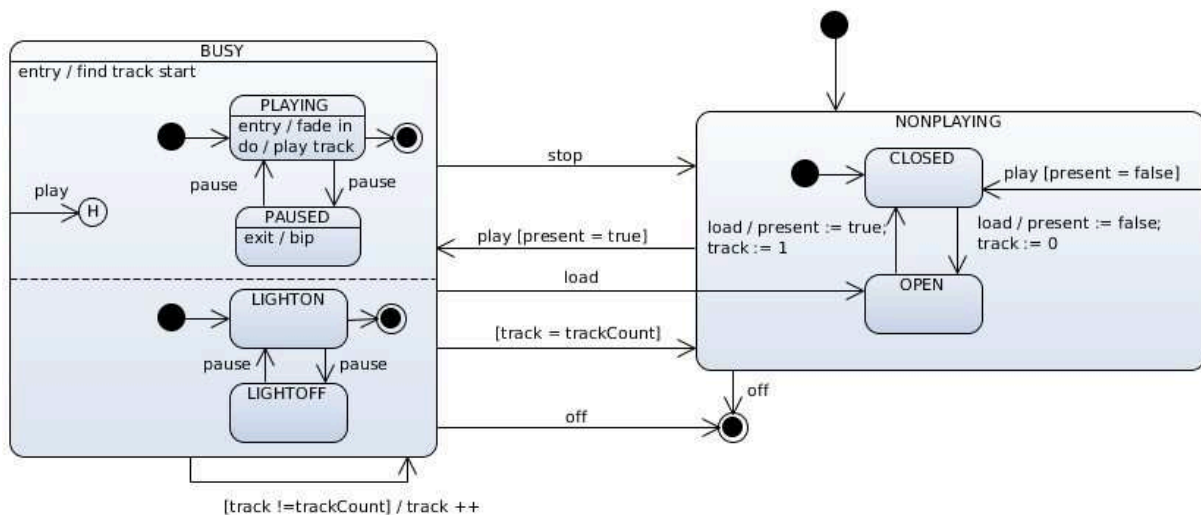


FIGURE 6.1 – Exemple du lecteur de CD

La Figure 6.1 montre le diagramme états-transitions du lecteur de CD¹.

La variable booléenne **present** encode la présence du CD dans le lecteur, et la variable entière **track** encode le numéro de la piste en cours de lecture. Nous supposons que **trackCount** est une constante qui a une valeur prédéfinie (par exemple, 5).

Nous décrivons informellement dans ce qui suit chaque état simple du lecteur de CD :

- **CLOSED** dénote l'état où le lecteur est fermé et il n'y a aucune lecture en cours.
- **OPEN** dénote l'état où le lecteur est ouvert et il n'y a aucune lecture en cours.
- **PLAYING** dénote l'état où le lecteur est en cours de lecture d'une piste (jouer de la musique).

1. Cet exemple est inspiré d'un modèle présenté dans [?], auquel nous avons ajouté certains éléments tels que les états orthogonaux.

- 1 • **PAUSED** dénote l'état où le lecteur est en mode pause (la lecture de la piste est suspendue).
- 2 • **LIGHTON** dénote l'état où la lumière (la lampe sur le devant du lecteur) est allumée.
- 3 • **LIGHTOFF** dénote l'état où la lumière est éteinte.

4 Le lecteur passe de l'état **NONPLAYING** à l'état **BUSY** lorsque l'utilisateur presse le bouton **play**
 5 et que le CD est présent dans le lecteur (ce qui est modélisé par la garde **present = true**). Le
 6 lecteur passe de l'état **CLOSED** à l'état **OPEN** lorsque l'utilisateur presse le bouton **load** et vice
 7 versa. L'utilisateur a la possibilité de passer de l'état **PLAYING** vers l'état **PAUSED** en pressant
 8 le bouton **pause**. Le même bouton permet le changement de l'état **LIGHTON** à l'état **LIGHTOFF**
 9 (et vice versa). Si l'utilisateur presse le bouton **load** alors le lecteur lit toutes les pistes du CD
 10 (représenté par une transition avec une garde sur la variable **track**); cependant si l'utilisateur
 11 presse le bouton **load** pendant la lecture, alors le lecteur passe de l'état **BUSY** à l'état **OPEN**.
 12 L'utilisateur peut éteindre le lecteur (transition étiquetée avec l'événement **off** vers l'état final
 13 racine).

14 Cet exemple (le lecteur de CD) est relativement simple : deux variables, deux états compo-
 15 sites, trois états finaux, six états simples, un pseudo-états histoire plat, et quelques transitions.
 16 Cependant, il n'est pas clair si les propriétés suivantes sont vérifiées pour toutes les configurations
 17 du système :

- 18 1. "le lecteur de CD ne peut être ouvert et fermé en même temps"
- 19 2. "si le lecteur de CD est dans l'état **PLAYING** alors le CD est inséré dans le lecteur"
- 20 3. "si le lecteur est en pause alors la lumière est éteinte"
- 21 4. "la valeur de **track** ne dépasse jamais **trackCount**"

22 Ces propriétés ne peuvent être vérifiées sans une sémantique formelle. Dans les sections qui
 23 suivent, nous traduisons le SMD correspondant au lecteur de CD afin de vérifier de telles pro-
 24 priétés.

25 Traduction du diagramme états-transitions du lecteur de CD

26 Nous présentons dans cette section l'application de nos algorithmes sur l'exemple du lecteur
 27 de CD (présenté dans la [Section 6.2.1](#)) La traduction a été faite manuellement en appliquant
 28 l'[Algorithme 1](#) et en faisant la fusion des transitions selon la [Section 5.9.2.2](#) (une implémentation
 29 afin d'automatiser le processus de traduction est le sujet d'un travail en cours [Chapitre 7](#)).

30 Nous gérons l'arrivée des événements avec un schéma légèrement différent de celui dans
 31 la [Section 5.9.1](#) : afin de modéliser la situation où un événement peut arriver à n'importe quel
 32 moment, qu'il peut être distribué à n'importe quel moment, nous gardons un événement de chaque
 33 valeur possible dans la place des événements. Afin de faire cela, nous ajoutons initialement un
 34 événement de chaque valeur à la place des événements (c'est-à-dire que **load**, **off**, **pause**, **play**,
 35 **stop**); ensuite, au lieu de consommer l'événement lors de sa distribution, l'événement est renvoyé
 36 dans la place des événements. Ainsi, chaque événement est disponible à tout moment en gardant
 37 l'espace d'états fini (ayant un nombre non-borné d'événements en attente et/ou des valeurs des
 38 jetons • dans la place des événements peut conduire à un espace d'états infini en tout cas si le
 39 model-checker n'utilise pas des techniques de réduction appropriées).

1 Limites des formalismes discrets pour des comportements continus

2 Une simulation rapide en utilisant CPN Tools nous a permis de réaliser que la traduction
 3 du lecteur n'était pas le bon résultat que nous voulions. En particulier, dans le cas où l'état
 4 **BUSY** est actif, les transitions étiquetées avec les événements *pause / off / load / stop* ne sont
 5 jamais exécutées. Cela est dû à la présence du comportement "do" dans l'état **PLAYING**. Nous
 6 supposons que le comportement "do" peut s'exécuter zéro, une ou plusieurs fois. En conséquence,
 7 la transition de terminaison de **PLAYING** peut être exécutée à n'importe quel moment et, comme
 8 la transition de terminaison est plus prioritaire qu'une transition avec événement, la transition
 9 (avec l'événement **pause**) vers **PAUSED** ne sera pas exécutée à cause de la priorité de la transition
 10 de terminaison. De même pour l'état **LIGHTON**. Cela n'est pas uniquement une limite de notre
 11 traduction, mais c'est le cas de tout formalisme discret (tels que les automates, les réseaux de
 12 Petri, CSP, Promela, etc). En effet, sans un formalisme continu (qui permet d'exprimer l'aspect
 13 temporel continu), il est impossible de modéliser le comportement "playing a track" de manière
 14 continue, autrement dit dès que le comportement termine son exécution il sera possible de sortir
 15 de la région qui le contient.

16 Rappelons que cette étude de cas est une version modifiée de l'exemple dans [?], et que le
 17 problème du comportement do n'a pas été détectée car la gestion de priorités entre transitions
 18 n'était pas considérée.

19 **Remarque :** Techniquement, à cause de la présence du comportement "do" dans l'état
 20 **PLAYING**, il est possible d'avoir des exécutions où le comportement "do" est exécuté, et par
 21 conséquent la transition avec l'événement de terminaison ne peut pas empêcher d'autres transi-
 22 tions avec événements d'être exécutées. Cependant, cela n'est pas possible dans notre traduction,
 23 car la transition avec événement peut être franchie empêche d'autres transitions d'être exécutées.
 24 Afin d'éviter ce cas de figure, une possibilité est de modifier notre encodage afin de permettre
 25 aux transitions de terminaison ou avec événement d'être exécutées (pour un état donné). Cepen-
 26 dant, afin de conserver une cohérence avec la sémantique, si une transition avec événement est
 27 exécutée, alors la transition de terminaison peut être exécutée uniquement après une exécution
 28 du comportement "do". Cela peut être facilement géré en utilisant une variable globale (pour
 29 chaque transition "do") dans le *CPN* résultat afin de bloquer la transition de terminaison dans
 30 le cas où une transition avec événement est exécutée jusqu'à une exécution de la transition du
 31 comportement "do".

32 Modification du modèle

33 Dans notre étude de cas (présentée dans [Section 6.2](#), [Figure 6.1](#)), nous suggérons une dis-
 34 crétisation de la nature continue de "playing a track". Comme première étape, nous supprimons
 35 le comportement "do" de l'état **PLAYING**. Ensuite, nous représentons le comportement do par
 36 le passage d'un certain temps (discret). Ceci dit, nous supposons que la lecture du CD a une
 37 durée `minuteCount` (tous les morceaux ont la même durée afin que le modèle reste simple).
 38 Nous créons une variable `minute`, avec comme valeur initiale 0 lors du début d'une nouvelle lec-
 39 ture. Le passage du temps est représentée par l'incrémentación de la valeur de `minute` (tant que
 40 `minute < minuteCount`) à chaque occurrence de l'événement `timeElapse` quand l'état **PLAYING**
 41 est actif. Enfin, les deux régions de l'état **BUSY** peuvent atteindre leurs états finaux seulement
 42 dans le cas où `minute = minuteCount`. De cette manière les transitions de terminaison seront
 43 plus prioritaires que dans le cas où la lecture du morceau atteint la fin. Notons que l'exten-
 44 sion de notre travail avec l'aspect temporel permettra de considérer les études de cas avec des
 45 comportements "do" ayant une nature continue sans avoir besoin de faire une discrétisation.

46 Le modèle modifié est présenté dans la [Figure 6.2](#).

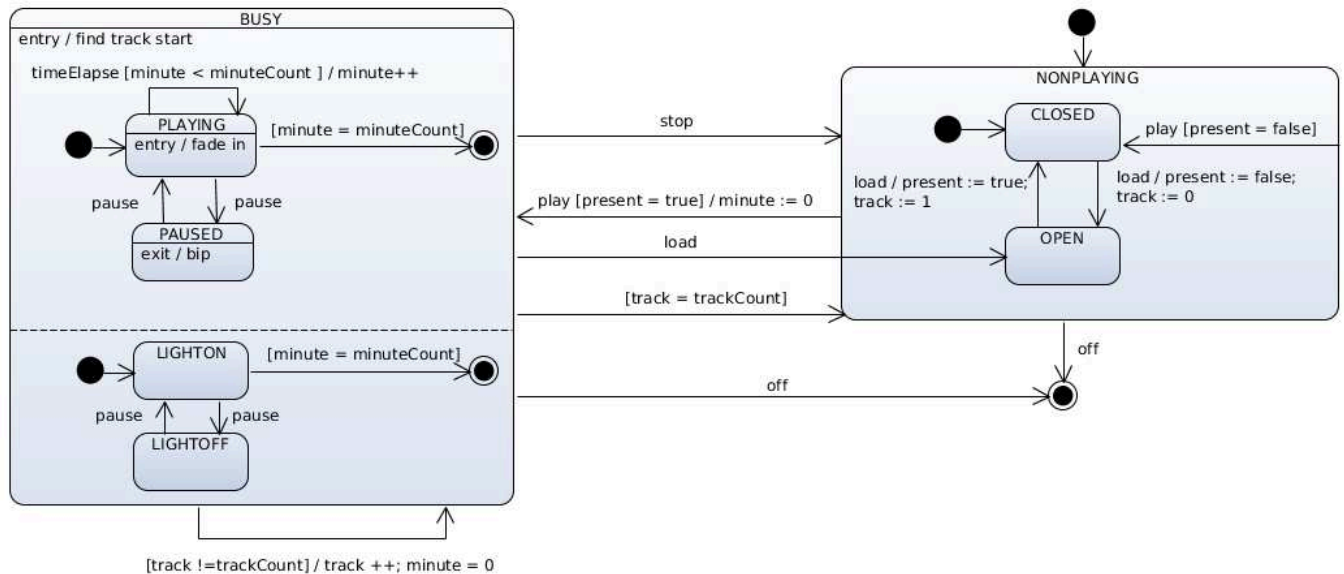


FIGURE 6.2 – Exemple du lecteur de CD (version modifiée)

1 Vérification

2 Nous engendrons le *CPN* correspondant au SMD modifié du lecteur de CD (voir [Figure 6.2](#)).
 3 Le résultat de la traduction est montré dans la [Figure 6.3](#) sans présence du segment de code
 4 CPNML pour des raisons de lisibilité. La place *Pe* représente la place des événements et qui
 5 initialement contient tous les événements. Toutes les autres places modélisent les états simples
 6 en incluant les états finaux du SMD. Les transitions fusionnées sont celles avec les nombreux
 7 arcs entrants et sortants. Notre modèle CPN Tools incluant tous les segments de code ainsi que
 8 les commandes de vérifications des propriétés est disponible en ligne².

9 Nous présentons également dans la [Figure 6.4](#) la version simplifiée de la traduction du lecteur
 10 de CD (la version modifiée) en utilisant la notion d'arc double (c'est-à-dire un arc à deux sens).
 11 Cependant, cette version est uniquement utilisée pour une visibilité meilleure car elle ne permet
 12 pas de faire de la vérification. La raison est que l'utilisation des arcs à deux sens n'est pas
 13 une option qui existe dans CPN Tools et par conséquent elle n'est pas prise en compte lors de
 14 la simulation ou vérification. Enfin nous présentons dans la [Figure 6.5](#) la version complète de
 15 traduction de l'exemple du lecteur de CD avec la présence du segment de code dans les transitions.

2. <http://www.lipn.fr/~benmoussa/SMD2CPN/>

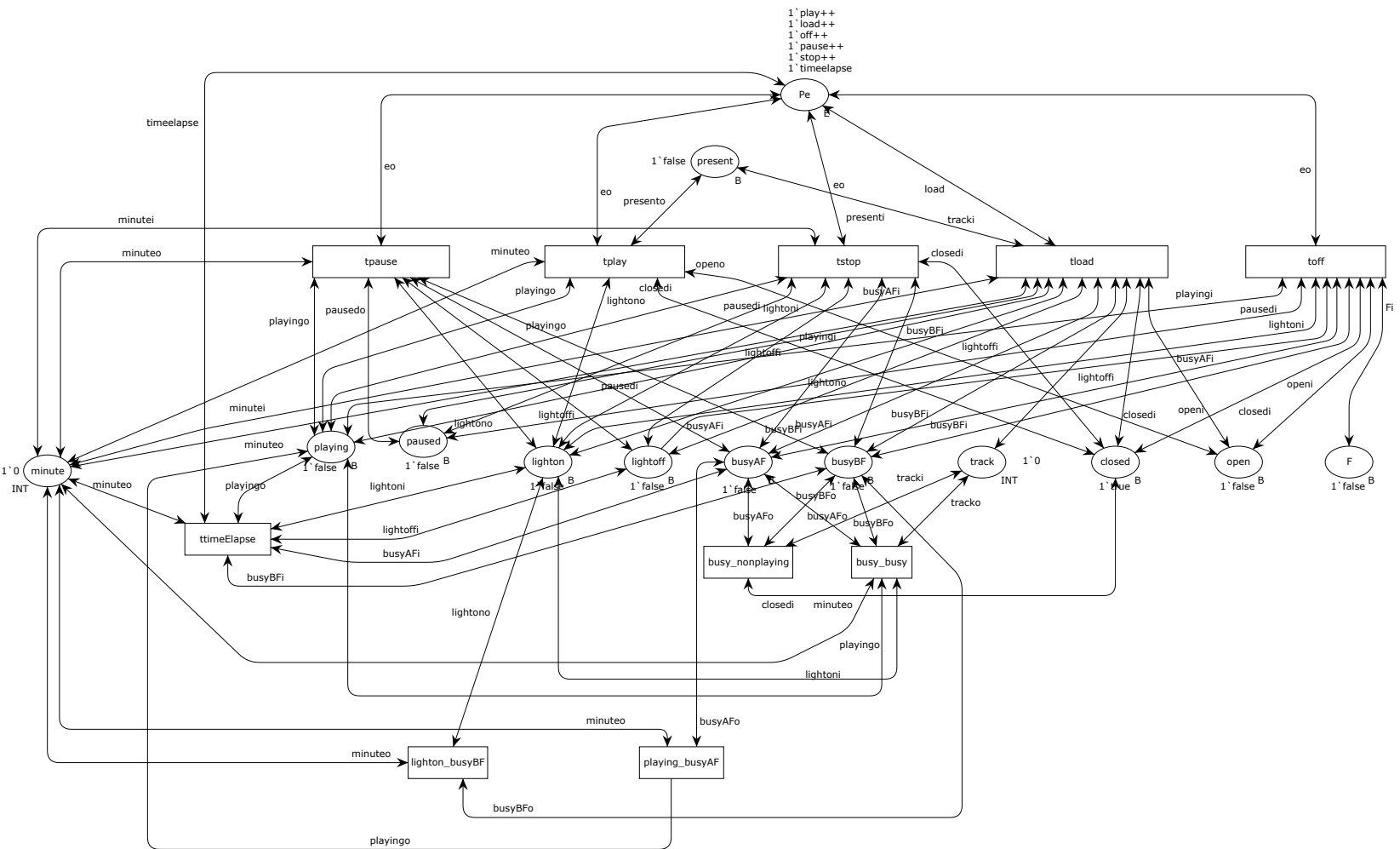


FIGURE 6.4 – Traduction en CPN de la version révisée de l'exemple du lecteur de CD : version simplifiée

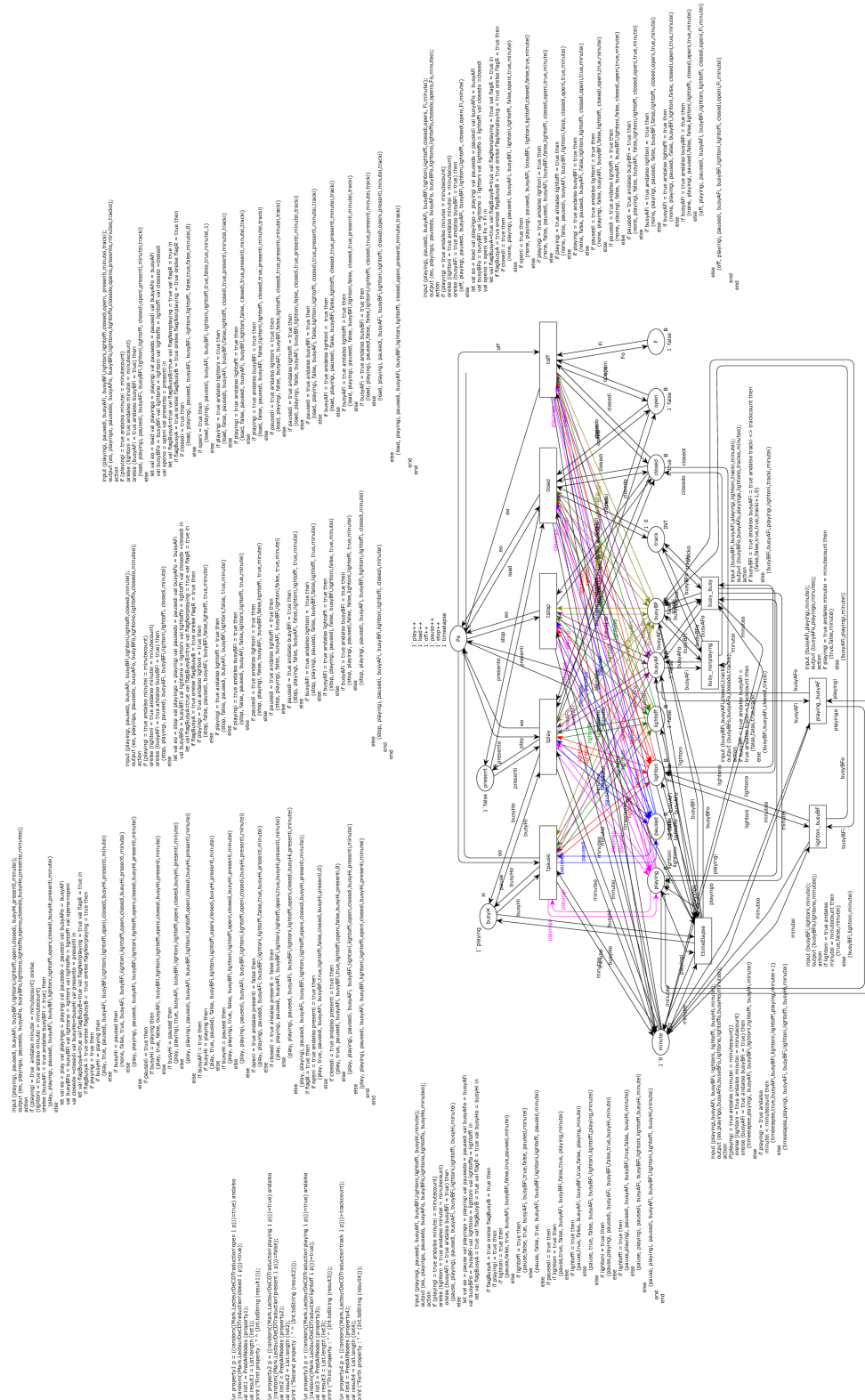


FIGURE 6.5 – Traduction en CPN de la version révisée de l'exemple du lecteur de CD : version complète

- 1 Nous utilisons CPN Tools 4.0 [?] pour vérifier les propriétés sur notre modèle modifié du
- 2 lecteur de CD.

CPN Tools n'est pas un model-checker à la volée, et il a besoin d'engendrer tout l'espace d'états avant de faire la vérification des propriétés. La première phase de vérification est de spécifier les propriétés en utilisant le langage CPN ML (le langage utilisé dans CPN Tools pour les segments de code ainsi que la spécification des propriétés). Dans la seconde phase, nous engendrons l'espace d'états en utilisant le générateur d'espace d'états de CPN Tools. Enfin, nous évaluons les propriétés sur l'espace d'états engendré en utilisant CPN Tools.

Nous fixons la valeur de `trackCount` à 5 et la valeur de `minuteCount` à 3. L'espace d'état engendré en utilisant CPN Tools comporte 159 états symboliques et 1551 arcs (calculés en 4.0s en utilisant un ordinateur avec un Intel core i5-4570 CPU 3.2GHz with 8 GiB RAM). Nous formulons, après la génération de l'espace d'état, les propriétés présentées dans [Section 6.2](#).

Propriété 1 (“le lecteur de CD ne peut être ouvert et fermé en même temps”)

Afin de vérifier cette propriété il suffit de vérifier dans chaque marquage de l'espace d'état qu'il n'y a aucun jeton dans `closed` et `open` en même temps. La propriété est validée instantanément par CPN Tools sachant que l'espace d'état est déjà engendré et que sa taille est raisonnable (la même situation s'applique pour les autres propriétés). Le code CPN ML correspondant à la première propriété est le suivant.

```

17 fun property1 p = ((random((Mark.LecteurDeCDTraductionLastVersion 'open 1 p)))=true)
18 andalso
19 ((random((Mark.LecteurDeCDTraductionLastVersion 'closed 1 p)))=true);
202 val list1 = PredAllNodes (property1);
213 val result1 = List.length (list1);
224 print ("First property : " ^ (Int.toString (result1)));
235

```

Propriété 2 (“si le lecteur de CD est dans l'état PLAYING alors le CD est inséré dans le lecteur”)

Vérifier cette propriété revient à vérifier si dans le cas où il y a un jeton dans la place `playing` alors forcément le jeton de la place `present` est égale `true`. La propriété est prouvée avec CPN Tools et aucun cas contradictoire n'est possible. Le code CPN ML correspondant à la seconde propriété est le suivant.

```

30 fun property2 p = ((random((Mark.LecteurDeCDTraductionLastVersion 'playing 1 p)))=true)
31 andalso
32 ((random((Mark.LecteurDeCDTraductionLastVersion 'present 1 p)))=false);
333 val list2 = PredAllNodes (property2);
343 val result2 = List.length (list2);
354 print ("Second property : " ^ (Int.toString (result2)));
365

```

Propriété 3 (“si le lecteur est en pause alors la lumière est éteinte”)

Vérifier cette propriété revient à vérifier si dans le cas où il y a un jeton dans `paused` alors il y a forcément un jeton dans la place `lightoff`. La propriété est vérifiée avec CPN Tools et son code CPN ML est le suivant.

```

42 fun property3 p = ((random((Mark.LecteurDeCDTraductionLastVersion 'playing 1 p)))=true)
43 andalso
44 ((random((Mark.LecteurDeCDTraductionLastVersion 'lightoff 1 p)))=true);
452 val list3 = PredAllNodes (property3);
463 val result3 = List.length (list3);
474 print ("Third property : " ^ (Int.toString (result3)));
485

```

1 **Propriété 4** (“la valeur de `track` ne dépasse jamais `trackCount`”) Afin de valider
 2 cette propriété il suffit de vérifier si dans n’importe quel état de l’espace d’états que `track` \leq
 3 `trackCount`. La propriété est vérifiée avec CPN Tools et son code CPN ML est le suivant.

```

51 fun property4 p = ((random((Mark.LecteurDeCDTraductionLastVersion 'track 1 p)))>trackcount
6   );
72 val list4 = PredAllNodes (property4);
83 val result4 = List.length (list4);
104 print ("Forth property : " ^ (Int.toString (result4)));

```

11 La Figure 6.6 montre l’évaluation des quatre propriétés présentée en haut après génération
 12 de l’espace d’états et en utilisant CPN Tools. Le résultat est montré dans le carré vert au dessus
 des expressions CPN ML des propriétés.

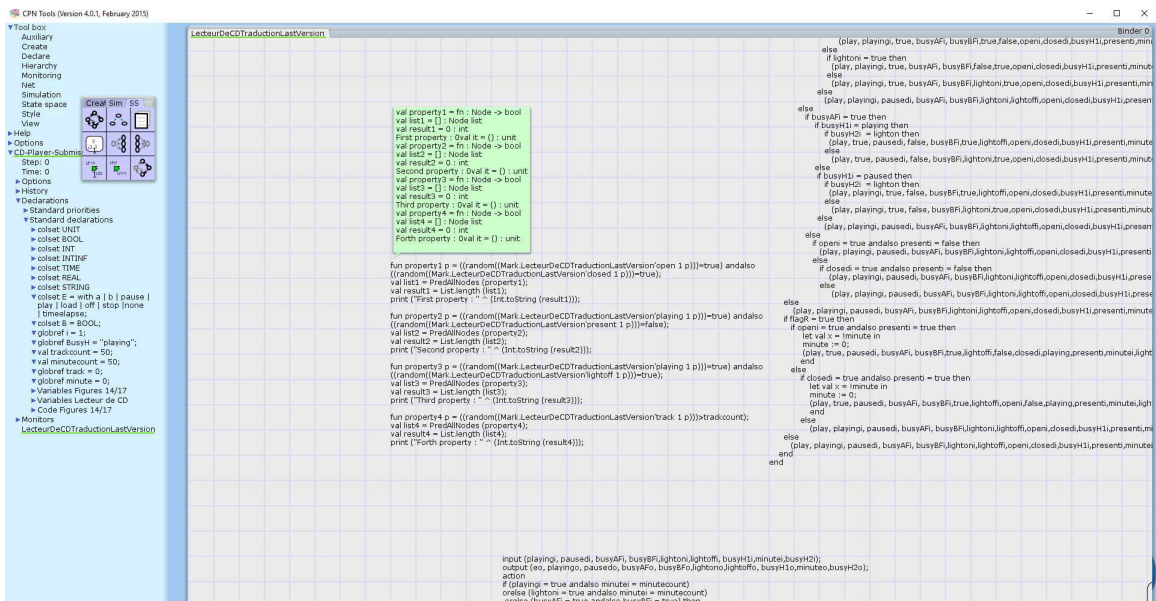


FIGURE 6.6 – Évaluation des propriétés dans CPN Tools

14 **Remarque** Notre vérification a été effectuée en utilisant des valeurs fixes pour les variables
 15 `trackCount` et `minuteCount`. Effectuer la vérification en utilisant *n’importe* quelle valeur pour
 16 `trackCount` et `minuteCount` nécessite des techniques de vérification paramétrées (ce qui sort du
 17 sujet de ce travail et de cette thèse).

18 Étude du passage à l’échelle

19 Nous présentons dans ce qui suit le tableau récapitulatif des différents résultats de la géné-
 20 ration de l’espace d’états en fonction des valeurs des variables `trackcount` et `minuteCount`. Nous
 21 présentons comme éléments le temps de génération de l’espace d’états, la taille de l’espace d’états
 22 (nombre des nœuds et des transitions), et le temps de génération du rapport comportant les dif-
 23 férentes informations sur l’espace d’états. Nous utilisons l’outil de génération d’espace d’états de
 24 CPN Tools pour faire nos calculs. CPN Tools génère l’espace d’état (le temps de génération est
 25 indiqué dans le tableau par la colonne *TempsGénération*) et ensuite il vérifie les priorités (où le
 26 temps de vérification est instantané).

tracccount/minutecount	TempsGénération	Nœuds	Transitions
5/3	3,40sec	159	1551
10/10	3,57sec	887	8647
20/20	4,49sec	3367	32827
30/30	6,42sec	7447	72607
40/40	08,97sec	13127	127987
50/50	13,63sec	20407	198967
100/100	1min,56sec	80807	787867

TABLE 6.1 – Performances de la génération de l'espace d'état du lecteur de CD

1 Conclusion

2 Afin d'appliquer nos algorithmes et faire de la vérification nous avons présenté dans ce cha-
 3 pitre la traduction de l'exemple du lecteur de CD. En partant du diagrammes états-transitions du
 4 lecteur de CD nous avons engendré le réseaux de Petri colorés correspondant. Nous avons montré
 5 une version sans la présence des segments de codes dans les transitions et une version complète
 6 avec du code. Enfin, nous avons vérifié quelques propriétés en utilisant l'outil CPN Tools.

Chapitre 7

Implémentation de la traduction

Contents

7.1	Introduction	166
7.2	Implémentation	167
7.3	UML to CPN tool	168
7.4	Conclusion	174

Introduction

Nous avons présenté dans le [Section 5.1](#) la transformation des diagrammes états-transitions vers les réseaux de Petri colorés. Afin d’automatiser cette transformation, nous proposons dans ce chapitre deux implémentations. La première implémentation (présentée dans la [Section 7.2](#)) est une exploration des techniques de l’ingénierie dirigée par les modèles. La technique la plus utilisée est celle de la transformation des modèles, c’est-à-dire transformation de modèle vers modèle. L’avantage de cette technique de transformation est la présence d’outils intégrés l’implémentant, et il n’y a pas besoin d’utiliser d’analyseur syntaxique (*parser*). Cependant, afin d’utiliser ce genre de techniques il y a besoin d’avoir en entrée le modèle et le méta-modèle du formalisme d’entrée ainsi que le méta-modèle du formalisme de sortie afin d’engendrer le modèle correspondant. Rappelons qu’un méta-modèle définit le langage d’expression d’un modèle. C’est un niveau d’abstraction supérieur car il est plus générique que le modèle, par conséquent le modèle doit être conforme au méta-modèle pour un système donné.

Dans notre traduction, nous utilisons comme formalisme d’entrée les diagrammes états-transitions (qui possèdent un méta-modèle officiel présenté dans [?]) et en sortie les réseaux de Petri colorés qui malheureusement ne possèdent pas de méta-modèle (officiel) qui prenne en compte tous les éléments dont nous avons besoin. Il existe des travaux tels que [?] qui propose un méta-modèles pour les réseaux de Petri et leurs variantes mais qui ne prend pas en compte par exemple la gestion des segments de code dans les transitions. Une alternative est d’utiliser des outils basés sur la transformation de modèles vers du texte (par exemple, [Acceleo](#)¹). Ces outils prennent en entrée le modèle et le méta-modèle du formalisme d’entrée et engendrent le modèle (sous forme de texte) du formalisme en sortie sans avoir besoin de son méta-modèle. Cependant, nous avons réalisé que l’utilisation de ces outils est limitée qu’il y avait un certain

1. <https://www.eclipse.org/acceleo>

1 nombre d'inconvénients, par exemple l'absence de variables ou de structures de données (deux
2 notions nécessaires pour le fonctionnement de nos algorithmes).

3 En se basant sur notre expérience de l'utilisation des techniques de l'ingénierie dirigée par
4 les modèles, nous proposons une implémentation (présentée en [Section 7.3](#)) en développant notre
5 propre outil. Cet outil (UML2CPN) est développé en utilisant java et des analyseurs syntaxiques.

6 Implémentation avec Acceleo

7 Nous présentons dans cette section [?] un travail d'exploration afin d'implémenter une pre-
8 mière version de la traduction des diagrammes états-transitions vers les réseaux de Petri colo-
9 rés [?] en utilisant un outil de transformation de texte vers du modèle (Acceleo).

10 Acceleo est un outil permettant d'implémenter des générateurs de code. Il est basé sur les
11 techniques modèle vers texte et surtout ne nécessite pas d'avoir le méta-modèle du formalisme
12 de sortie. L'autre avantage d'Acceleo (et un avantage des model2text en général) est que nous
13 pouvons définir le format du code engendré pour qu'il soit conforme à l'outil qui l'utilise. Nous
14 utilisons CPN tools [?] pour manipuler les réseaux de Petri colorés et le fichier que CPN Tools
15 accepte est sous format XMI. Avec Acceleo nous engendrons donc un fichier XMI ayant le même
16 format qu'un fichier de CPN Tools et contenant les informations dont nous avons besoin. Cette
17 implémentation a été présentée dans le travail de [?]. La [Figure 7.1](#) montre un exemple de code
18 Acceleo. La partie gauche de la figure représente un mélange entre du code Acceleo et du texte
19 alors que la partie droite représente l'implémentation d'une fonction avec du code Acceleo : les
20 lignes de code 1, 2 et 3 de la figure à gauche représentent le code Acceleo utilisé dans l'outil
21 (le code sert pour l'implémentation de l'algorithme) et que les lignes 4-8 de la figure à gauche
22 représentent du texte au format des fichiers utilisés dans CPN Tools.

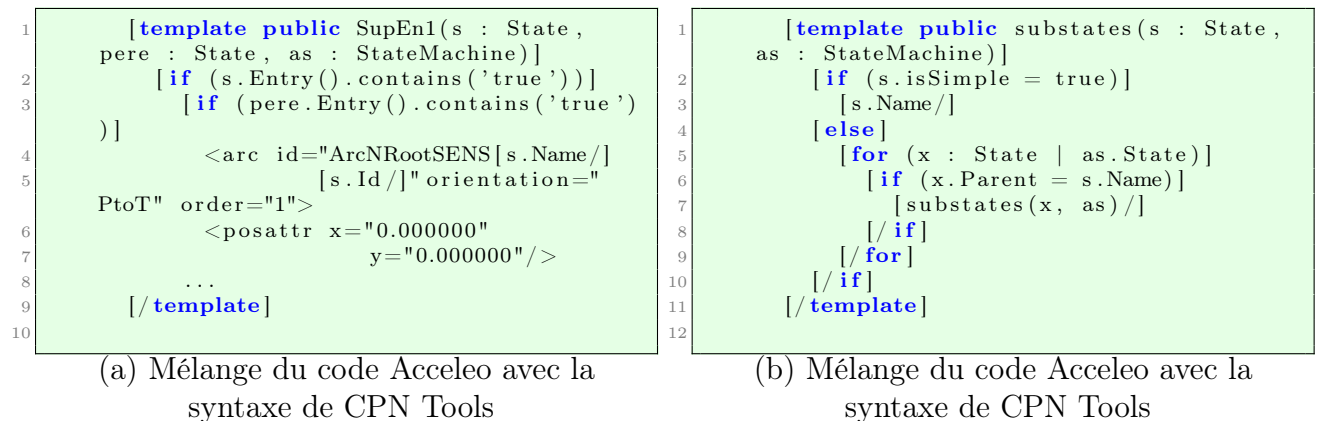


FIGURE 7.1 – Exemple du code Acceleo

23 Cependant il y a plusieurs inconvénients à cette approche. D'une part, Acceleo ne permet
24 pas de manipuler les variables, les fonctions et les structures de données. D'autre part, la géné-
25 ration du code dans Acceleo est basée sur la syntaxe de l'outil du formalisme de sortie ce qui
26 implique qu'il n'est pas possible d'adapter l'implémentation dans le cas du changement d'outil
27 du formalisme de sortie (c'est un inconvénient des outils model2text en général).

28 Au vu des inconvénients de l'utilisation d'Acceleo mais aussi de l'impossibilité d'utiliser un
29 outil de transformation de modèles, nous travaillons sur un outil dont nous implémentons les
30 différentes parties (telles que l'analyseur syntaxique, la fonction de transformation, etc). Cela

1 nous permettra non seulement de l'adapter en cas de changement d'outils (pour le formalisme
2 de sortie ou d'entrée) mais aussi en cas de modification des algorithmes.

3 Implémentation d'un nouvel outil : UML2CPN

4 En nous basant sur notre expérience de l'utilisation des techniques modèle vers le texte, nous
5 avons décidé d'implémenter notre traduction ([?]) en définissant notre propre outil UML2CPN.
6 Nous présentons dans cette section les différentes parties de cet outil.

7 Notons qu'UML2CPN est en cours de développement et qu'uniquement une partie de l'Algorithme 1
8 (présenté en Section 5.5) est considérée. La partie implémentée est l'étape 1 (en ligne 1) et une
9 partie de l'étape 2 (en ligne 5) de l'Algorithme 1. L'étape 1 permet la génération des places de
10 tous les états ainsi que les transitions de leurs comportements Do. L'étape 2 permet la génération
11 de toutes les transitions dans le diagramme états-transitions ceux avec ou sans événements. La
12 partie que nous implémentons de cette étape (c'est-à-dire l'étape 2) est celle des transitions sans
13 événements. Le développement final de notre outil implémentera l'intégralité des Algorithmes 1
14 et 2.

15 Nous utilisons pour le développement d'UML2CPN le langage de programmation *Java*, l'en-
16 vironnement de développement *Netbeans* ainsi que l'analyseur syntaxique *JDOM 2.0.5*². Il est
17 possible d'utiliser d'autres analyseurs syntaxiques pour compléter le développement de ce dernier.
18 UML2CPN prend en entrée un fichier *XMI* qui représente le diagramme états-transitions. Ce
19 fichier est édité en utilisant le plugin *Papyrus*³ dans *Eclipse*⁴. Une version de notre outil est en
20 cours de développement afin de prendre en compte les fichiers engendrés par *Visual paradigm*⁵.

21 Afin d'expliquer au mieux UML2CPN, nous présentons dans ce qui suit les différents éléments
22 qui le composent :

- 23 • Bibliothèques
- 24 • Packages et classes
- 25 • Interfaces

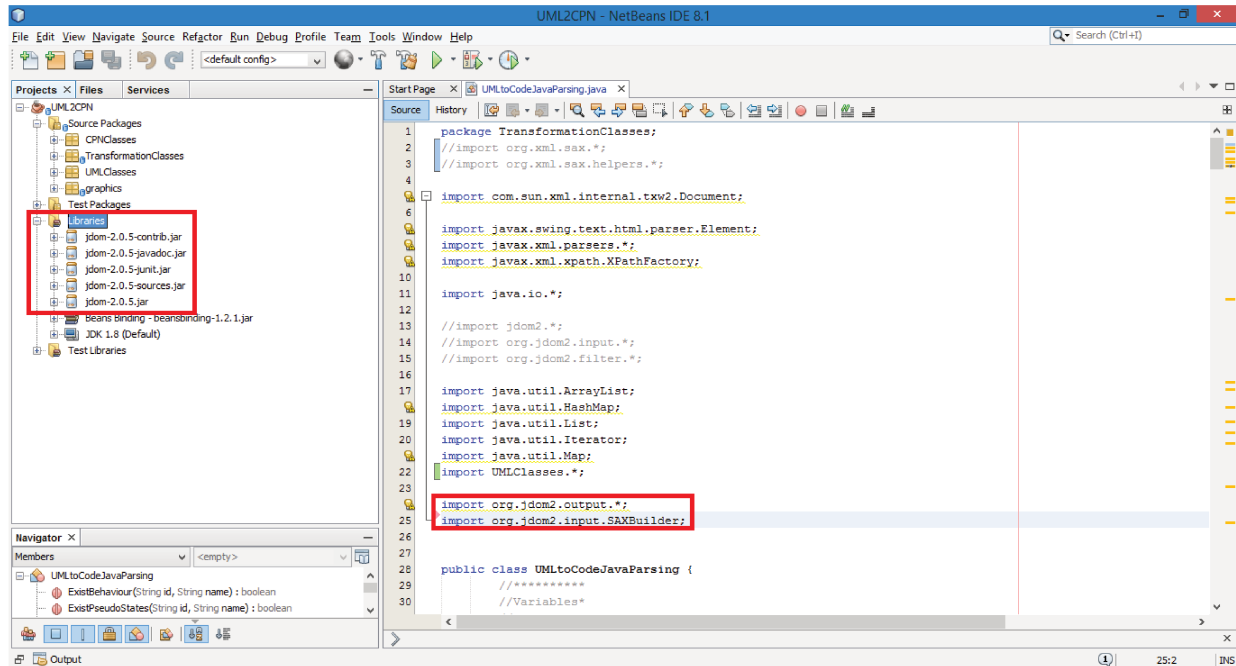
26 Bibliothèques

27 Comme indiqué ci-dessus, nous utilisons *JDOM* comme analyseur syntaxique et cela pour
28 parcourir, manipuler et gérer le fichier *XMI* que l'outil UML2CPN prend en entrée. Il est possible
29 d'utiliser *SAX*⁶ comme analyseur syntaxique, cependant *JDOM* est plus simple et plus facile à
30 utiliser. Par contre, il est nécessaire de noter que lors de la manipulation du fichier *XMI* nous
31 utilisons *SAXBuilder* qui est inclus dans les fichiers importés de *JDOM*. Afin de pouvoir accéder
32 à toutes les fonctionnalités de ce plugin il est nécessaire d'une part d'importer les fichiers dans le
33 code Java (les lignes de code 24-25 encadrées avec du rouge dans la partie droite de la Figure 7.2)
34 mais également d'ajouter à la bibliothèque du plugin *JDOM* dans le projet (les lignes encadrées
35 avec du rouge dans la partie gauche de la Figure 7.2).

36 Packages et classes

37 Nous avons développé pour UML2CPN 4 packages et 15 classes (voir la Figure 7.3) :

-
2. <http://www.jdom.org/>
 3. <http://www.eclipse.org/papyrus/>
 4. <https://www.eclipse.org/>
 5. <https://www.visual-paradigm.com/>
 6. <https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html>

FIGURE 7.2 – Fichiers *JDOM*

- 1 • Le package *CPNClasses* qui comporte les classes *cpnArc*, *cpnPlace* et *cpnTransition*. Ces
- 2 classes permettent de stocker les données des places, transitions et arcs résultats de l'ap-
- 3 plication des algorithmes de traductions.
- 4 • Le package *TransformationClasses* qui comporte les classes *Function*, *UMLtoCPNTrans-*
- 5 *lation* et *UMLtoCodeJavaParsing*. La première classe comporte l'ensemble des fonctions
- 6 utilisées dans l'outil. La seconde classe comporte le code qui implémente nos algorithmes.
- 7 Enfin, la troisième classe comporte le code qui permet le passage d'un fichier *XMI* vers du
- 8 code Java.
- 9 • Le package *UMLClasses* qui comporte les classes *Behaviour*, *ForkJoin*, *History*, *Pseudo-*
- 10 *State*, *Region*, *SpecialTransition*, *State* et *Transition*. Ces classes contiennent les informa-
- 11 tions sur tous les éléments syntaxiques du modèle récupéré en entrée dans notre outil.
- 12 Par exemple, la classe *Behaviour* contient toutes les informations concernant les compor-
- 13 tements d'entrée, de sortie et do pour chaque état dans le diagramme états-transitions.
- 14 • Le package *graphics* qui comporte les classes *ChooserFile*, *Interface* et *UML2CPNInterface*.
- 15 Les deux premières classes sont utilisées uniquement pour des tests, par contre la troisième
- 16 classe permet la définition de l'interface de l'outil.

17 Interface et utilisation de l'outil UML2CPN

18 Nous présentons dans cette section l'interface de notre outil ainsi que comment l'utiliser.

19 La Figure 7.4 montre l'interface de notre outil UML2CPN. Cette interface comporte des

20 affichages et un ensemble de boutons :

- 21 • Le bouton *Choose the model* qui permet de choisir le fichier *XMI* à utiliser (ce fichier
- 22 contient le diagramme états-transitions à traduire).

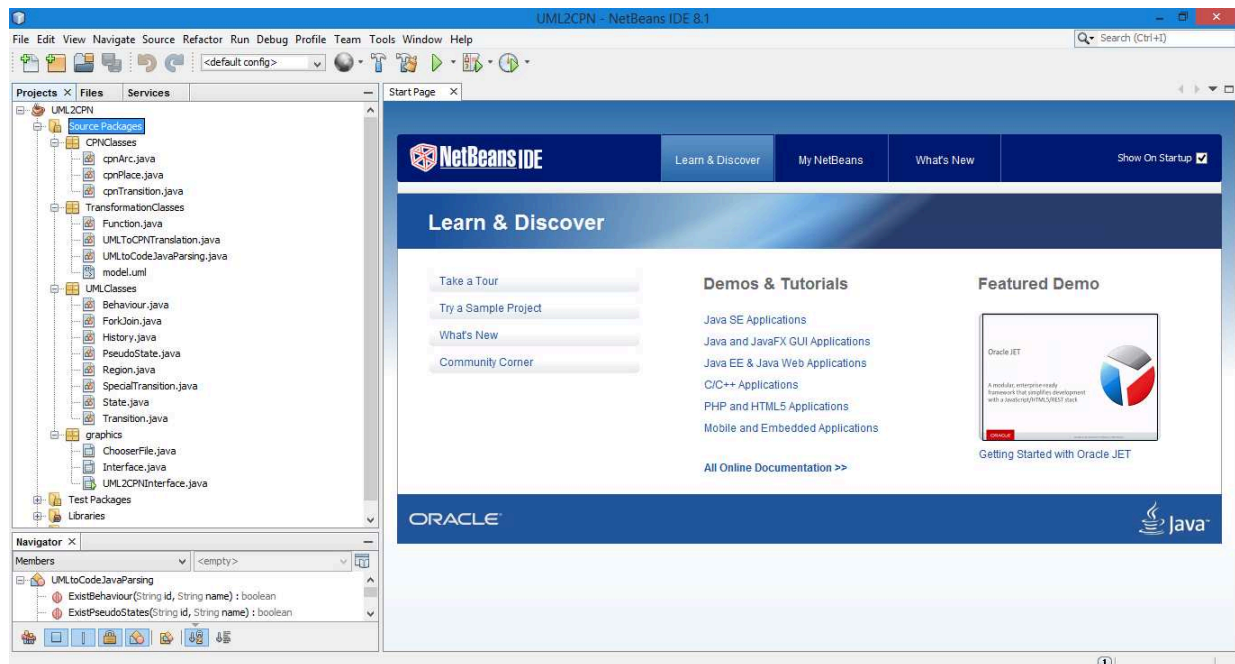


FIGURE 7.3 – Packages et classes utilisés dans UML2CPN

- 1 • Le bouton *Parsing the SMD* qui permet de faire le passage de la représentation graphique
- 2 (en utilisant le fichier *XMI*) des diagrammes états-transitions vers la représentation cor-
- 3 respondante en Java (c'est-à-dire les données que le diagramme représente seront stockées
- 4 dans des structures de données).
- 5 • Le bouton *Translation to CPN* qui permet de traduire le diagramme états-transitions vers
- 6 le réseau de Petri coloré correspondant en utilisant nos algorithmes.
- 7 • Le bouton *Generation of CPNTools file* qui permet d'engendrer le fichier contenant le
- 8 résultat de la traduction et qui est adapté à ce que l'outil CPN Tools prend en compte
- 9 comme fichier.
- 10 • L'affichage *SMD states* contient tous les états du diagramme états-transitions récupérés à
- 11 partir du fichier *XMI* en entrée.
- 12 • L'affichage *SMD transitions* contient toutes les transitions du diagramme états-transitions
- 13 source.
- 14 • L'affichage *Fork and Join Transition* contient toutes les transitions Fork et Join du dia-
- 15 gramme états-transitions source.
- 16 • L'affichage *CPN places* contient toutes les places du réseau de Petri coloré après la tra-
- 17 duction.
- 18 • L'affichage *CPN transitions* contient toutes les transitions du réseau de Petri coloré résul-
- 19 tat.
- 20 • L'affichage *CPN arcs* contient tous les arcs du réseau de Petri coloré résultat.

21 Nous avons défini une interface simple pour notre outil afin de faciliter l'utilisation de ce
 22 dernier par les utilisateurs. L'utilisateur commence par récupérer son fichier *XMI* en cliquant

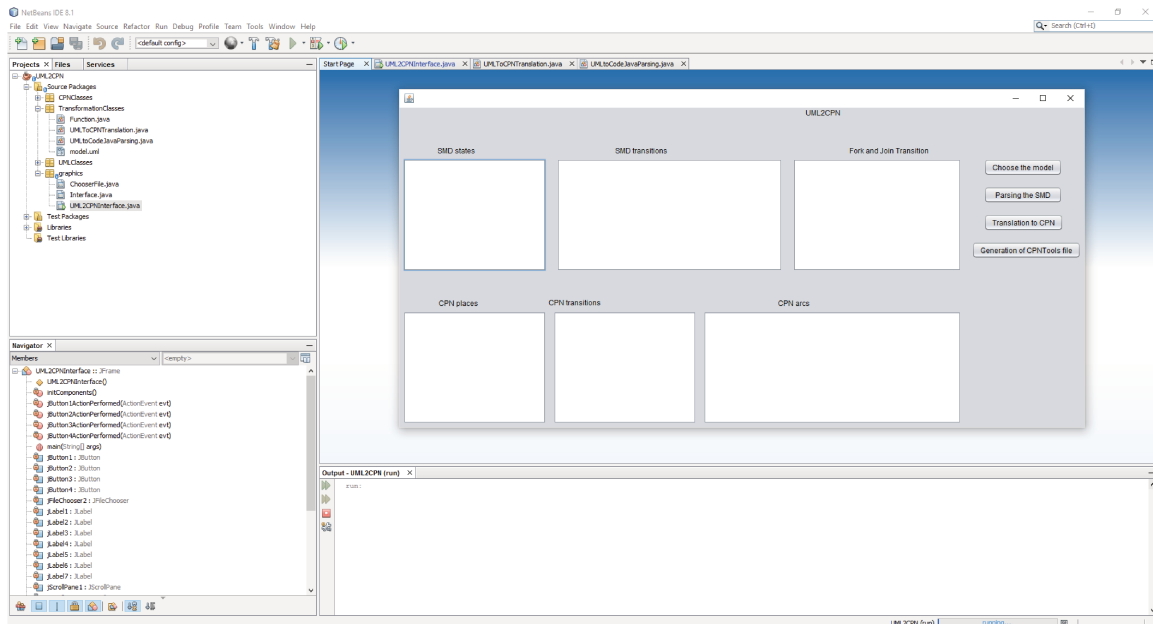


FIGURE 7.4 – L’interface de l’outil UML2CPN

1 sur le bouton *Choose the model*. Dans les fichiers engendrés par *Papyrus* il y a le fichier *.di* qu’il
 2 faut charger comme montré dans la [Figure 7.5](#) (le fichier est *model.di* dans la figure).

3 Une fois que l’utilisateur choisit le fichier *XMI*, il peut récupérer les différentes données de son
 4 diagramme (tels que les noms des états, les comportements, les transitions etc) afin des les stocker
 5 et les utiliser après en cliquant sur le bouton *Parsing of the SMD*. Une fois le code engendré,
 6 l’outil affichera tous les états, les transitions et les transitions Fork et Join que le diagramme
 7 états-transitions (récupéré en entrée) comporte. La [Figure 7.6](#) montre l’état de l’outil après
 8 l’utilisation du bouton *Parsing of the SMD*.

9 Dans l’étape suivante, l’utilisateur peut appliquer nos algorithmes afin d’engendrer le réseau
 10 de Petri coloré (correspondant au diagramme états-transitions source). Afin de faire la traduction,
 11 l’utilisateur doit cliquer sur le bouton *Translation to CPN* qui permet d’afficher les places,
 12 transitions et arcs du réseau de Petri coloré résultat (voir les zones d’affichagees *CPN places*, *CPN*
 13 *transitions* et *CPN arcs* dans l’outil). Rappelons que notre outil est en cours de développement
 14 et actuellement nous n’avons implémenté qu’une partie de nos algorithmes. La [Figure 7.7](#) montre
 15 l’application de la traduction sur le diagramme états-transitions source et le résultat (les réseaux
 16 de Petri colorés ne sont pas visibles graphiquement car c’est un module que nous n’avons pas
 17 encore implémenté) .

18 Enfin, l’utilisateur peut engendrer (en cliquant sur le bouton *Generation of CPNTools file*
 19 voir la [Figure 7.8](#) pour le bouton) le fichier compatible avec CPNTools et qui contiendra le
 20 réseau de Petri coloré (résultat de la traduction). Ce fichier sera directement utilisable avec
 21 l’outil CPNTools. Rappelons que cette fonctionnalité n’est pas encore implémentée dans notre
 22 outil et par conséquent le réseau de Petri coloré ne sera pas affiché.

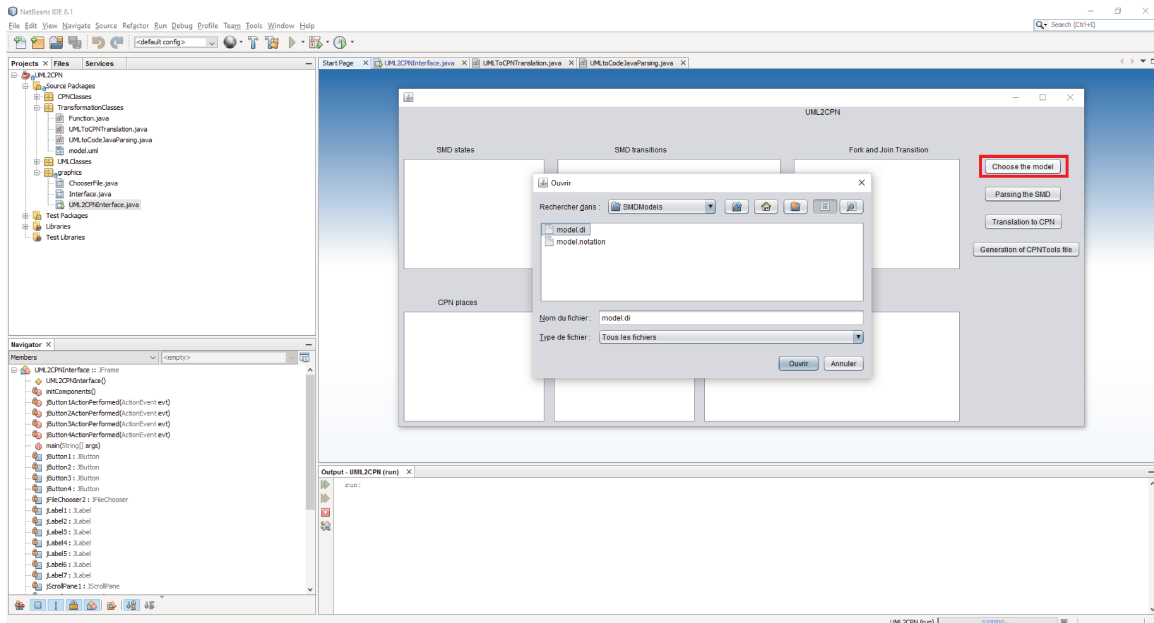


FIGURE 7.5 – La récupération du fichier XMI dans UML2CPN

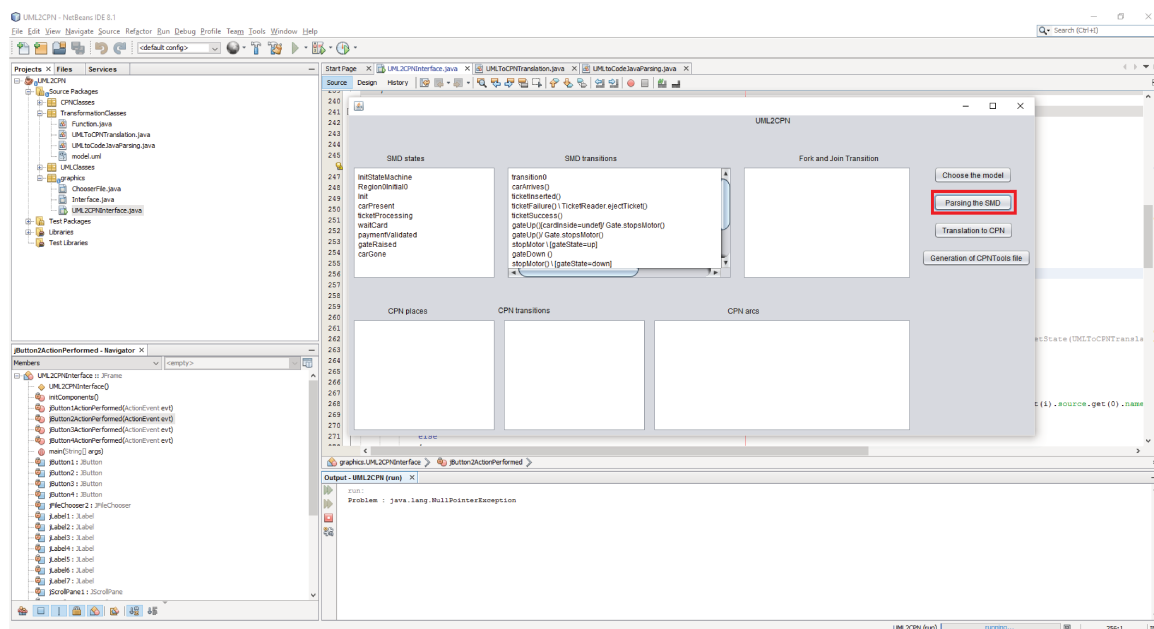


FIGURE 7.6 – La récupération des données du SMD à partir du fichier XMI dans UML2CPN

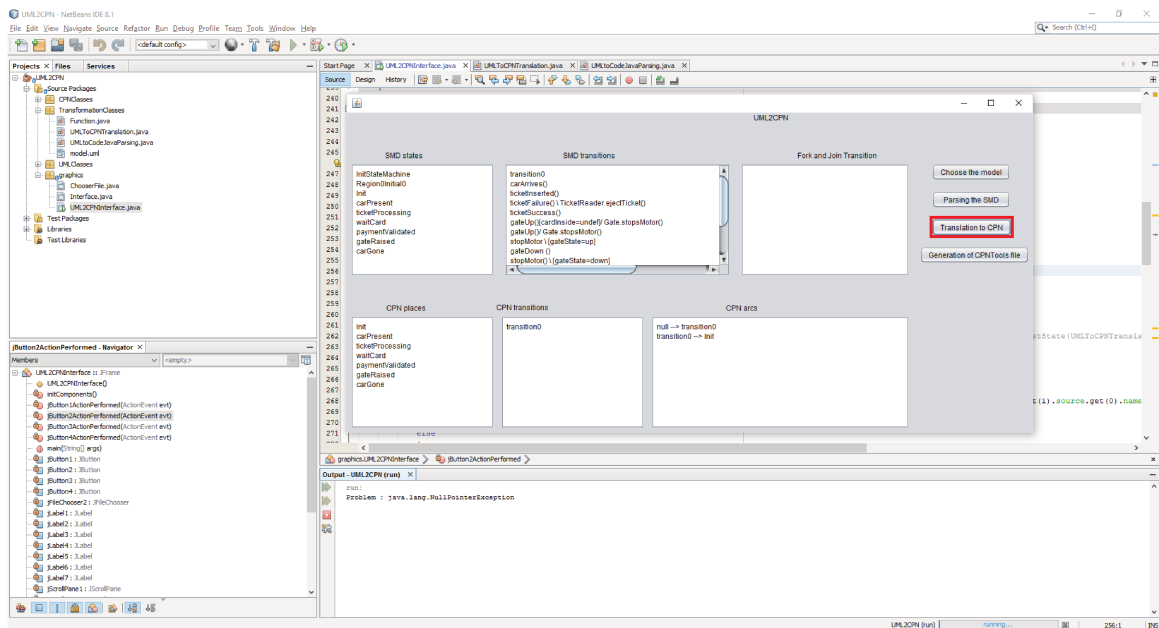


FIGURE 7.7 – La traduction du diagramme états-transitions vers le réseaux de Petri coloré correspondant dans UML2CPN (version non graphique)

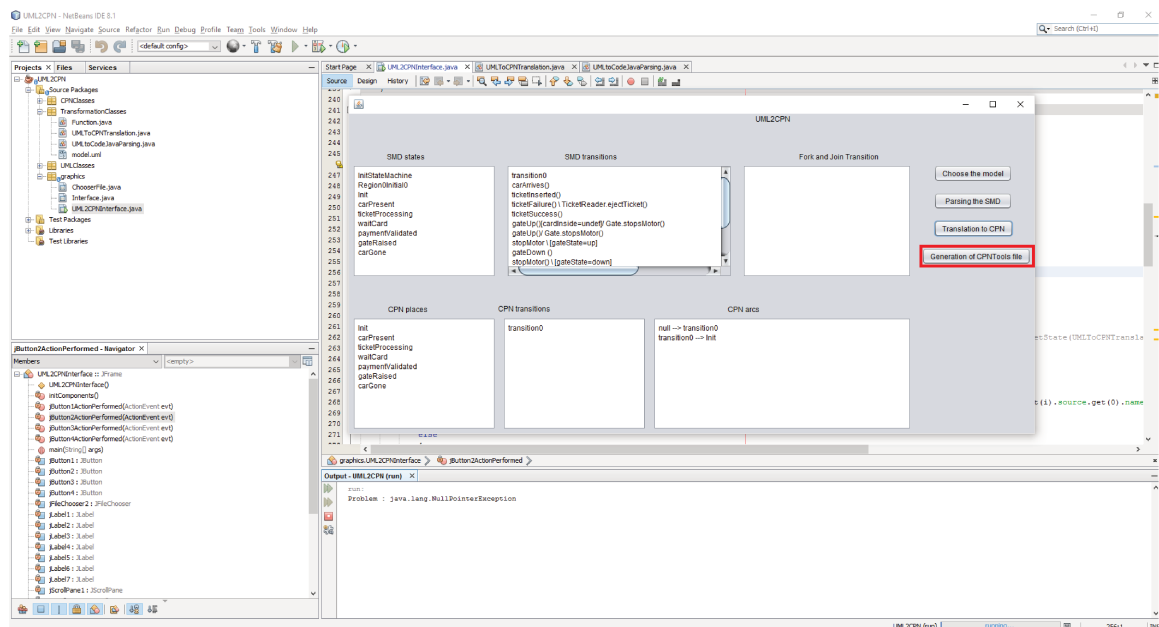


FIGURE 7.8 – La génération du fichier CPN Tools dans UML2CPN (module en cours de développement)

1 Conclusion

2 Notre outil UML2CPN est en cours de développement et nous souhaitons le finaliser afin
3 d'avoir une traduction automatique des diagrammes états-transitions vers les réseaux de Petri
4 colorés. La version actuelle d'UML2CPN implémente une partie de l'[Algorithme 1](#) de notre
5 traduction. Afin de compléter l'outil et donc implémenter entièrement notre traduction, nous
6 souhaitons terminer l'implémentation du premier algorithme et écrire l'implémentation du second
7 algorithme. Enfin, nous avons l'intention de créer un outil qui combine à la fois l'outil *EasySM*
8 (outil d'aide à la modélisation avec les diagrammes états-transitions présenté en [Section 4.9](#)) ainsi
9 que l'outil UML2CPN (l'outil de traduction des diagrammes états-transitions vers les réseaux de
10 Petri colorés). Cela nous permettra d'avoir un outil complet que l'utilisateur peut utiliser pour
11 faire de la modélisation et de la vérification.
12 Notons que notre outil UML2CPN sera également mis à jour une fois que nous aurons intégré
13 les expressions temporelles dans notre traduction.

Chapitre 8

Expression temporelle

Contents

8.1	Introduction	175
8.2	Choix des contraintes temporelles	176
8.3	Annotations temporelles	176
8.4	Unités de temps	183
8.5	Comparaisons	184
8.6	Exemples : prise en compte des contraintes temporelles	186
8.7	Le temps dans les réseaux de Petri colorés	191
8.8	Introduction des contraintes temporelles dans la traduction	194
8.9	Conclusion	200

Introduction

Plusieurs travaux existent concernant les contraintes temporelles et leurs utilisation dans les diagrammes états-transitions (par exemple, MARTE). Ces travaux offrent une grande expressivité et une syntaxe très riche pour la représentation des contraintes temporelles afin de considérer le plus large ensemble des systèmes temporisés. Cependant, l'utilisation de ces travaux peut être difficile pour les ingénieurs non-experts n'ayant pas de connaissances poussées en terme de contraintes temporelles ; car, souvent, avoir une syntaxe riche et une représentation graphique complète signifie une présence d'une sémantique correspondante complexe.

Nous présentons dans ce chapitre une annotation (syntaxe et sémantique) pour les diagrammes états-transitions avec des contraintes temporelles. Nous utiliserons par la suite ces annotations dans nos approches afin prendre en compte les systèmes avec du temps.

Ce chapitre est présenté comme suit : une partie est dédiée à la présentation de notre syntaxe ainsi que sa sémantique afin d'annoter les diagrammes états-transitions (Sections 8.2 à 8.6), une partie concernant les réseaux de Petri colorés temporisés (Section 8.7), ensuite l'utilisation de cette syntaxe dans nos approches (Section 8.8).

Choix des contraintes temporelles

Le but principal de ce chapitre est de proposer des annotations des contraintes temporelles pour les diagrammes états-transitions. Ces annotations ont une syntaxe simple pour la facilité d'utilisation et de compréhension pour les ingénieurs non-experts. En effet, nous souhaitons proposer d'une part une syntaxe qui prend en compte des contraintes temporelles parmi les plus utilisées pour la représentation des systèmes temporisés. D'autre part, que l'utilisateur puisse utiliser cette syntaxe sans difficulté.

Une originalité de notre syntaxe est que nous représentons les contraintes temporelles dans les événements mais également dans les états.

- Dans les événements : étant donné que le but principal des diagrammes états-transitions est de modéliser les comportements d'un système et son changement d'un état à un autre ; nous considérons comme primordiale la présence des contraintes temporelles par rapport aux événements. En effet, le système change d'un état en fonction de l'occurrence des événements, donc la présence de contraintes pour conditionner avec des contraintes temporelles l'occurrence de ces événements est nécessaire.
- Dans les états : un état dans les diagrammes états-transitions peut avoir un comportement qui modélise les actions à faire dans le cas où le système est dans cet état-là. Dans certains cas, la présence d'une contrainte temporelle pour avoir une condition sur le temps d'exécution de ce comportement est justifiée. En effet, la contrainte temporelle ne concerne pas toujours l'occurrence des événements mais peut concerner également l'exécution des comportements et, par conséquent, les états.

Nous considérons dans notre syntaxe un ensemble de contraintes temporelles qui est le suivant : la ponctualité, le délai (minimum et maximum), la latence, la séquence et la précédence d'événements. Les expressions qui contiennent des contraintes temporelles peuvent concerner l'occurrence des événements (ou l'exécution des comportements) par rapport à un temps donné (nous appelons cela expressions temporelles) ou bien par rapport à d'autres événements (nous appelons cela des expressions logiques).

Une autre originalité de notre travail est la combinaison entre les expressions temporelles et les expressions logiques. Cette combinaison nous permet d'élargir l'expressivité de notre syntaxe. Nous pensons que nos annotations fournissent un bon compromis entre simplicité d'utilisation et expressivité ; car nous ne souhaitons pas avoir une expressivité maximale.

Annotations temporelles

Classification

Nous proposons dans ce qui suit notre classification des contraintes temporelles. La classification est basée sur deux notions : les contraintes quantitatives (telles que l'apparition d'un événement à un temps précis) et les contraintes qualitatives (telles que l'apparition d'un événement avant un autre). Cette classification montre les différentes propriétés que nous prenons en compte concernant les contraintes temporelles, à savoir : le délai maximum et minimum, la ponctualité, la latence, la séquence d'événements et la précédence d'événements. La classification prend aussi en compte l'utilisation d'une contrainte temporelle dans les états portant sur le temps d'exécution du comportement *do*. Notons que pour la partie syntaxique de la classification, nous sommes inspiré des travaux présentés dans la [Section 3.3.4](#).

1 La Figure 8.1 montre le diagramme de classes de la classification des contraintes temporelles
 2 avec les différents besoins temporels qui sont considérés dans cette thèse.

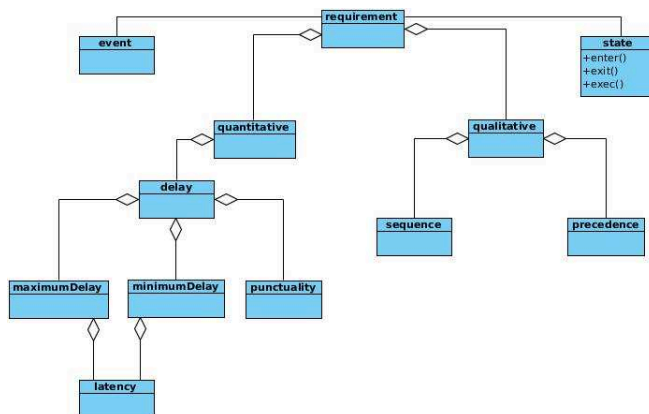


FIGURE 8.1 – Nouvelle classification des besoins temporels

3 Comme la Figure 8.1 le montre nous considérons les contraintes temporelles vis-à-vis des
 4 événements mais également dans les états. La classification (présente dans la Figure 8.1) comporte
 5 deux types de contraintes : contraintes quantitatives (contraintes temporelles) et contraintes
 6 qualitatives (contraintes logiques). La première partie permet de représenter le délai (minimum
 7 et maximum), la ponctualité ainsi que la latence. La seconde partie permet de représenter la
 8 séquence et la précedence entre événements. La représentation des contraintes temporelles vis-
 9 à-vis les événements est faite en utilisant les deux parties. En revanche, la représentation des
 10 contraintes temporelles dans les états est faite en utilisant uniquement la première partie (c'est-
 11 à-dire les contraintes quantitatives).

12 Nous présentons dans ce qui suit la syntaxe et la sémantique basée sur cette classification.
 13 Nous présentons également la combinaison des contraintes qualitatives et des contraintes quan-
 14 titatives.

15 Syntaxe et sémantique des annotations temporelles

16 Nous présentons dans cette sous-section la syntaxe/sémantique correspondantes à la classi-
 17 fication que nous proposons dans le cadre de cette thèse. Nous illustrons cette syntaxe par une
 18 grammaire (voir la Figure 8.4) avec sa description. Nous accompagnons cette grammaire par le
 19 Tableau 8.1 qui comporte les différents terminaux utilisés.

20 Nous avons besoins de définir certains concepts que nous utiliserons pour la description de
 21 notre syntaxe comme suit. Soit **S1** et **S2** deux états et soit **Transit** une transition entre **S1** et
 22 **S2**. L'état **S1** comporte un comportement *do* noté **doexp**. *E* est l'événement qui étiquette la
 23 transition **Transit**. Ces éléments seront utilisés pour l'explication des différentes annotations du
 24 Tableau 8.1. Deux points à noter :

- 25 • Les temps utilisés dans notre syntaxe (par exemple, *E at T*) sont des temps relatifs et les
 26 intervalles pour l'évaluation des contraintes sont fermés.
- 27 • L'événement dans une transition est enrichi par une contrainte temporelle, donc c'est
 28 toujours l'événement tout à gauche de la contrainte qui est celui de la transition.

29 Notre choix de la classification proposée dans la Figure 8.1 est basé sur la gestion des
 30 contraintes temporelles portant sur les événements (mais également sur les états voir Section 8.3.3).

1 Chaque événement peut apparaître au plus T unités de temps (**at_most**), au minimum T d'unités
 2 de temps (**at_least**), exactement T unités de temps (**at**), ou bien en séquence/précédence avec
 3 d'autres événements (**after**, **before**). Il faut noter que la contrainte temporelle est vérifiée pour
 4 un événement donné dans le cas où l'état source de la transition étiquetée par l'événement (sous
 5 cette contrainte) est actif (c'est-à-dire l'état dans lequel se trouve le système). Donc le temps est
 6 mesuré à partir du moment où l'état source de la transition est actif. Dans le cas où la transition
 7 a comme source un pseudo-état join alors il faut que tous les états sources des transitions allant
 8 vers le pseudo-état join soient actifs afin que la contrainte de la transition puisse être évaluée.
 9 En se basant sur cette classification, nous définissons les annotations suivantes (Tableau 8.1).

Éléments	Annotation	Contrainte
punctuality	at	Quantitative
maximumDelay	at_most	Quantitative
minimumDelay	at_least	Quantitative
latency	between	Quantitative
sequence	after	Qualitative
precedence	before	Qualitative

TABLE 8.1 – Syntaxes des terminaux utilisés dans la grammaire de la Figure 8.4

10 Afin d'exprimer la sémantique informelle de chaque annotation syntaxique présentée dans le
 11 Tableau 8.1, nous présentons dans ce qui suit la description annotation :

- 12 • Dans le cas de **at** : $E \text{ at } T$, c'est-à-dire l'événement E apparaît exactement T unités
 13 de temps après l'activation de l'état source de cette transition étiquetée par l'événement
 14 E . Dans le cas où l'événement E n'apparaît pas exactement T unités de temps après
 15 l'activation de l'état S_1 , la contrainte ne sera pas satisfaite. Nous avons également *doexp*
 16 **at** T , c'est-à-dire l'exécution du comportement *do* doit se terminer exactement T unité de
 17 temps après l'activation de l'état concerné par le comportement *do*.
- 18 • Dans le cas de **at_most** : $E \text{ at_most } T_{max}$, c'est-à-dire l'événement E apparaît au plus
 19 T_{max} unités de temps après l'activation de l'état source de cette transition. Dans le cas
 20 du comportement *do*, nous avons *doexp at_most* T_{max} , c'est-à-dire l'exécution du com-
 21 portement *do* doit se terminée au plus T_{max} unités de temps après l'activation de l'état
 22 concerné par le comportement *do*.
- 23 • Dans le cas de **at_least** : $E \text{ at_least } T_{min}$, c'est-à-dire l'événement E apparaît au
 24 minimum T_{min} unités de temps après l'activation de l'état source de cette transition. Nous
 25 avons également *doexp at_most* T_{min} , c'est-à-dire l'exécution du comportement *do* doit
 26 se terminer au minimum T_{min} unités de temps après l'activation de l'état concerné par le
 27 comportement *do*.
- 28 • Dans le cas de **between** : $E \text{ between } [T_{min}, T_{max}]$, c'est-à-dire l'événement E apparaît au
 29 minimum T_{min} unités de temps et au plus T_{max} unités de temps après l'activation de l'état
 30 source de cette transition. Dans le cas du comportement *do*, nous avons *doexp between*
 31 $[T_{min}, T_{max}]$, c'est-à-dire l'exécution du comportement *do* doit se terminer au minimum
 32 T_{min} unités de temps et au plus T_{max} unités de temps après l'activation de l'état concerné
 33 par le comportement *do*.
- 34 • Dans le cas de **after** : $E_2 \text{ after } E_1$ (où E_2 est l'événement de la transition enrichi par la
 35 contrainte), c'est-à-dire l'événement E_2 apparaît après l'événement E_1 . La sémantique que

1 nous considérons par rapport au **after** est que pour chaque occurrence de l'événement E_1
 2 on peut avoir au plus une occurrence de l'événement E_2 .

3 Afin d'expliquer la sémantique que nous considérons pour l'élément syntaxique **after** (le
 4 même principe sera pour l'élément syntaxique **before**), nous présentons l'exemple de la
 5 [Figure 8.2](#). Dans cet exemple, l'occurrence de l'événement e_1 est conditionnée par l'occur-
 6 rence de l'événement e_2 . Nous présentons les séquences de franchissement de transitions
 7 possibles et impossibles que nous autorisons ou interdisons.

- 8 – La séquence d'occurrence d'événements $e_1 e_2 e_1 e_2 b$ est autorisée car pour chaque oc-
 9 currence de l'événement e_1 il est possible de franchir une transition ayant la contrainte
 10 $e_2 \text{ after } e_1$ (autrement dit avoir une occurrence de l'événement e_2). La séquence
 11 d'occurrence d'événements suivante $e_1 e_2 e_1 e_1 e_2 b$ est également autorisée car on
 12 considère la dernière occurrence de l'événement e_1 .
- 13 – La séquence d'occurrence d'événements $e_1 e_2 e_2 b$ n'est pas autorisée car la première
 14 occurrence e_1 permet l'occurrence e_2 , par contre la seconde occurrence e_2 n'est pas
 15 autorisée car il n'y a eu aucune occurrence de l'événement e_1 .

16 Un cas particulier que nous avons remarqué dans cet exemple est la présence d'une situation
 17 d'inter-blocage dans le cas où l'événement b n'apparaît pas, que l'événement e_1 n'apparaît
 18 qu'une fois. Cette situation implique que la transition entre l'état **S12** et l'état **S13** peut
 19 être franchie par contre le reste des transitions non.

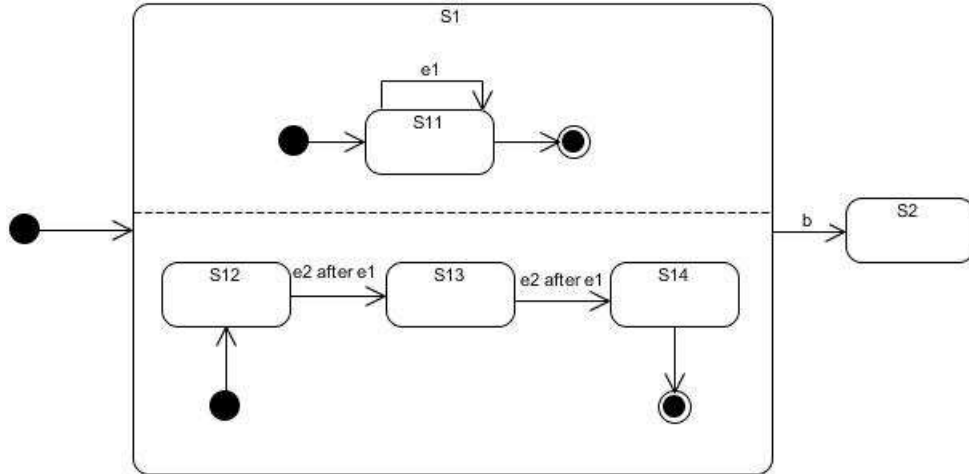
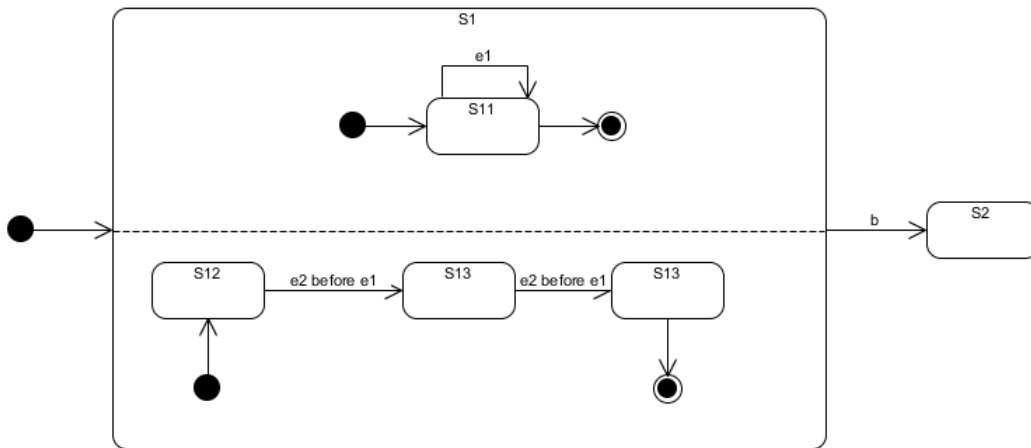
- 20 • Dans le cas de **before** : $E_1 \text{ before } E_2$ (où E_1 est l'événement de la transition enrichi par la
 21 contrainte), c'est-à-dire si l'événement E_2 apparaît, alors il y a l'occurrence de l'événement
 22 E_1 avant au moins une fois. Nous illustrons la sémantique du **before** avec un exemple
 23 et nous présentons les séquences de transitions autorisées. Dans la [Figure 8.3](#) un exemple
 24 diagramme états-transitions utilisant l'élément syntaxique **before**. Dans ce qui suit les
 25 séquences de transitions autorisées et non autorisées avec un cas d'inter-blocage.

- 26 – La séquence d'occurrence d'événements e_2, e_1, e_2, e_1, b est autorisée car pour chaque
 27 occurrence de l'événement e_1 il y a eu une occurrence de l'événement e_2 .
- 28 – La séquence d'occurrence d'événements e_2, e_1, e_1, e_1, b n'est pas autorisée car afin
 29 que la deuxième occurrence de e_1 soit possible il est nécessaire que e_2 soit apparu
 30 avant au moins une fois.
- 31 – Dans le cas où l'événement b n'apparaît pas et dans le cas où on a la séquence de
 32 transitions e_2, e_2, e_1 , alors on ne peut plus franchir de transitions car l'événement
 33 e_1 ne peut apparaître que si l'événement e_2 apparaît juste avant.

34 Il est nécessaire que cette contrainte soit évaluée au moment où l'état **S1** est actif.

35 **Remarques** Une remarque à prendre en compte est la non présence des contraintes tempo-
 36 relles exprimées avec les éléments syntaxiques **after** et **before** dans les états. Nous considérons
 37 que l'exécution du comportement \mathcal{D} ne dépend pas de l'occurrence des événements mais dépend
 38 de son temps d'exécution.

39 Nous présentons dans ce qui suit la grammaire représentant l'expressivité de la syntaxe que
 40 nous proposons au-dessus (voir [Figure 8.4](#)).

FIGURE 8.2 – Exemple d’explication pour l’élément syntaxique **after**FIGURE 8.3 – Exemple d’explication pour l’élément syntaxique **before**

1 Description de la grammaire

2 Nous décrivons dans ce qui suit les différentes clauses de la grammaire proposée dans [Figure 8.4](#).
 3 Nous supposons qu’il y a contrainte pour préciser que la clause *EventExpo* est destinée pour les
 4 expressions temporelles concernant les événements et que la clause *StateExp* est destinée pour
 5 les expressions temporelles concernant les états.

- 6 • *val* est une variable avec une valeur rationnelle représentant une valeur temporelle.
- 7 • *e*, *e1* et *e2* sont des événements.
- 8 • *TimeExp* ::= *EventExp* / *StateExp*, une expression temporelle (*TimeExp*) qui peut être :
 9 (i) *EventExp* une expression temporelle par rapport aux événements (ii) *StateExp* une
 10 expression temporelle par rapport aux comportements de des états.
- 11 • *EventExp* ::= *e TimedAnnotation* / *e LogicalAnnotation* / *e CombinationAnnotation*, une
 12 expression temporelle par rapport aux événements qui peut être : (i) *TimedAnnotation* une
 13 annotation temporelle qui utilise des contraintes quantitatives (c’est-à-dire **at**, **at_least**,
 14 etc) (ii) *LogicalAnnotation* une annotation logique qui utilise des contraintes qualitatives

```

1 Grammaire
2
3 TimeExp ::= EventExp | StateExp
4
5 EventExp ::= e TimedAnnotation | e LogicalAnnotation | e CombinationAnnotation
6
7 StateExp ::= do doExpression TimedAnnotation
8
9 TimedAnnotation ::= at val | at_least val | at_most val | between [val1, val2]
10
11 LogicalAnnotation ::= after e1 | before e1
12
13 CombinationAnnotation ::= TimedAnnotation LogicalAnnotation

```

FIGURE 8.4 – Grammaire de la classification temporelle proposée dans Figure 8.1

- (c'est-à-dire **before** et **after**) (iii) *CombinationAnnotation* une combinaison entre annotations temporelle et logique.
- *StateExp* ::= *do doExpression TimedAnnotation* une expression temporelle par rapport aux états et qui concerne les comportements *do*. Cette expression ne peut comporter que des annotations temporelles (*TimedAnnotation*).
 - *TimedAnnotation* ::= *at val* / *at_least val* / *at_most val* / *between [val1, val2]* éléments syntaxiques utilisés dans les annotations temporelles afin de représenter des contraintes quantitatives.
 - *LogicalAnnotation* ::= *after e1* / *before e1* éléments syntaxiques utilisés dans les annotations logiques afin de représenter des contraintes qualitatives.
 - *CombinationAnnotation* ::= *TimedAnnotation LogicalAnnotation* : combinaison des éléments syntaxiques des annotations temporelles et des annotations logiques afin d'avoir des expressions qui utilisent les deux expressions (c'est-à-dire logiques et temporelles).
- Nous illustrons notre grammaire avec un diagramme états-transitions annoté avec des contraintes temporelles en utilisant notre syntaxe (voir la Figure 8.5).

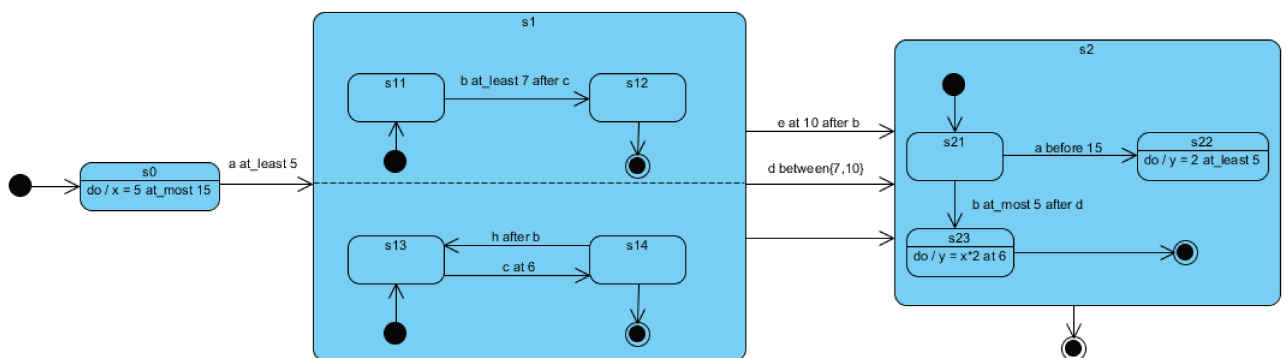


FIGURE 8.5 – Représentation des contraintes temporelles dans les diagrammes états-transitions

- Chaque expression temporelle représentée dans le diagramme états-transitions de la Figure 8.5 est un cas considéré par notre grammaire (présentée dans la Figure 8.4) :

- 1 • **a at_least 5** représente une expression temporelle pour les événements. Cette expres-
2 sion est incluse dans la clause *EventExp := e TimedAnnotation*. Le diagramme états-
3 transitions de la Figure 8.5 comporte d'autres expressions incluses dans la même clause,
4 telles que : **c at 6**, **d between[7, 10]**.
- 5 • **b at_least 7 after c** représente une combinaison d'expression temporelle et logique.
6 Cette expression est incluse dans la clause *EventExp := e CombinationAnnotation*. Le
7 diagramme états-transitions de la Figure 8.5 comporte également d'autres expressions in-
8 cluses dans la même clause, telles que : **e at 10 after b**, **b at_most 5 after d**.
- 9 • **h after b** représente une expression logique. Cette expression est incluse dans la clause
10 *EventExp := e LogicalAnnotation*. Le diagramme états-transitions de la Figure 8.5 com-
11 porte une autre expression incluse dans la même clause : **a before 15**.
- 12 • **x = 5 at_most 15** représente une expression temporelle pour le comportement do de l'état
13 **s0**. Cette expression est incluse dans la clause *StateExp := do doExpression TimedAn-*
14 *notation*. Le diagramme état-transitions de la Figure 8.5 comporte d'autres expressions
15 incluses dans la même clause : **y = x * 2 at 6**, **y = 2 at_least 5**.

16 Gestion de l'échec de la garde temporelle

17 Concernant la prise en compte de l'échec d'une condition temporelle sur l'exécution du com-
18 portement do d'un état, c'est-à-dire comment le système va réagir dans le cas où la condition
19 temporelle n'est pas vérifiée.

20 Une première solution est d'ajouter un état erreur qui modélise l'état du système s'il y a
21 un échec d'une condition temporelle. Dans la phase de vérification du réseau de Petri coloré
22 résultat de la traduction du modèle UML, il suffit de vérifier la présence de jetons dans la place
23 correspondante à cet état erreur. Par exemple, dans la Figure 8.6 si la contrainte temporelle de
24 la transition avec l'événement **ticketInserted** n'est pas respectée alors la transition de l'état
25 **carPresent** vers **Failure** sera franchie. Quand l'échec concerne la condition temporelle d'une
26 transition alors c'est l'état précédant la transition qui aura un lien avec l'état **Failure** (par
27 exemple l'état **carPresent**).

28 Une seconde solution est de ne pas ajouter d'états d'échecs et de ne pas gérer l'échec de la
29 condition temporelle. La détection de l'échec de la condition temporelle sera géré au moment de
30 la vérification (après transformation vers les réseaux de Petri), car ça sera le model checker qui
31 le fera.

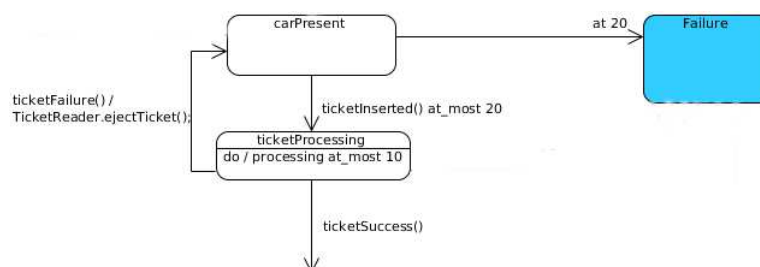


FIGURE 8.6 – Gestion de l'échec des contraintes temporelles

32 Exprimer le temps dans les états ou dans les transitions ?

33 Une question qu'on peut se poser sur l'expression d'une contrainte temporelle, est-ce qu'on
34 doit l'exprimer dans les états ou dans les transitions ?

1 Dans le cas où le temps concerne le passage du système d'un état à un autre et donc par
 2 conséquent l'occurrence de l'événement, alors nous exprimons le temps dans les transitions. Par
 3 exemple, dans la Figure 8.7, la transition étiquetée avec l'événement `carLeaves` comporte une
 4 contrainte temporelle. La présence de cette contrainte est parce quelle concerne l'occurrence de
 5 l'événement `carLeaves` et non pas le comportement `do` de l'état `gateRaised` (qui dans l'exemple
 6 ne possède pas un comportement `do`).

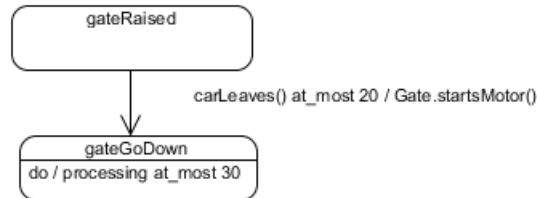


FIGURE 8.7 – Le temps dans les transitions

7 Dans le cas où le temps concerne l'exécution d'un comportement dans un état donné et
 8 non pas l'occurrence d'un événement, alors nous exprimons le temps dans les états en défi-
 9 nissant la contrainte par rapport au comportement `do`. Par exemple dans la Figure 8.8, l'état
 10 `ticketProcessing` comporte une contrainte temporelle concernant son comportement `do` (c'est-
 11 à-dire le comportement `do processing` qui a pour contrainte 30 secondes). La contrainte tempo-
 12 relle est exprimée dans l'état et non pas dans la transition (étiquetée avec l'événement `ticketSuccess`)
 13 car cela concerne l'exécution du comportement `do` et non pas l'occurrence de l'événement. La
 14 présence de la contrainte dans l'état permet également d'éviter de gérer la contrainte dans la tran-
 15 sition étiquetée avec l'événement `ticketSuccess` et dans la transition étiquetée avec l'événement
 16 `ticketFailure`.

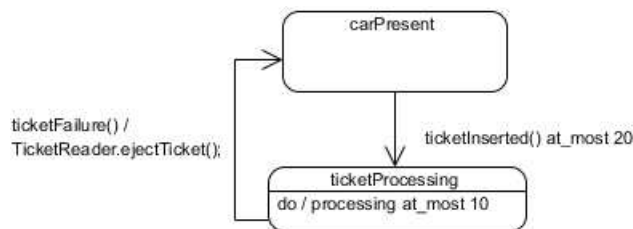


FIGURE 8.8 – Le temps dans les états

17 **Mesurer le temps** Les horloges ne sont pas utilisées dans notre annotation des diagrammes
 18 états-transitions. Cependant, il est possible d'utiliser une horloge globale pour chronométrer le
 19 temps globale du modèle mais également définir des horloges locales pour chaque état. L'absence
 20 d'horloge globale dans nos annotations temporelles est due à la phase de vérification de notre
 21 modèle qui se fait dans les réseaux de Petri colorés temporisés (après la traduction) et non pas
 22 dans les diagrammes états-transitions.

23 Unités de temps

24 La diversité des contraintes temporelles dans les systèmes temporisés implique l'utilisation
 25 de différentes unités de temps. Nous considérons un ensemble d'unités qui accompagnent les
 26 expressions temporelles dans les diagrammes états-transitions. L'unité de temps par défaut est

1 la seconde et dans le cas où il n'y a pas d'unité dans l'expression temporelle cela implique que
2 l'unité est la seconde.

3 Les unités prises en compte dans cette définition sont les suivantes :

- 4 • *s* secondes qui est l'une des mesures de base
- 5 • *ms* milliseconde définie par un facteur de 0,001 par rapport aux secondes
- 6 • *ns* nanoseconde définie par un facteur de 0,001 par rapport aux millisecondes
- 7 • *min* minutes définie par un facteur de 60 par rapport aux secondes
- 8 • *h* heure définie par un facteur de 60 par rapport aux minutes
- 9 • *day* jour défini par un facteur de 24 par rapport aux heures

10 D'autres travaux considèrent les unités de temps dans leurs annotations tels que MARTE. Ce
11 dernier comprend plus d'unités telles que les tic. Nous ne considérons pas les tic dans les unités
12 de temps que nous prenons en compte car c'est dédié à la représentation des unités dans le cas
13 de processeurs. Étant donné que nos études de cas ne considèrent pas les tic, nous les avons pas
14 considéré dans nos unités de temps.

15 Les unités de temps dans le model checking

16 Dans la phase de vérification les unités de temps ne sont pas considérées et par conséquent
17 dans le cas de la présence de ces unités dans les diagrammes états-transitions cela peut créer des
18 problèmes. C'est pour cette raison qu'il faut faire une conversion des unités vers l'unité la plus
19 petite du modèle pour que le modèle soit homogène.

20 Comparaisons

21 Afin de renforcer la raison de notre choix nous proposons une comparaison des différents élé-
22 ments pris en considération selon notre classification avec d'autres travaux (voir le [Tableau 8.2](#)).

23 ✓ : la contrainte est considérée

24 × : la contrainte peut être modélisée par l'approche

25 **Le temps dans UML** Nous avons présenté dans la [Section 3.3](#) les contraintes temporelles
26 qui sont considérés dans la spécification d'UML. Les deux éléments syntaxiques utilisés pour la
27 représentation des contraintes temporelles dans UML sont **at** et **after**. L'élément syntaxique
28 **after** correspond à l'élément **at** que nous utilisons pour la représentation de la latence. L'élé-
29 ment syntaxique **at** correspond à la représentation du temps absolu vis à vis l'occurrence des
30 événements. Nous avons la possibilité de représenter cette contrainte (représentée par l'élément
31 syntaxique **at** dans UML) en supposant qu'il y a un événement de début qui apparaît et à par-
32 tir du quel nous pouvons dire un événement *e* apparaît un temps *T* après cet événement. Par
33 exemple, si l'événement de début est t_0 qui intervient au temps 0, alors la contrainte *e at n* (en
34 utilisant le **at** d'UML) sera traduite par *e at n after t₀* dans notre traduction. Cette sémantique
35 correspondra exactement à la sémantique utilisée dans UML avec l'élément syntaxique **at**.

Éléments	nous	[?]	[?]	[?]	BasicUML
punctuality	✓	✓	✓	✓	
maximumDelay	✓		✓	✓	
minimumDelay	✓	✓	✓	✓	
latency	✓		✓	✓	✓
sequence	✓		✓		
precedence	✓		✓		
simultaneity			✓		
absolute date	×			×	✓
state	✓				
Delay+sequence	✓			✓	
Delay+precedence	✓	✓			
punctuality+sequence	✓	✓		✓	
punctuality+precedence	✓				
others constraints				✓	

TABLE 8.2 – Comparaison

1 MARTE [?]

2 La gestion du temps présentée dans MARTE est différente de ce que nous considérons dans
 3 cette thèse. MARTE base sa prise en compte du temps par la gestion des horloges physiques et
 4 logiques et par la prise en compte de trois contraintes principales.

- 5 • Contraintes basées sur la coïncidence (*Coincidence-based constraints*)
- 6 • Contraintes basées sur la précéence (*Precedence-based constraints*)
- 7 • Combinaison des relations de coïncidence et de précéence (*Mixed constraints*)

8 Contraintes

9 Le [Tableau 8.2](#) montre que nous considérons dans notre travail les contraintes temporelles les
 10 plus utilisés. Par exemple, notre représentation de la séquence en utilisant l'élément syntaxique
 11 **after** est équivalente à la clause 12 (*CyclicTimedResponses*) représentée dans [?] (le travail dans
 12 [?] reprend le même esprit de la représentation des contraintes temporelles que celui dans [?]).

13 Nous sommes convaincu qu'il existe d'autres contraintes que nous ne considérons pas dans
 14 nos annotations (comme par exemple la gestion des horloges et toutes les contraintes autour, etc.)
 15 car nous souhaitons prendre en compte les contraintes les plus utilisés et garder notre syntaxe à
 16 la fois simple et expressive.

1 Exemples : prise en compte des contraintes temporelles

2 **Barrière de parking** La [Figure 8.9](#) montre l'introduction des contraintes temporelles sur
3 un exemple de barrière de parking. Comme montré dans l'exemple, une seule voiture est traitée
4 à la fois, et le temps entre l'arrivée d'une voiture et son départ ne doit pas dépasser 150 secondes.
5 Étant donné que le passage de la voiture par la barrière est composée de plusieurs étapes, le temps
6 que la voiture ne doit pas dépasser dans son passage sera réparti sur ces différentes étapes. En
7 effet, la valeur 150 secondes a été obtenue en additionnant les différentes valeurs des contraintes
8 dans le diagramme. Cette addition est possible car il n'y a que des contraintes `at_most` où toute
9 transition avec cette contrainte doit être franchie au plus une durée de temps. Dans le cas où
10 il y a d'autres contraintes (telles que `at` ou `at_least`) l'addition sera impossible. Par exemple,
11 l'insertion du ticket doit se faire au plus 20 secondes. La détection de l'échec de la contrainte
12 temporelle se fera dans la phase de vérification. La prise en compte des contraintes temporelles
13 dans l'exemple de la [Figure 8.9](#) se manifeste de deux manières. Dans la première nous gérons
14 le temps d'apparition des événements. Par exemple, afin de définir une condition temporelle sur
15 l'insertion d'un ticket nous utilisons la contrainte suivante `ticketInserted() at_most 20`, qui
16 veut dire "l'insertion du ticket doit se faire au plus 20 secondes après l'arrivée de la voiture".
17 Cette contrainte est définie pour l'occurrence de l'événement car le temps ici concerne le moment
18 où le système change d'état.

19 Dans le seconde nous gérons le temps d'exécution du comportement `do` de chaque état.
20 Par exemple, afin de définir une condition temporelle concernant le traitement d'un ticket nous
21 utilisons dans l'état `ticketProcessing` la contrainte suivante `do/processing at_most 10`, qui
22 veut dire "le traitement du ticket (représenté ici par le comportement `do processing`) doit se
23 faire au plus 10 secondes après l'activation de l'état `ticketProcessing`". Autrement dit, le
24 système peut rester dans l'état `ticketProcessing` au plus 10 secondes.

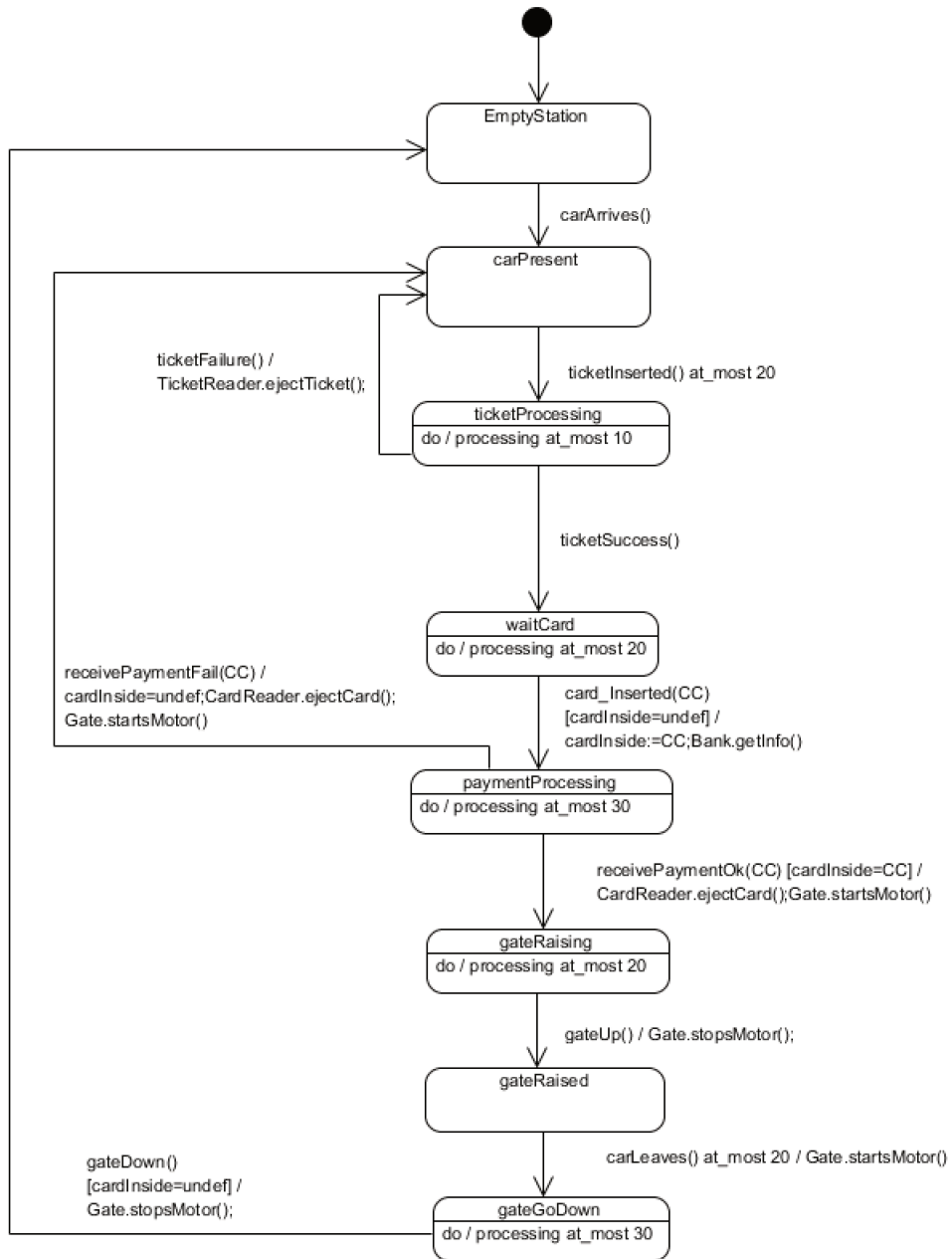


FIGURE 8.9 – Exemple de la barrière de parking temporisé

1 **Bibliothèque** La Figure 8.10 montre l'introduction des contraintes temporelles sur l'exemple
 2 de la bibliothèque. Dans la modélisation de l'exemple de la bibliothèque (voir Chapitre 4) nous
 3 avons remarqué que certaines parties nécessitaient des contraintes temporelles. De ce fait nous
 4 avons intégré le temps dans l'exemple et cela dans deux parties principales. La première dans
 5 l'état `tagged`, le bibliothécaire ne peut pas retirer un livre plus 168 heures (7 jours). La seconde
 6 vis-à-vis de l'événement `bringBack`, l'utilisateur ne peut pas garder un livre plus de 168 heures.
 7 Cela est modélisé dans la transition étiquetée avec l'événement `bringBack` avec la contrainte
 8 `bringBack at_most 168h after lend()`. Autrement dit, l'événement `bringBack` doit apparaître
 9 au plus 168 heures après l'occurrence de l'événement `lend`. L'erreur engendrée en cas de dépas-
 10 sement dans la contrainte de temps est prise en compte lors de la phase de vérification et non
 11 pas dans le diagramme états-transition.

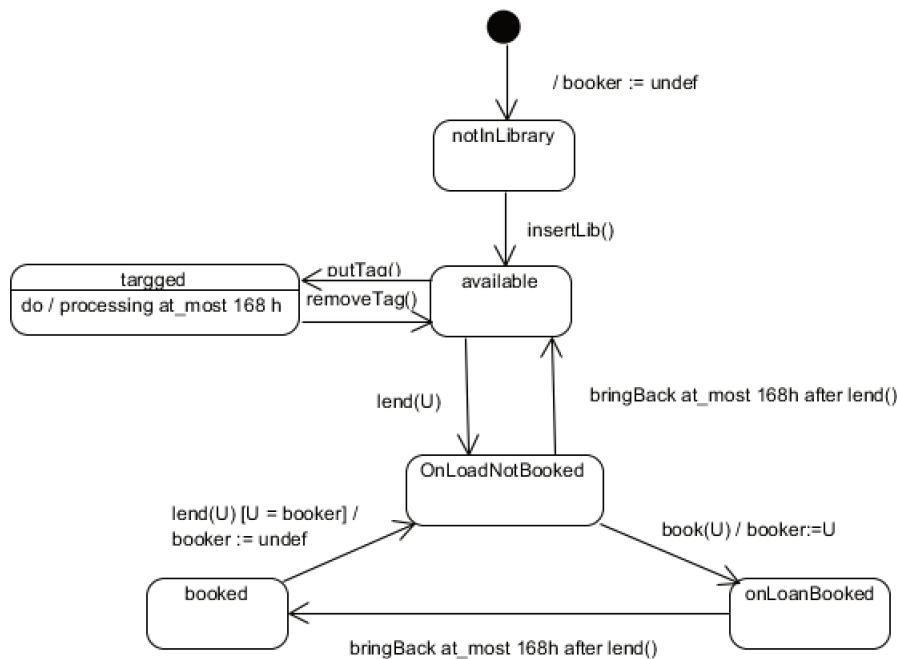


FIGURE 8.10 – L'exemple de la bibliothèque temporisé

1 Location de scooter

2 Nous présentons dans ce qui suit la description textuelle de l'exemple de *location de scooter*
 3 qui sera utilisée pour la conception du diagramme états-transitions correspondant.

4 *Afin de faciliter le déplacement des touristes lors de leurs visites, une compagnie a décidé de créer des stations de location de scooters. Chaque touriste (ou client) a le droit de louer un scooter pour une durée déterminée mais qui ne doit pas dépasser 12h. La réparation du scooter en cas de dysfonctionnement ne doit également pas dépasser 12h. Le client peut vérifier le scooter au moment de la location et signaler par conséquent que le scooter est défectueux. Le client ne paye rien dans le cas où le scooter est cassé avant qu'il le loue, et ce sera la compagnie qui s'occupera de réparer le scooter. Cependant s'il signale une panne après avoir pris le scooter, alors ce sera lui qui payera les frais de réparation. Au moment de louer le scooter, le client a le droit de payer en liquide ou par carte bancaire. Une fois que le client termine l'usage du scooter, il le ramène à la même station où il l'a retiré. Les stations de location de scooters sont ouvertes 24h sur 24 et 7 jours sur 7.*

5 Afin de modéliser cet exemple, nous avons constaté qu'il est possible de considérer différents
 6 points de vue :

- 7 • Point de vue la station : nous pouvons modéliser cet exemple en considérant le fonction-
 8 nement de la station comme acteur principal.
- 9 • Point de vue scooter : nous pouvons également considérer le scooter et les différentes étapes
 10 de son allocation comme acteur principal.
- 11 • Point de vue client : enfin, nous pouvons considérer le client et ce qu'il peut faire dans une
 12 station comme acteur principal.

13 Ce qui nous intéresse dans cet exemple sont les différentes étapes de la location du scooter
 14 et donc par conséquent modéliser cet exemple du point de vue scooter. La [Figure 8.11](#) montre
 15 le diagramme états-transitions correspondant à l'exemple du scooter en prenant en compte la
 16 gestion des contraintes temporelles.

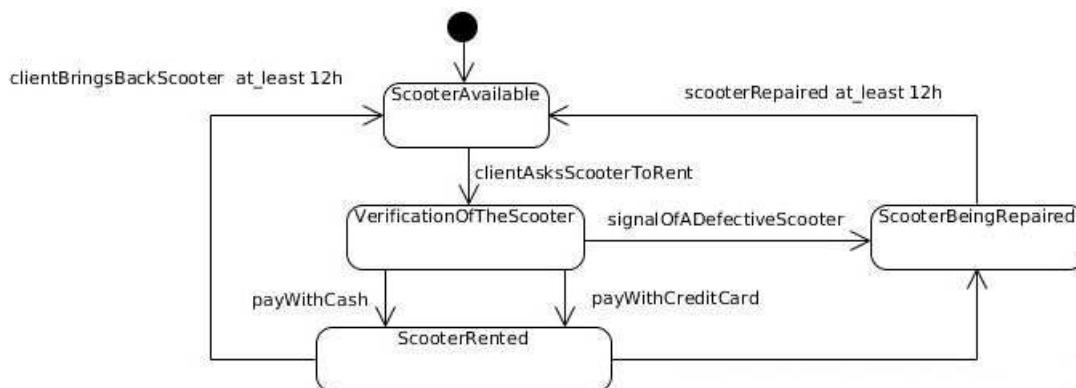


FIGURE 8.11 – L'exemple de location de scooter temporisé (version 1)

17 Nous avons constaté lors de la modélisation des contraintes temporelles que les deux événe-
 18 ments venant de l'état **ScooterRented** ont une contrainte temporelle qui est la même. Afin d'évi-
 19 ter de gérer cette contrainte séparément pour chaque événement et par conséquent augmenter la
 20 complexité, nous avons exprimé la contrainte temporelle au sein de l'état **ScooterRented**. Cette
 21 contrainte concerne le fait que le scooter soit loué pour une durée de 12h. Dans la [Figure 8.12](#)
 22 la contrainte temporelle est supprimée des deux transitions venant de l'état **ScooterRented** et

- 1 ajoutée dans l'état `ScooterRented` en considérant que le comportement de cet état (qui est
- 2 l'utilisation du scooter) ne dépasse pas 12h.

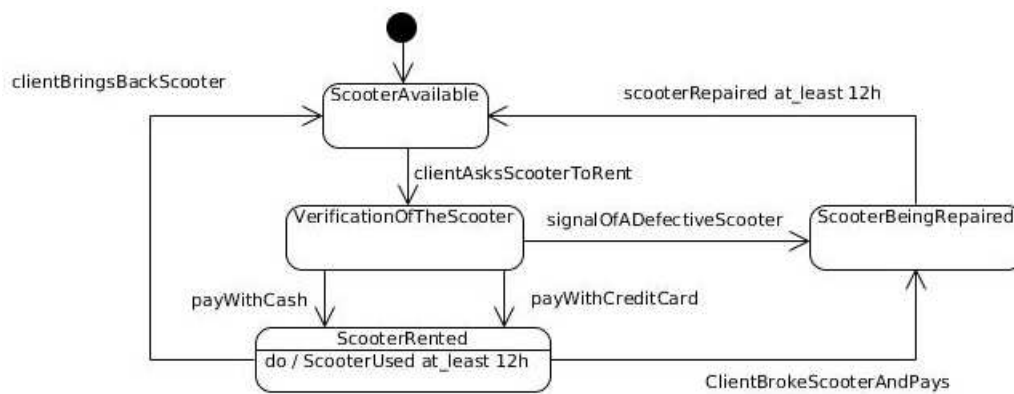


FIGURE 8.12 – L'exemple de location de scooter temporisé (version 2)

1 Le temps dans les réseaux de Petri colorés

2 Il est possible dans les réseaux de Petri colorés d'ajouter des contraintes temporelles et cela
 3 afin de modéliser et vérifier les systèmes temps-réel. En effet, on s'intéresse ici à l'ajout du temps
 4 sur les places. La différence majeure entre un réseau de Petri coloré et un réseau de Petri coloré
 5 temporisé est la présence d'une valeur appelée estampillage de date (*time stamp*). Cela signifie
 6 que le marquage d'une place qui a des jetons avec estampillage de date est maintenant un multi-
 7 ensemble temporisé spécifiant les éléments du multi-ensemble avec leur nombre d'occurrence, leur
 8 valeur et leur estampillage de date. Par exemple dans la [Figure 8.13](#), le nombre d'occurrence du
 9 jeton $1'(2, "g \text{ and } An")@0$ est 1, la valeur du jeton est (2, "g and An") et l'estampillage de date
 10 est 0. La valeur de l'estampillage de date peut être un entier ou un réel positif. L'estampillage de
 11 date permet de savoir à quel moment le jeton peut être consommé de la place avec une transition
 12 franchissable (ayant comme source cette place). La valeur des jetons peut avoir un estampillage
 13 de date dans le cas où la place contenant ces jetons a un type temporisé.

14 Notons que l'estampillage de date par défaut est $@0$ et par conséquent les jetons avec cet
 15 estampillage peuvent être consommés à n'importe quel moment.

16 Le modèle *CPN* possède une horloge globale qui représente le temps du modèle. L'exécution
 17 d'un modèle *CPN* avec du temps est contrôlée avec cette horloge globale et cela fonctionne de la
 18 même manière que la gestion de la file des événements utilisée dans la plupart des simulations des
 19 événements discrets. La valeur de cette horloge ne change pas tant qu'il y a des transitions qui
 20 sont franchissables, c'est-à-dire des transitions ayant des places sources avec un type temporisé
 21 ou non. La valeur de l'horloge globale change s'il n'y a pas de transition franchissable et sa valeur
 22 prend alors celle du jeton ayant l'estampillage de date le plus proche. Chaque marquage se trouve
 23 dans un intervalle fermé dans le modèle *CPN* temporisé.

24 Notons qu'il est possible d'avoir un conflit dans le cas où il y a des transitions franchissables
 25 en même temps. Notons également que la distribution des jetons avec leur estampillage de date
 26 dans les places ainsi que la valeur de l'horloge globale est appelée *marquage temporisé*.

27 La [Figure 8.13](#) modélise l'exemple d'envoi de paquets avec accusé de réception (l'exemple
 28 est présenté dans [?]). Nous remarquons dans cet exemple qu'il y a un ensemble de jetons sans
 29 présence de temps, c'est-à-dire que tous les estampillages de date sont à 0 par défaut. Le chan-
 30 gement de valeur des estampillages de date se fait grâce aux expressions dans le compartiment
 31 temporel des transitions ou aux expressions dans les arcs. Par exemple, dans le compartiment
 32 temporel de la transition `SendPacket` il y a l'expression $@ + 9$. Cela veut dire qu'au moment de
 33 franchir la transition l'estampillage de date du jeton consommé sera augmenté de 9. Notons que
 34 la valeur de l'horloge globale de ce modèle est initialement égale à 0 (au début de la vérification).

35 La [Figure 8.14](#) montre le réseau de Petri coloré de la [Figure 8.13](#) avec le marquage initial.
 36 Notons que les transitions franchissables sont celles entourées d'un halo vert. Dans le marquage
 37 initial de cet exemple, seule la transition `SendPacket` est franchissable car :

- 38 • Il y a un jeton dans la place `NextSend` ($n = 1$), il y a un jeton dans la place `Send` ($n = 1$)
 39 et il y a un jeton dans la place `Wait` (ces places sont les places source de la transition
 40 `SendPacket`).
- 41 • La valeur de chaque jeton de ces trois places comporte un estampillage de date égal à 0 et
 42 qui est égal à la valeur de l'horloge globale (c'est-à-dire 0).

43 La [Figure 8.15](#) montre l'étape suivante après le franchissement de la transition `SendPacket`.

44 Après le franchissement de la transition `SendPacket` nous remarquons plusieurs changements
 45 comme suit :

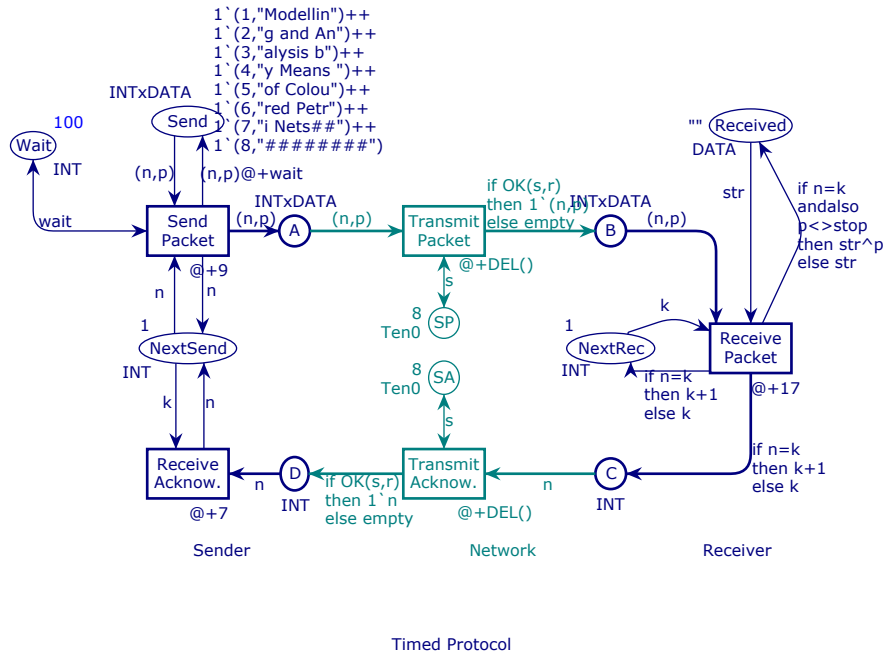


FIGURE 8.13 – Exemple d’un réseau de Petri coloré avec du temps ([?])

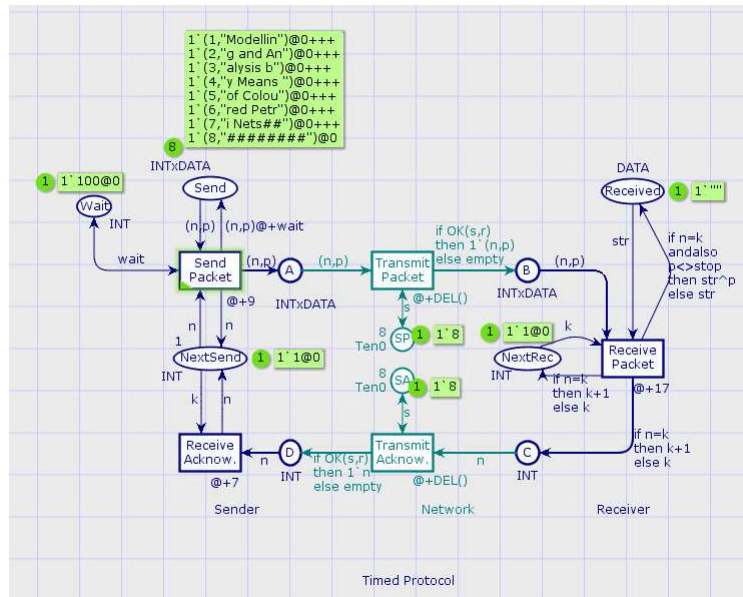


FIGURE 8.14 – Exemple d’un réseau de Petri coloré avec du temps avec marquage ([?])

- 1 • La consommation du jeton (1, "Modellin")@0 dans la place Send et sa remise dans la même
- 2 place mais avec un estampillage différent (1, "Modellin")@109. Ce changement d'estampillage de date est dû à l'expression @ + 9 dans le compartiment temporel de la transition
- 3 SendPacket, mais aussi à l'expression dans l'arc allant de la transition SendPacket à la
- 4 place Send ((n, p)@ + wait) avec wait = 100).
- 5
- 6 • La consommation du jeton 100@0 dans la place Wait et sa remise dans la même place
- 7 mais avec un estampillage différent 100@9. Ce changement d'estampillage de date est dû
- 8 à l'expression @ + 9 dans le compartiment temporel de la transition SendPacket.

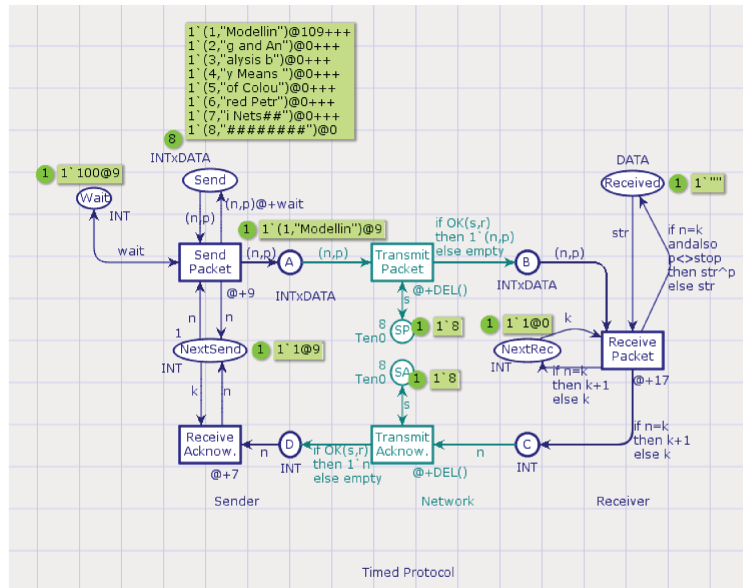


FIGURE 8.15 – Franchissement de la transition SendPacket

- 1 • La production du jeton (1,"Modellin")@9 dans la place A.
- 2 Après le franchissement de la transition SendPacket, on ne peut plus franchir de transition.
- 3 En effet, la transition SendPacket est infranchissable car l'estampillage de date du jeton 100 dans
- 4 la place Wait est égal à 9. La transition TransmitPacket est infranchissable car l'estampillage
- 5 de date du jeton (1,"Modellin") dans la place A est égal à 9. Le reste des transitions ne sont
- 6 pas franchissables car il n'y a pas de jetons dans toutes leurs places source. Par conséquent, il
- 7 est nécessaire d'incrémenter la valeur de l'horloge à 9, c'est-à-dire l'estampillage de date le plus
- 8 proche afin que la transition TransmitPacket soit franchissable. La Figure 8.16 montre la valeur
- 9 de l'horloge globale avant et après l'incrémentation de l'horloge globale.

TimedProtocol.cpn	TimedProtocol.cpn
Step: 0	Step: 1
Time: 0	Time: 9
Options	Options
History	History
Declarations	Declarations
colset INT	colset INT
colset DATA	colset DATA
colset INTxDATA	colset INTxDATA
var n k wait	var n k wait
var p str	var p str
val stop	val stop
colset Ten0	colset Ten0
colset Ten1	colset Ten1
var s	var s
var r	var r
fun OK	fun OK
colset NetDelay	colset NetDelay
fun DEL	fun DEL
Monitors	Monitors
Top	Top
Avant	Après

FIGURE 8.16 – Valeur de l'horloge globale avant et après le franchissement de la transition SendPacket

1 Une fois que la valeur de l'horloge globale est incrémentée, la transition TransmitPacket
2 devient franchissable car :

- 3 • Il y a un jeton dans la place A.
4 • L'estampillage de date du jeton dans cette place est égal à la valeur de l'horloge globale,
5 c'est-à-dire 9.

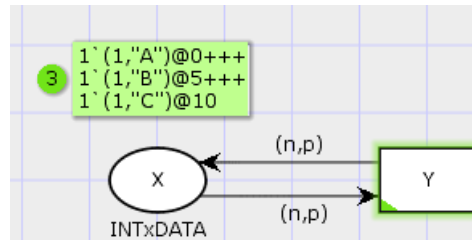


FIGURE 8.17 – Gestion des estampillages de date dans les transitions

6 La Figure 8.17 montre un simple exemple avec une place (ayant plusieurs jetons) et une
7 transition qui consomme et produit dans cette place. Afin que la transition soit franchissable il
8 suffit qu'un seul jeton possède un estampillage de date inférieur ou égal à la valeur de l'horloge
9 globale. Par exemple, dans la place X il y a trois jetons, le premier avec un estampillage de date
10 de 0, le second avec un estampillage de date de 5 et le dernier avec un estampillage de date de
11 10. La valeur de l'horloge globale est initialement égale à 0 et par conséquent la transition Y est
12 franchissable car il y a au moins un jeton (celui avec l'estampillage de date égal à 0) qui a un
13 estampillage de date inférieur ou égal à 0.

14 Introduction des contraintes temporelles dans la traduc- 15 tion

16 Afin de prendre compte les contraintes temporelles et de vérifier les systèmes temporisés,
17 nous présentons dans cette section l'intégration des contraintes temporelles dans notre traduc-
18 tion. Nous utilisons les diagrammes états-transitions annotés avec du temps (présentés dans le
19 Chapitre 8), ainsi que les réseaux de Petri colorés temporisés (présentés dans la Section 2.3).

20 L'intégration du temps dans notre traduction est basée sur un ensemble de règle permettant
21 de représenter chaque contrainte (sa syntaxe et sa sémantique) dans les réseaux de Petri colorés.
22 Nous utilisons un ensemble de diagrammes états-transitions pour représenter les contraintes et
23 nous donnons leurs correspondant dans les réseaux de Petri colorés temporisés.

24 Traduction du délai minimum

25 Le délai minimum est modélisé en utilisant l'élément syntaxique `at_least` et qui veut dire
26 qu'un événement doit apparaître au minimum un délai donné après l'activation de son état
27 source. L'exemple de la Figure 8.18 montre le diagramme états-transition (représentant le délai
28 minimum) et sa traduction correspondante.

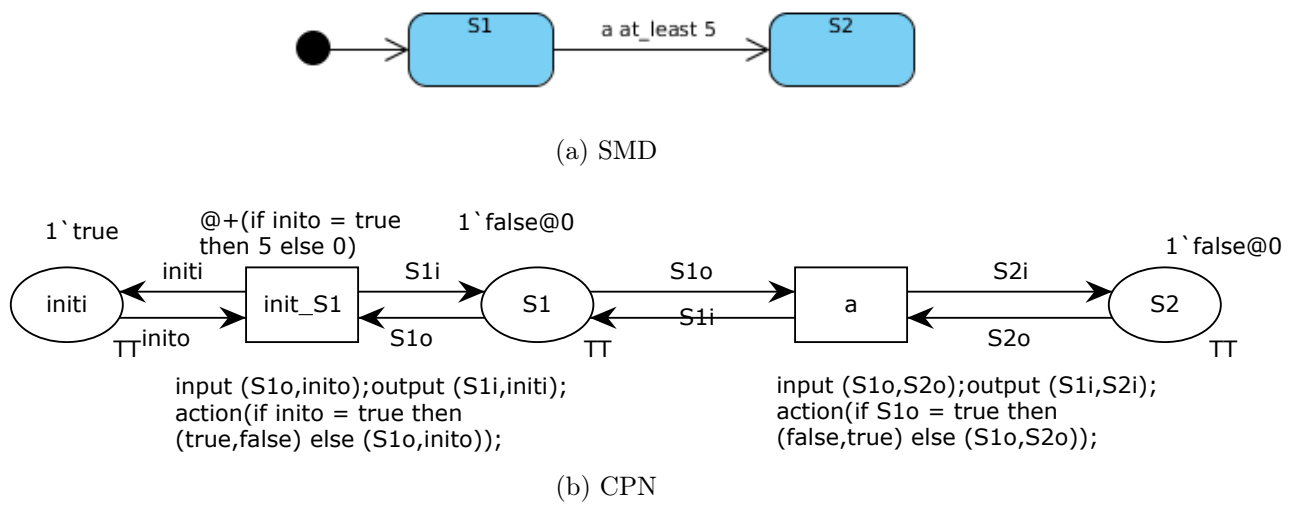


FIGURE 8.18 – Traduction d'at_least

1 Dans l'exemple de la [Figure 8.18a](#) nous avons une transition étiquetée avec l'événement **a**
 2 (allant de l'état **S1** vers l'état **S2**). Cette transition ne peut pas être franchie que si l'événement **a**
 3 apparaît au minimum 5 secondes (ou plus) après l'activation de l'état **S1**. Rappelons qu'un jeton
 4 avec un estampillage de date dans les réseaux de Petri colorés temporisés ne peut être consommé
 5 que si son estampillage est inférieur ou égal à celui de l'horloge globale; cette sémantique cor-
 6 rrespond à la sémantique du délai minimum. Par conséquent, nous traduisons cet exemple en
 7 utilisant les [Algorithmes 1](#) et [2](#) et en ajoutant un jeton avec 5 unités de temps (l'estampillage de
 8 ce jeton est engendré grâce à la partie temporelle de la transition `init_S1`). Une fois que la place
 9 **S1** sera active le jeton aura la valeur 5 et par conséquent la transition ne sera franchie que si
 10 l'horloge globale soit à la valeur 5 ou plus.

11 Traduction de la ponctualité

12 La ponctualité est modélisée en utilisant l'élément syntaxique `at` et qui veut dire qu'un événe-
 13 ment doit apparaître exactement à un délai donné après l'activation de son état source. L'exemple
 14 de la [Figure 8.19](#) montre le diagramme états-transition (représentant la ponctualité) et sa tra-
 15 duction vers les réseaux de Petri colorés temporisés correspondante.

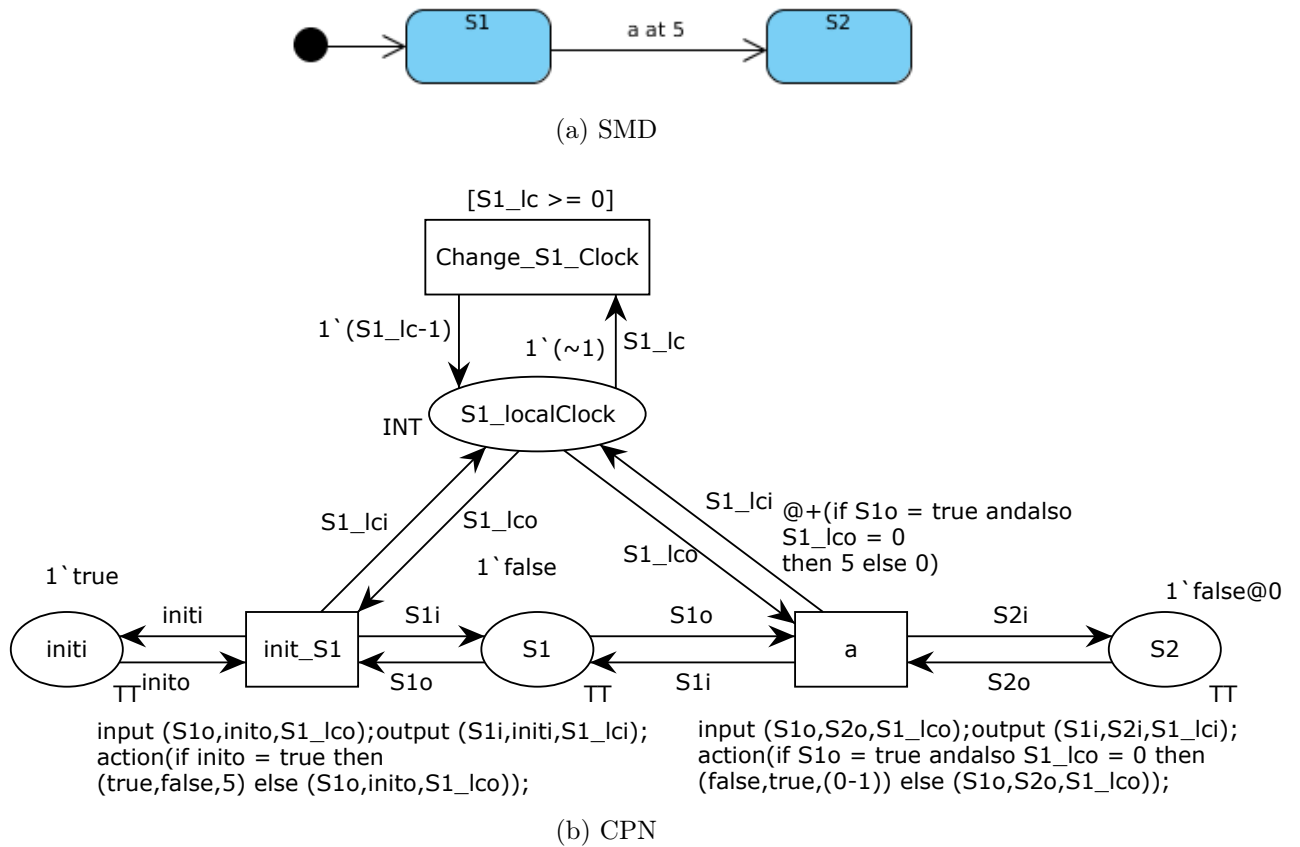
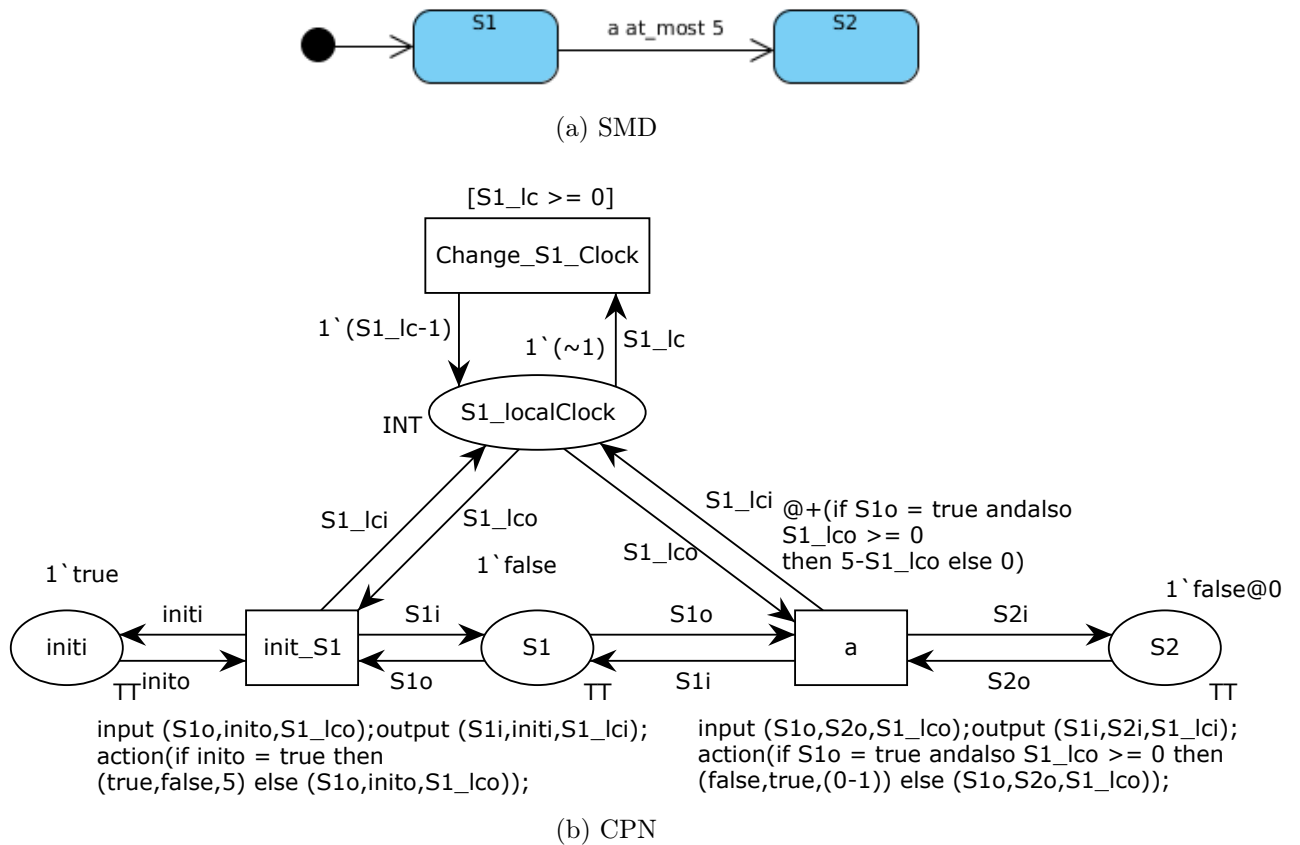


FIGURE 8.19 – Traduction d'at

1 Dans l'exemple de la [Figure 8.19a](#) nous avons une transition étiquetée avec l'événement **a**
 2 (allant de l'état **S1** vers l'état **S2**). Cette transition ne peut pas être franchie que si l'événement **a**
 3 apparaît exactement 5 secondes après l'activation de l'état **S1**. Étant donné que la sémantique de
 4 la ponctualité est différente de la sémantique donnée pour la consommation des jetons avec un
 5 estampillage de date, nous ajoutons un mécanisme afin que la sémantique de cette contrainte (la
 6 ponctualité) soit respectée. Nous utiliserons un compteur (modélisé par la place **S1_localClock**)
 7 et qui contiendra la valeur 5 et une transition (modélisée avec **Change_S1_Clock**) qui permet de
 8 décrémenter la valeur stockée dans la place **S1_localClock**. La place **S1_localClock** est initiale-
 9 ment à la valeur -1 car la place **S1** (concernée par le compteur) n'est pas encore active. Une fois
 10 que cette place est active la valeur du jeton dans la place **S1_localClock** sera à 5. La transition
 11 **Change_S1_Clock** pourra être franchie tant que la valeur du jeton est inférieure ou égale à 5. La
 12 transition **a** sera franchissable dans le cas où la place **S1** comporte un jeton **true** (c'est-à-dire la
 13 place est active) et dans le cas où la valeur du jeton dans la place **S1_localClock** est à 0. Dans
 14 le cas où la valeur du jeton dans la place **S1_localClock** devient négative il ne sera plus possible
 15 de franchir la transition **a**.

16 Traduction du délai maximum

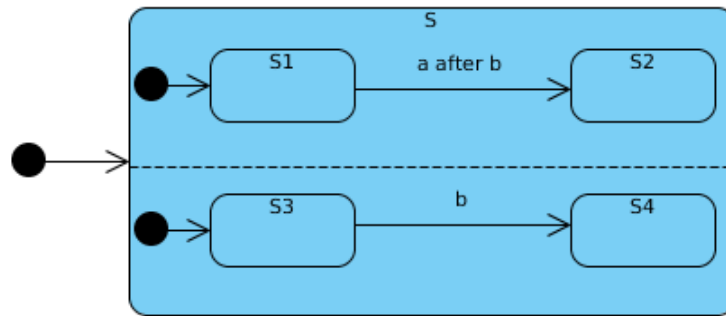
17 Le délai maximum est modélisé en utilisant l'élément syntaxique **at_most** et qui veut dire
 18 qu'un événement doit être apparaître au maximum à un délai donné après l'activation de son
 19 état source. L'exemple de la [Figure 8.20](#) montre le diagramme états-transitions (représentant le
 20 délai maximum) et sa traduction vers les réseaux de Petri colorés temporisés correspondante.

FIGURE 8.20 – Traduction d'*at_most*

1 Dans l'exemple de la Figure 8.19a nous avons une transition étiquetée avec l'événement *a*
 2 (allant de l'état *S1* vers l'état *S2*). Cette transition ne peut pas être franchie que si l'événement *a*
 3 apparaît au plus 5 secondes après l'activation de l'état *S1*. Étant donné que la sémantique du délai
 4 maximum est différente de la sémantique donnée pour la consommation des jetons avec un estam-
 5 pillage de date, nous reprenons le mécanisme utilisé pour la ponctualité afin que la sémantique de
 6 cette contrainte (le délai maximum) soit respectée. Nous utiliserons la place *S1_localClock* (conten-
 7 ant initialement un jeton avec la valeur *-1*) et la transition *Change_S1_Clock* pour représenter
 8 le mécanisme (celui du compteur). La différence avec le mécanisme utilisé dans la ponctualité est
 9 la garde dans la transition *a*, c'est-à-dire au lieu d'avoir comme garde $S1_lco = 0$ nous avons la
 10 garde $S1_lco \leq 5$ (*S1_lco* est la variable transportant le jeton de la place *S1_localClock*). La
 11 transition *a* sera franchissable dans le cas où la place *S1* comporte un jeton *true* (c'est-à-dire la
 12 place est active) et dans le cas où la valeur du jeton dans la place *S1_localClock* est supérieure ou
 13 égale 0 (cela est modélisé par la garde $S1_localClock \geq 0$ dans le code segment de la transition
 14 *a*). Dans le cas où la valeur du jeton dans la place *S1_localClock* devient négative il ne sera plus
 15 possible de franchir la transition *a*.

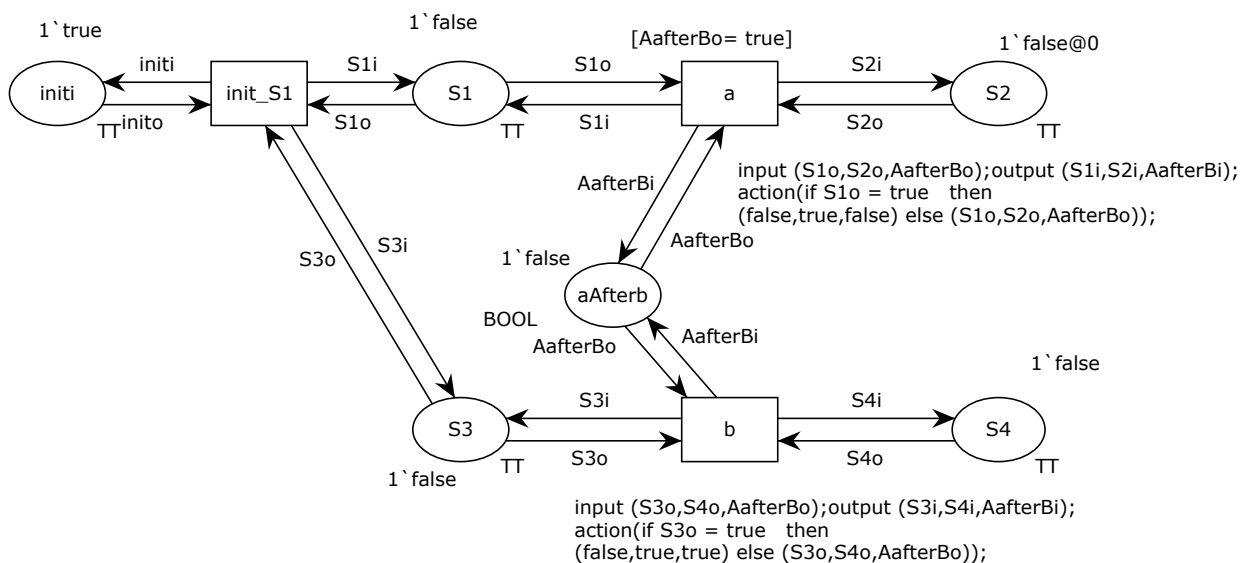
16 Traduction de la séquence d'événements

17 La séquence d'événements est modélisée en utilisant l'élément syntaxique *after* et qui veut dire
 18 qu'un événement doit être apparaître après l'occurrence d'un autre. L'exemple de la Figure 8.21
 19 montre le diagramme états-transitions (représentant la séquence d'événement) et sa traduction
 20 vers les réseaux de Petri colorés temporisés correspondante.



(a) SMD

```
input (inito,S1o,S3o);output (initi,S1i,S3i);
action(if inito = true then
(false,true,true) else (inito,S1o,S3o));
```



(b) CPN

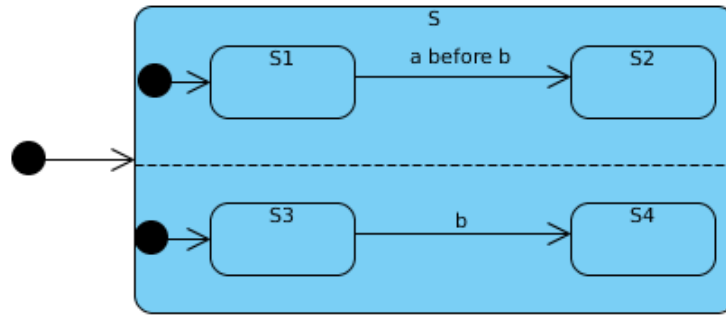
FIGURE 8.21 – Traduction d'after

1 Dans l'exemple de la Figure 8.21a nous avons une transition étiquetée avec l'événement **a**
2 (allant de l'état **S1** vers l'état **S2**) et une transition étiquetée avec l'événement **b** (allant de l'état
3 **S3** vers l'état **S4**). La transition avec l'événement **a** ne peut être franchie que si la transition
4 avec l'événement **b** soit déjà franchie. Dans la Figure 8.21 nous présentons le réseaux de Petri
5 coloré résultat de la traduction de l'exemple dans la Figure 8.21a. Afin de prendre en compte
6 le séquençement des deux événements **a** et **b** nous utilisons une place supplémentaire **aAfterb**
7 qui comporte initialement un jeton de valeur *false* ($1'false$). Tant que la valeur du jeton dans
8 cette place est à *false* la transition **a** est infranchissable (cela est modélisé par la présence de la
9 garde $[AafterBo = true]$). Une fois que la transition **b** est franchie, la valeur du jeton change à
10 *true* et la transition **a** devient franchissable. La valeur du jeton est remise à *false* une fois que la
11 transition **a** est franchie. Notons que dans notre traduction pour la prise en compte des priorités
12 entre événements (voir la Section 5.9), nous définissons une transition pour chaque ensemble de
13 transitions étiquetées avec le même événement. Par conséquent, la définition que nous présentons
14 concernant le séquençement de transitions est prise en compte. En effet, la transition **a** doit être
15 franchie à la dernière occurrence de l'événement **b**; et étant donné qu'il y a une seule transition

1 pour l'événement **b** alors la définition est respectée.

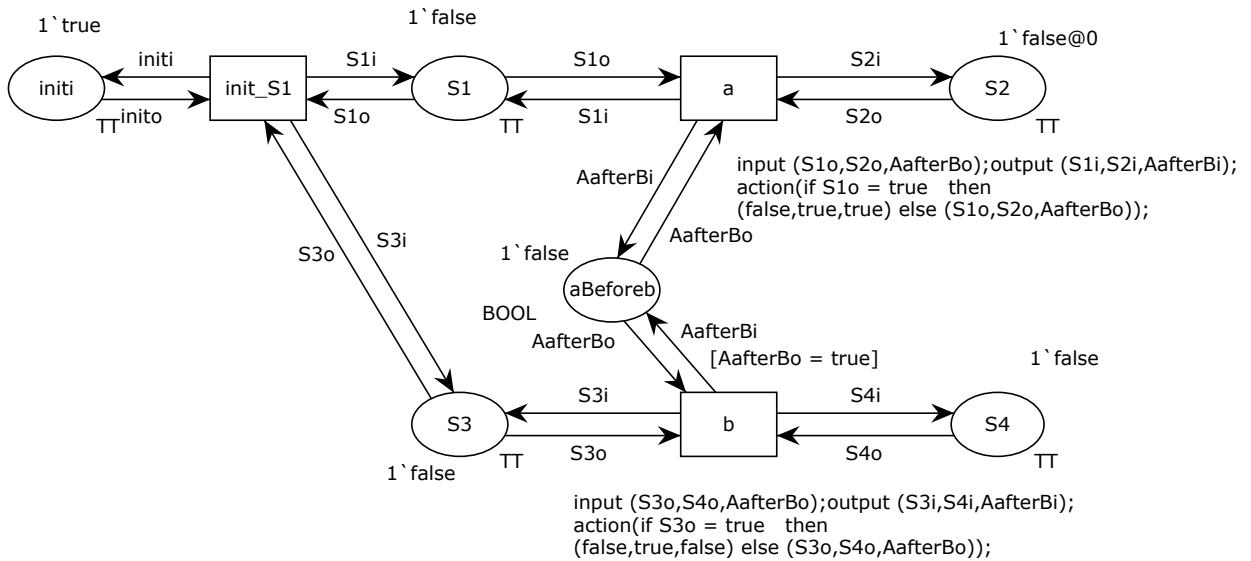
2 **Traduction de la précédence**

3 La précédence d'événements est modélisée en utilisant l'élément syntaxique **before** et qui veut
 4 dire qu'un événement doit être apparaître avant un autre. L'exemple de la [Figure 8.22](#) montre le
 5 diagramme états-transitions (représentant la précédence d'événement) et sa traduction vers les
 6 réseaux de Petri colorés temporisés correspondante.



(a) SMD

```
input (inito,S1o,S3o);output (initi,S1i,S3i);
action(if inito = true then
(false,true,true) else (inito,S1o,S3o));
```



(b) CPN

FIGURE 8.22 – Traduction de **before**

7 Dans l'exemple de la [Figure 8.22a](#) nous avons une transition étiquetée avec l'événement **a**
 8 (allant de l'état **S1** vers l'état **S2**) et une transition étiquetée avec l'événement **b** (allant de l'état
 9 **S3** vers l'état **S4**). La transition avec l'événement **b** ne peut être franchie que si la transition
 10 avec l'événement **a** soit déjà franchie. Dans la [Figure 8.21](#) nous présentons le réseaux de Petri
 11 coloré résultat de la traduction de l'exemple dans la [Figure 8.21a](#). Le mécanisme pour gérer la
 12 précédence des événements est le même que celui pour le séquençement (présence de la place
 13 supplémentaire et utilisation de gardes).

1 Conclusion

2 Nous avons présenté dans cette section un travail préliminaire concernant l'intégration des
3 contraintes temporelles dans ce travail de thèse. Nous proposons une syntaxe et une sémantique
4 pour annoter les diagrammes états-transitions avec du temps. Nous présentons ensuite un début
5 d'intégration des contraintes temporelles dans notre traduction des diagrammes états-transitions
6 vers les réseaux de Petri colorés.

Chapitre 9

Conclusion

Résultats

Le travail proposé dans cette thèse a pour but de valider et de vérifier les systèmes. Nous avons proposé comme première contribution une amélioration et une extension d'une méthode de spécification des systèmes en utilisant les diagrammes états-transitions. Le but de la méthode est de guider l'utilisateur dans sa démarche de spécification via un ensemble d'étapes et pour de spécifier/modéliser les systèmes en évitant les erreurs communes commises lors de la phase de spécification/modélisation. Cette méthode prend en entrée la description textuelle du système et le diagramme de classes pour engendrer le diagramme états-transitions correspondant. Nous présentons également l'outil *EasySM* qui implémente la méthode de spécification. Nous avons contribué à améliorer l'outil et à proposer une extension.

Notre seconde contribution consiste à traduire les diagrammes états-transitions (qui modélisent différents systèmes) vers les réseaux de Petri colorés. Nous prenons en compte l'ensemble des éléments syntaxiques suivants : les états (simples, finaux, composites et les états orthogonaux), les comportements (d'entrée, de sortie et do), les variables, les pseudo-états (initiaux, histoires plats, fork et join), la hiérarchie des comportements et des états, les transitions (simples, avec événements, gardes et comportements, internes, externes, locaux, inter-niveau). Chaque élément de la syntaxe des diagrammes états-transitions est représenté par son correspondant dans les réseaux de Petri colorés (par exemple, chaque état simple est représenté par une place, chaque comportement d'entrée ou de sortie est représenté dans la partie code des transitions en réseau de Petri coloré, etc). Notre traduction est basée sur deux algorithmes, le premier algorithme est pour traduire les éléments syntaxiques et le second algorithme est pour la sémantique de sélection des transitions. Nous présentons également l'outil UML2CPN qui implémente la traduction et qui est en cours de développement.

Enfin, nous présentons un travail préliminaire pour l'extension de nos approches en intégrant les contraintes temporelles. Nous proposons une annotation (une syntaxe et une sémantique) pour les diagrammes états-transitions avec du temps. Le but de ces annotations est de les utiliser lors de la méthode de spécification et de les intégrer lors de la traduction des diagrammes états-transitions (en utilisant comme formalisme temporisé les réseaux de Petri colorés temporisés).

Le travail effectué dans cette thèse sera une base pour travailler sur d'autres aspects concernant la modélisation et la vérification. Ces aspects sont présentés dans nos perspectives à court et long terme.

1 Perspectives à court terme

2 **Implémentation** Nous souhaitons améliorer le prototype UML2CPN que nous avons pré-
3 senté dans ce document pour automatiser notre traduction; cela afin de vérifier formellement
4 des diagrammes états-transitions plus complexes mais aussi pour étudier la complexité de notre
5 traduction pour l'optimiser.

6 **Intégration avec d'autres diagrammes** Une autre perspective importante est l'intégra-
7 tion des diagrammes d'activités dans la traduction des diagrammes états-transitions UML. La
8 combinaison des deux diagrammes permettra la modélisation d'autres aspects des systèmes, tels
9 que la représentation des données. Il y a par exemple des travaux précédents que nous pouvons
10 utiliser pour la formalisation des diagrammes d'activités, par exemple, en utilisant les réseaux
11 de Petri colorés [?]. Nous pouvons également nous inspirer du travail qui traduit un ensemble de
12 la syntaxe des machines à états SysML et les diagrammes d'activités vers CSP [?].

13 **Extension de la méthode de spécification** Le travail que nous présentons dans cette
14 thèse à propos de la méthode de spécification considère un ensemble d'éléments syntaxiques
15 des diagrammes états-transitions. Par exemple, les états simples, les états composites, les transi-
16 tions (avec événements, gardes et comportements). Cependant, une perspective importante serait
17 de prendre en compte l'ensemble des éléments syntaxiques considérés dans la traduction. Cela
18 permettrait, une fois que la méthode de spécification a été utilisée, de traduire le diagrammes
19 états-transitions (résultat de la méthode) vers les réseaux de Petri colorés (en utilisant notre
20 traduction).

21 **Contraintes temporelles** Un autre point important à prendre en compte dans nos pers-
22 pectives est l'intégration complète du temps dans nos approches. En effet, nous avons présenté
23 dans cette thèse un travail préliminaire afin de considérer les contraintes temporelles et par consé-
24 quent vérifier les systèmes temporisés. Nous souhaitons compléter l'intégration du temps dans
25 nos approches.

26 **Études de cas** Une autre perspective que nous souhaitons effectuer est d'élargir l'ensemble
27 des études de cas auxquelles nous appliquons nos approches. En effet, dans cette thèse nous
28 détaillons que deux études de cas (la barrière de péage et le lecteur de CD). La prise en compte
29 de plusieurs études de cas permettra de démontrer que nous considérons un ensemble large des
30 systèmes dans nos approches.

31 **Extension de la traduction** Nous souhaitons étendre notre traduction en prenant en
32 compte l'ensemble des éléments syntaxiques des diagrammes états-transitions. Nous pensons que
33 les éléments considérés dans notre thèse sont importants et suffisent à modéliser la plus part des
34 systèmes. Cependant prendre l'ensemble des éléments syntaxiques est un ajout non négligeable
35 afin de considérer l'intégralité des systèmes.

36 Perspectives à long terme

37 **Comparaison avec d'autres outils** Nous souhaitons une fois le développement de notre
38 outil UML2CPN de le comparer avec d'autres outils. La comparaison consiste à utiliser les mêmes
39 diagrammes états-transitions en entrée et comparer le résultat de la traduction. Faire une telle

1 comparaison (en termes d'aspects syntaxiques considérés, de sémantique définie pour l'outil, de
2 performances et en termes de vérification pouvant être effectuée) sera une perspective intéressante
3 que nous souhaitons effectuer.

4 **Équivalence de sémantiques** Nous souhaitons pour une perspective à long terme prouver
5 formellement l'équivalence entre le modèle SMD d'entrée et le modèle CPN de sortie; mais la
6 sémantique présentée par l'OMG des diagrammes états-transitions est semi-formelle. Cependant,
7 nous pouvons utiliser la sémantique opérationnelle des diagrammes états-transitions présentée
8 dans [?].

9 **Combinaison d'outils** Nous avons présenté dans ce travail de thèse deux outils, à savoir
10 *EasySM* qui implémente la méthode de spécification et UML2CPN qui implémente la traduc-
11 tion. Afin d'automatiser le processus de modélisation et de vérification (c'est-à-dire l'application
12 de la méthode de spécification ensuite la traduction vers les réseaux de Petri colorés), nous sou-
13 haitons créer un outil qui combine *EasySM* et UML2CPN. Le nouveau outil sera capable de
14 prendre en entrée un diagramme de classes, des observateurs d'états (avec leurs invariants), des
15 événements (avec leurs conditions/réactions) et donner directement (i) Un "Ok" dans le cas où
16 le système est vérifié correctement (ii) Un contre-exemple dans le cas où le système n'est pas
17 vérifié avec une possibilité de faire un raffinement automatique (ou semi-automatique) du dia-
18 gramme états-transitions en utilisant le contre-exemple. Par conséquent, avoir un outil unique
19 au fonctionnement transparent qui permettra à l'utilisateur de faire de la modélisation et de la
20 vérification.

¹ Annexe A

² Rapport engendré par l'outil *EasySM*

Project name: Library

1

Report EASY STATE MACHINE

Report generated by: Paladine, Thu Aug 20 16:00:19 CEST 2015

1. Input Class Diagram

1

[CL] Book

- + ATT: title:string,
- + ATT: author:string,

[CL] Copy

- + ATT: code:string,

[CL] User

- + ATT: name:string,
- + ATT: cardCode:string,

[CL] Date

[ASS] copyOf

- + CL Book
- + CL Copy

BRANCH NAME: Main

CONTEXT CLASS : Copy

2. Branch: Main, State Observers

1

[SO] inLib

Type: bool

Multiplicity: One

Invariant: $\text{not}(\text{inLib}) \Rightarrow (\text{not}(\text{onLoan}) \text{ and } (\text{not}(\text{booked}) \text{ and } ((\text{booker} = \text{Undef}) \text{ and } \text{not}(\text{tagged}))))$

[SO] onLoan

Type: bool

Multiplicity: One

Invariant: $\text{onLoan} \Rightarrow (\text{not}(\text{tagged}) \text{ and } \text{inLib})$

[SO] booked

Type: bool

Multiplicity: One

Invariant: $\text{booked} \Rightarrow ((\text{booker} \neq \text{Undef}) \text{ and } \text{not}(\text{tagged}))$

[SO] booker

Type: User

Multiplicity: FromZeroToOne

Invariant: $(\text{booker} = \text{Undef}) \Leftrightarrow \text{not}(\text{booked})$

[SO] tagged

Type: bool

Multiplicity: One

Invariant: $\text{tagged} \Rightarrow (\text{not}(\text{onLoan}) \text{ and } \text{not}(\text{booked}))$

[SO] final

Type: bool

Multiplicity: One

Invariant: $\text{not}(\text{final})$

3. Branch: Main, Events

1

[EV] created

+ cond: false

+ react: onLoan:=false;booked:=false;tagged:=false;inLib:=false;final:=false;

[EV] insertLib()

+ cond: not(inLib)

+ react: inLib:=true;

[EV] lend(U:User)

+ cond: inLib and (not(onLoan) and (not(tagged) and not(booked)))

+ react: onLoan:=true;

+ cond: inLib and (not(onLoan) and (not(tagged) and (booked and (booker = U))))

+ react: onLoan:=true;booked:=false;booker:=Undef;

[EV] bringBack()

+ cond: inLib and onLoan

+ react: onLoan:=false;

[EV] book(U:User)

+ cond: inLib and (onLoan and not(booked))

+ react: booked:=true;booker:=U;

[EV] putTag()

+ cond: inLib and (not(onLoan) and (not(tagged) and not(booked)))

+ react: tagged:=true;

[EV] removeTag()

+ cond: inLib and tagged

+ react: tagged:=false;

4. Branch: Main, State Table

inLib	onLoan	booked	tagged	final	name
true	true	true	false	false	State1
true	true	false	false	false	State2
true	false	true	false	false	State3
true	false	false	true	false	State4
true	false	false	false	false	State5
false	false	false	false	false	State6
true	true	true	true	true	Impossible (invariants)
true	true	true	true	false	Impossible (invariants)
true	true	true	false	true	Impossible (invariants)
true	true	false	true	true	Impossible (invariants)
true	true	false	true	false	Impossible (invariants)
true	true	false	false	true	Impossible (invariants)
true	false	true	true	true	Impossible (invariants)
true	false	true	true	false	Impossible (invariants)
true	false	true	false	true	Impossible (invariants)
true	false	false	true	true	Impossible (invariants)
true	false	false	false	true	Impossible (invariants)
false	true	true	true	true	Impossible (invariants)
false	true	true	true	false	Impossible (invariants)
false	true	true	false	true	Impossible (invariants)
false	true	true	false	false	Impossible (invariants)
false	true	false	true	true	Impossible (invariants)
false	true	false	true	false	Impossible (invariants)
false	true	false	false	true	Impossible (invariants)
false	true	false	false	false	Impossible (invariants)
false	false	true	true	true	Impossible (invariants)
false	false	true	true	false	Impossible (invariants)

false	false	true	false	true	Impossible (invariants)
false	false	true	false	false	Impossible (invariants)
false	false	false	true	true	Impossible (invariants)
false	false	false	true	false	Impossible (invariants)
false	false	false	false	true	Impossible (invariants)

Internal invariants:

State1:

- booker <> Undef
- booker <> Undef

State2:

- booker = Undef

State3:

- booker <> Undef
- booker <> Undef

State4:

- booker = Undef

State5:

- booker = Undef

State6:

- booker = Undef
- booker = Undef

Failed invariants:

Impossible (invariants) #1:

- onLoan => (not(tagged) and inLib)
- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #2:

- onLoan => (not(tagged) and inLib)

- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))

1

Impossible (invariants) #3:

- not(final)

Impossible (invariants) #4:

- onLoan => (not(tagged) and inLib)
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #5:

- onLoan => (not(tagged) and inLib)
- tagged => (not(onLoan) and not(booked))

Impossible (invariants) #6:

- not(final)

Impossible (invariants) #7:

- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #8:

- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))

Impossible (invariants) #9:

- not(final)

Impossible (invariants) #10:

- not(final)

Impossible (invariants) #11:

- not(final)

1

Impossible (invariants) #12:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #13:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))

Impossible (invariants) #14:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- not(final)

Impossible (invariants) #15:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)

Impossible (invariants) #16:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #17:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- tagged => (not(onLoan) and not(booked))

Impossible (invariants) #18:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)
- not(final)

Impossible (invariants) #19:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- onLoan => (not(tagged) and inLib)

Impossible (invariants) #20:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))
- not(final)

Impossible (invariants) #21:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- booked => ((booker <> Undef) and not(tagged))
- tagged => (not(onLoan) and not(booked))

Impossible (invariants) #22:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- not(final)

Impossible (invariants) #23:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))

Impossible (invariants) #24:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))
- not(final)

Impossible (invariants) #25:

- not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))

Impossible (invariants) #26:

- not(final)

1

StateMachine of Copy
Easy State Machine

Project: Library

Branch: Main

StateMachine of Copy

Built on: Thu Sep 08 19:49:07 CEST 2011

Building time: 15 milliseconds

State Machine:

Source	Guard	Event	Effect	Target
Initial				NotInLibrary
NotInLibrary		insertLib()		Available
Available		lend(U:User)		OnLoanNotBook ed
Booked	booker = U	lend(U:User)	booker:=Undef;	OnLoanNotBook ed
OnLoanBooked		bringBack()		Booked
OnLoanNotBook ed		bringBack()		Available
OnLoanNotBook ed		book(U:User)	booker:=U;	OnLoanBooked
Available		putTag()		Tagged
Tagged		removeTag()		Available

StateMachine Data

1 Initial state(s):

NotInLibrary

0 Unreachable state(s):

0 Final state(s) with outgoing transitions:

Control state(s) without outgoing transitions:

State Observer(s) not used in any Invariant:

0 State Observer(s) not used in any ConditionReaction:

0 Not used ConditionReaction(s):

Not used event(s):

5. Branch: Main, State Machine

6. Branch: Main, Output Class Diagram

[CL] Book

- + ATT: title:string,
- + ATT: author:string,

[CL] Copy

- + OP: insertLib(),
- + OP: lend(U:User),
- + OP: bringBack(),
- + OP: book(U:User),
- + OP: putTag(),
- + OP: removeTag(),

[CL] User

- + ATT: name:string,
- + ATT: cardCode:string,

[CL] Date

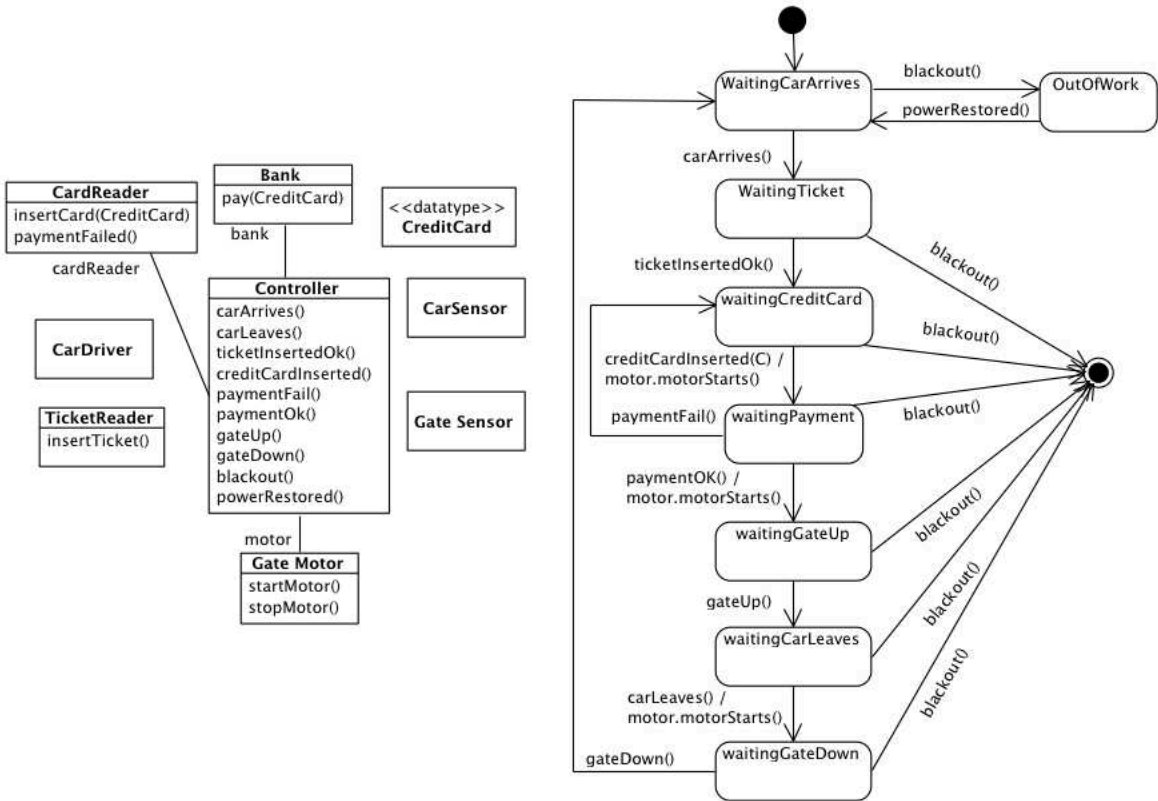
[ASS] booker

- + CL: Copy
- + CL: User

1 Questionnaires de l'expérience des métriques

Controller- Répondre aux questions ci-dessous relatives au diagramme états-transitions ci-dessous qui modélise le comportement des instances de la classe Controller décrite dans le diagrammes de classes à gauche.

Heure début : _____



C1) Est-il possible que le contrôleur atteigne l'état WaitingCarArrives après avoir été informé que le paiement est ok (PaymentOK) ?

OUI NON

C2) Est-il possible que le courant soit rétabli (powerRestored()) après une coupure de courant (blackout()) et que le contrôleur fonctionne de nouveau ?

OUI NON

C3) Que se passera-t-il si une coupure de courant intervient alors qu'une voiture attend que la barrière se lève (waitingGateUp) ?

La voiture restera là pour toujours parce que la barrière ne se lèvera jamais

On attend que le courant revienne et le conducteur insère à nouveau sa carte de crédit

C4) Que se passera-t-il si le paiement avec la carte de crédit ne réussit jamais ?

La voiture ne passera jamais la barrière

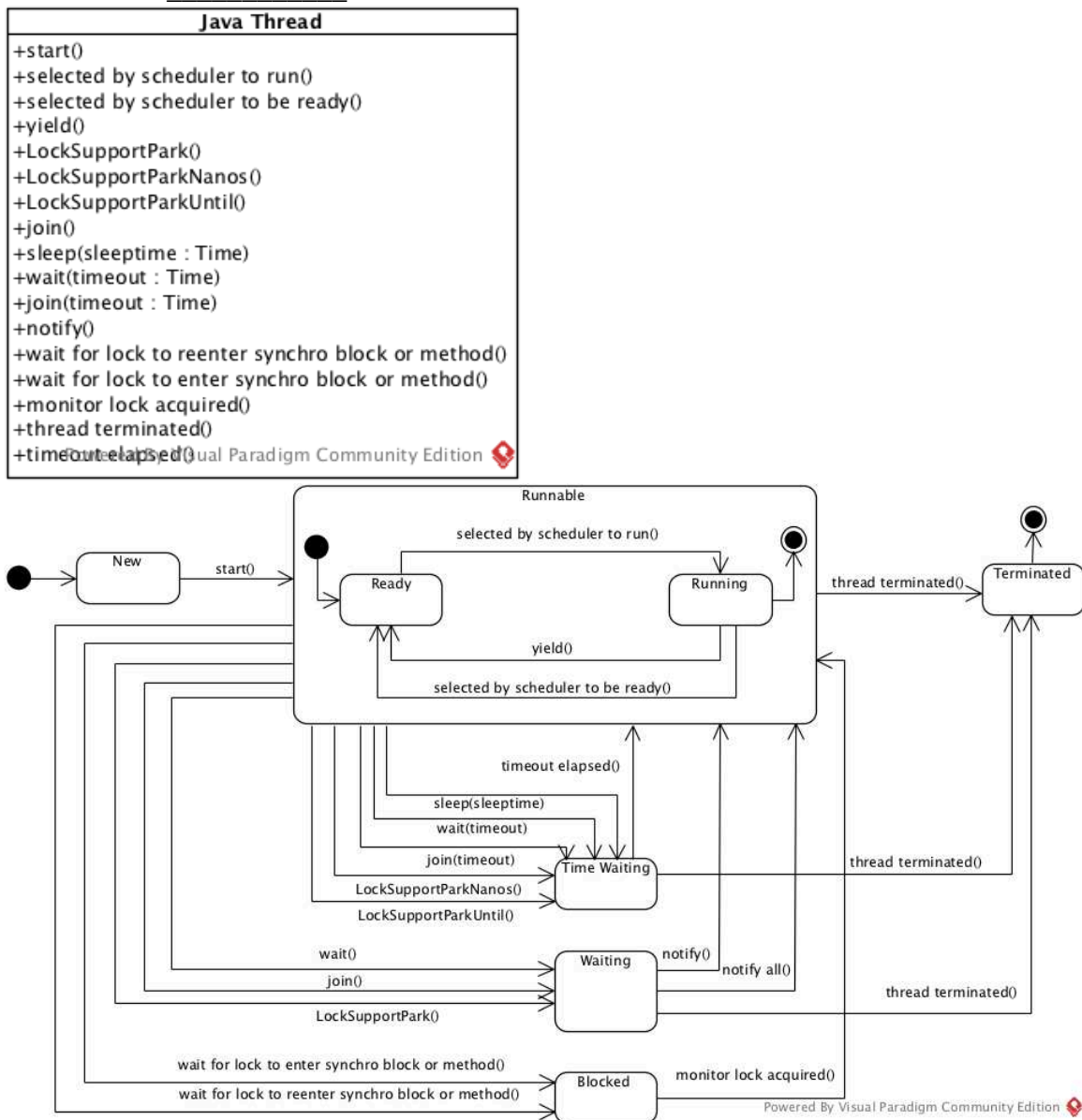
Le contrôleur tombera en panne

Une coupure de courant finira par se produire

Heure fin : _____

Thread - Répondre aux questions ci-dessous relatives au diagramme états-transitions ci-dessous qui modélise le comportement des instances de la classe Java Thread décrite dans le diagrammes de classes ci-dessous.

Heure début : _____



T1) Un thread qui est prêt (ready) peut être stoppé (terminated) ? OUI NON

T2) Un thread réagira à un même événement de la même manière après avoir reçu un événement `wait()` et un événement `wait(timeout)`? OUI NON

T3) Parmi ces séquences d'événements lesquelles vont conduire un thread qui est tout juste créé (New) à atteindre l'état Running?

`start() ; wait(timeout) ; timeout elapsed()` OUI NON

`start() ; wait() ; notify() ; selected by scheduler to run()` OUI NON

`start() ; join() ; notify all() ; selected by scheduler to run()` OUI NON

`start() ; join(timeout) ; notify all() ; selected by scheduler to run()` OUI NON

T4) Est-il vrai qu'un thread qui vient d'être créé et reçoit trois fois de suite l'événement `start()` sera dans l'état Ready? OUI NON

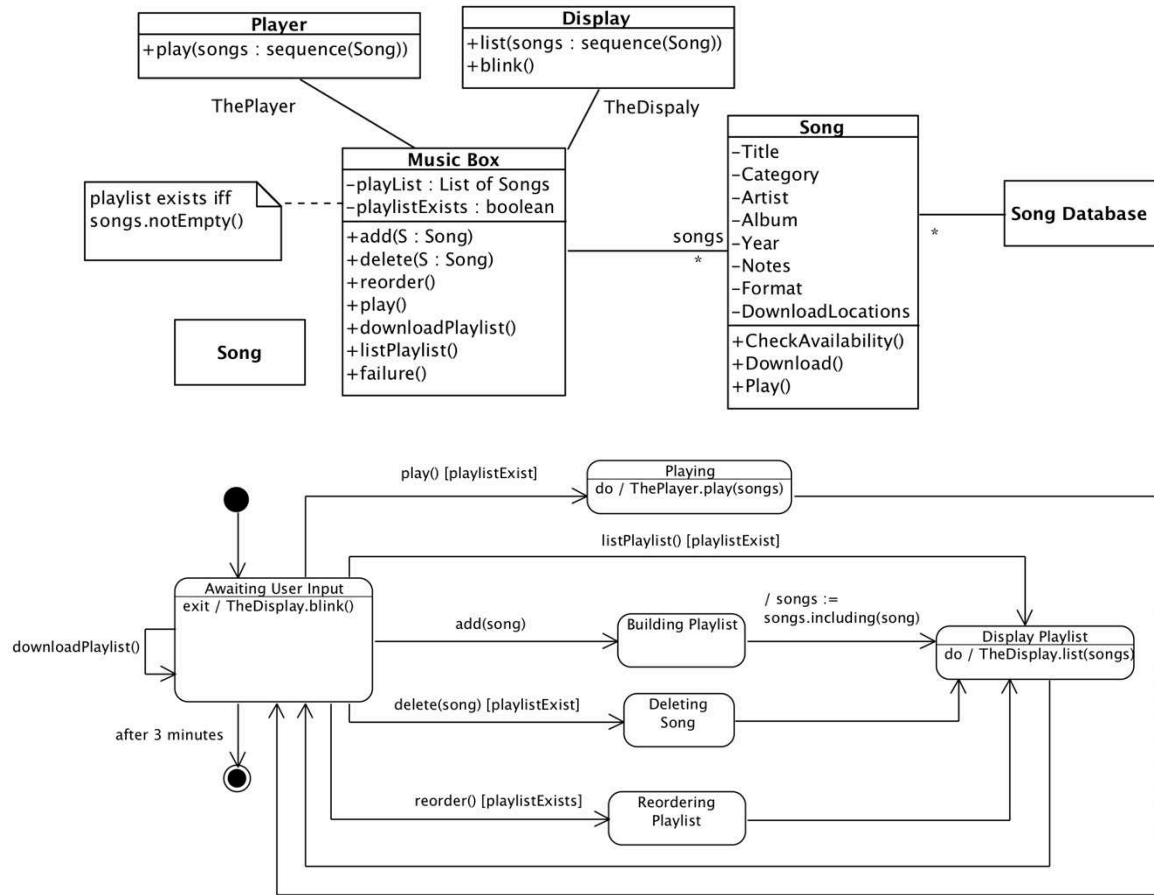
Heure fin : _____

Music Box- Répondre aux questions ci-dessous relatives au diagramme états-transitions

ci-dessous qui modélise le comportement des instances de la classe MusicBox décrite dans le diagramme de classes ci-dessous.

Heure début : _____

1



M1) Après lequel des événements ci-dessous la liste des morceaux sera-t-elle immédiatement affichée (display) ?

downloadPlaylist() OUI NON

play() OUI NON

listPlaylist() OUI NON

add(song) OUI NON

delete(song) OUI NON

reorder() OUI NON

M2) Sera-t-il possible d'ajouter (add) un nouveau morceau immédiatement après qu'un morceau ait été supprimé (delete) ou ajouté (add) ?

OUI NON

M3) Est-il possible d'effectuer un chargement (download) même si la liste des morceaux (Playlist) n'existe pas ?

OUI NON

M4) Que se passera-t-il si, alors qu'une instance de Music Box est dans l'état AwaitingUserInput, elle reçoit l'événement correspondant à la suppression (delete) d'un même morceau 2000 fois ?

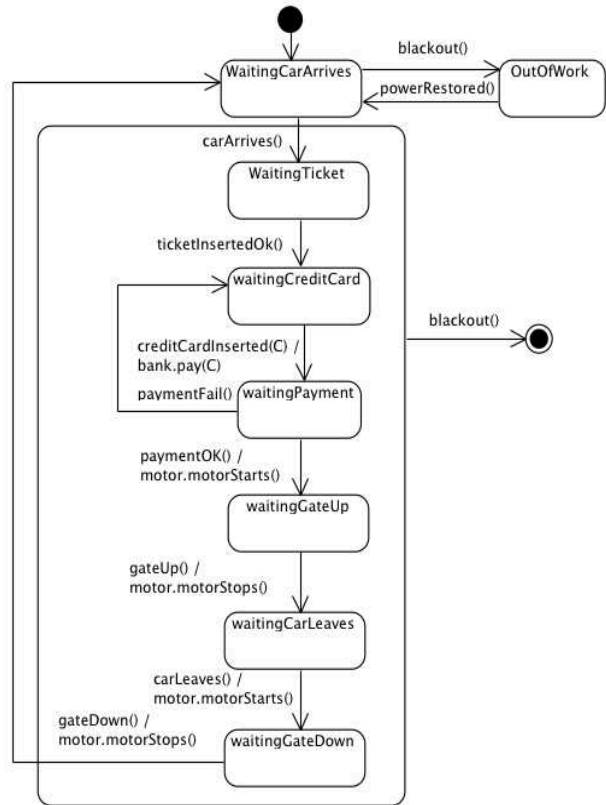
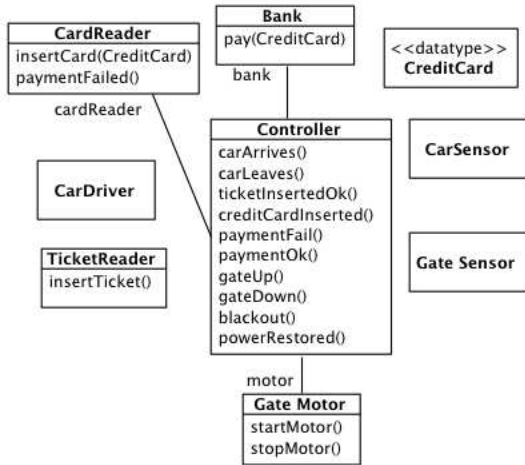
Rien ne change dans l'instance de Music Box

L'instance de Music Box réagit seulement au premier événement, et les 1999 suivants sont ignorés

Heure fin : _____

Controller- Répondre aux questions ci-dessous relatives au diagramme états-transitions ci-dessous qui modélise le comportement des instances de la classe Controller décrite dans le diagrammes de classes à gauche.

Heure début : _____



C1) Est-il possible que le contrôleur atteigne l'état WaitingCarArrives après avoir été informé que le paiement est ok (PaymentOK) ?

OUI NON

C2) Est-il possible que le courant soit rétabli (powerRestored()) après une coupure de courant (blackout()) et que le contrôleur fonctionne de nouveau ?

OUI NON

C3) Que se passera-t-il si une coupure de courant intervient alors qu'une voiture attend que la barrière se lève (waitingGateUp) ?

La voiture restera là pour toujours parce que la barrière ne se lèvera jamais

On attend que le courant revienne et le conducteur insère à nouveau sa carte de crédit

C4) Que se passera-t-il si le paiement avec la carte de crédit ne réussit jamais ?

La voiture ne passera jamais la barrière

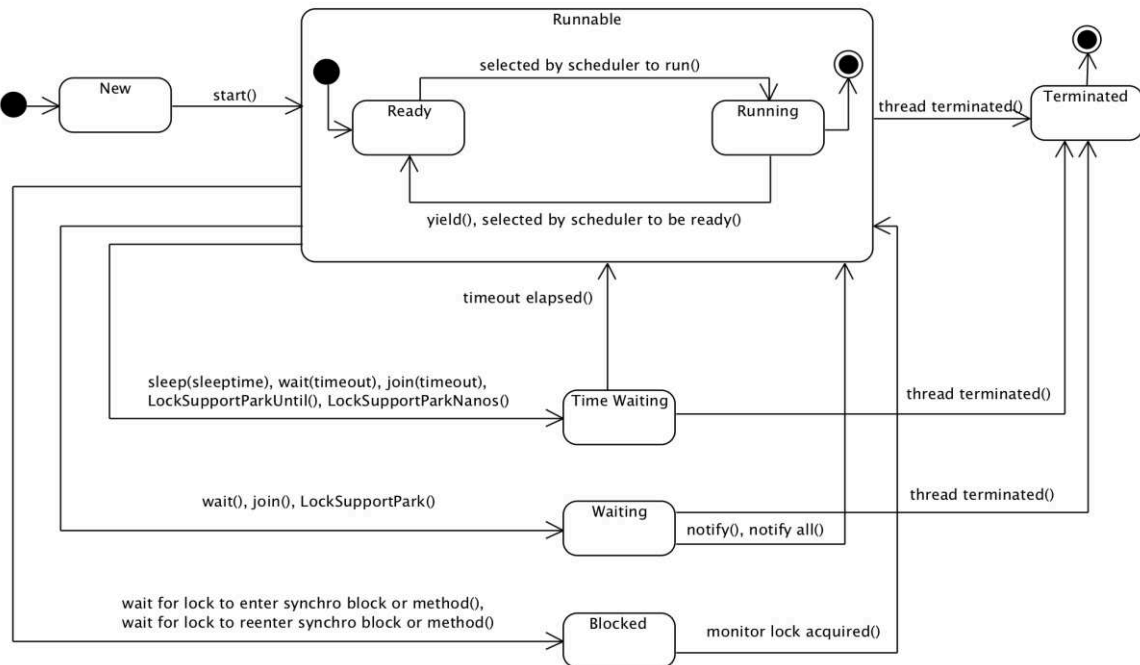
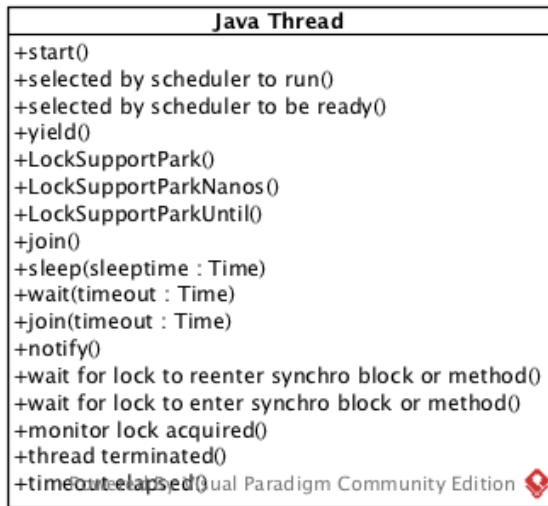
Le contrôleur tombera en panne

Une coupure de courant finira par se produire

Heure fin : _____

Thread - Répondre aux questions ci-dessous relatives au diagramme états-transitions ci-dessous qui modélise le comportement des instances de la classe Java Thread décrite dans le diagramme de classes ci-dessous.

Heure début : _____



T1) Un thread qui est prêt (ready) peut être stoppé (terminated) ? OUI NON

T2) Un thread réagira à un même événement de la même manière après avoir reçu un événement wait() et un événement wait(timeout)? OUI NON

T3) Parmi ces séquences d'événements lesquelles vont conduire un thread qui est tout juste créé (New) à atteindre l'état Running?

start() ; wait(timeout) ; timeout elapsed() OUI NON

start() ; wait() ; notify(); selected by scheduler to run() OUI NON

start() ; join() ; notify all(); selected by scheduler to run() OUI NON

start() ; join(timeout) ; notify all(); selected by scheduler to run() OUI NON

T4) Est-il vrai qu'un thread qui vient d'être créé et reçoit trois fois de suite l'événement start() sera dans l'état Ready? OUI NON

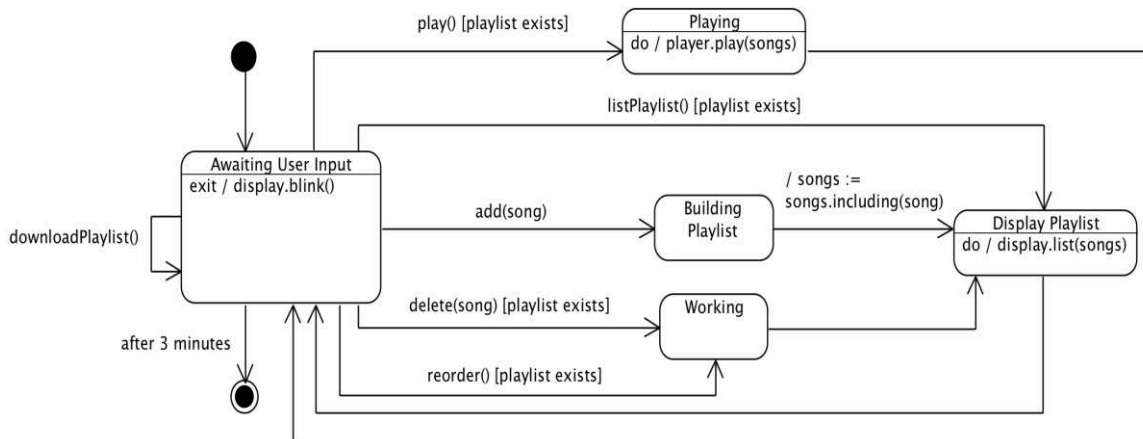
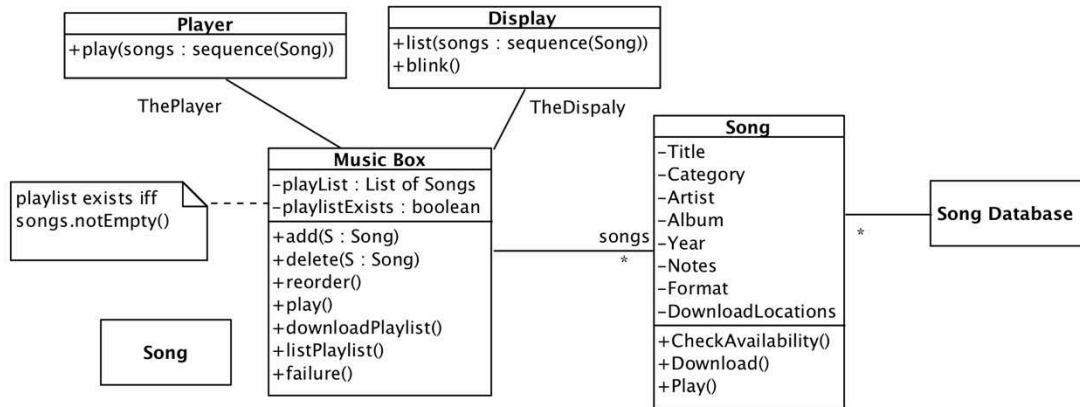
Heure fin : _____

Music Box- Répondre aux questions ci-dessous relatives au diagramme états-transitions

ci-dessous qui modélise le comportement des instances de la classe MusicBox décrite dans le diagrammes de classes ci-dessous.

Heure début : _____

1



M1) Après lequel des événements ci-dessous la liste des morceaux sera-t-elle immédiatement affichée (display) ?

downloadPlaylist() OUI NON

play() OUI NON

listPlaylist() OUI NON

add(song) OUI NON

delete(song) OUI NON

reorder() OUI NON

M2) Sera-t-il possible d'ajouter (add) un nouveau morceau immédiatement après qu'un morceau ait été supprimé (delete) ou ajouté (add) ?

OUI NON

M3) Est-il possible d'effectuer un chargement (download) même si la liste des morceaux (Playlist) n'existe pas ?

OUI NON

M4) Que se passera-t-il si, alors qu'une instance de Music Box est dans l'état AwaitingUserInput, elle reçoit l'événement correspondant à la suppression (delete) d'un même morceau 2000 fois ?

Rien ne change dans l'instance de Music Box

L'instance de Music Box réagit seulement au premier événement, et les 1999 suivants sont ignorés

Heure fin : _____

Exemples de l'expérience sur les métriques

Afin de présenter l'importance des critères de qualité dans la compréhension du diagramme états-transitions, une expérimentation a été élaborée (Section 4.8) en utilisant divers exemples. Nous présentons ici les exemples non utilisés dans l'expérimentation et qui seront utilisés dans le cadre d'autres expérimentations.

diagramme article [?] (-)	inscription séminaire [?] (-)	cycle de vie d'objet [?] (-)
diagramme article [?] (+)	inscription séminaire [?] (+)	cycle de vie d'objet [?] (+)

TABLE A.1 – Exemples utilisés dans l'expérimentation

Exemple simple Exemple utilisé pour l'étude du critère "de la suppression des états dupliqués" dans un diagramme états-transitions. La Figure A.1 représente un exemple de diagramme états-transitions avec des états dupliqués (**after+** et **after-**). La Figure A.2 représente le même exemple simple mais sans présence d'états dupliqués (les deux états **after-** et **after+** sont remplacés par **toSet**).

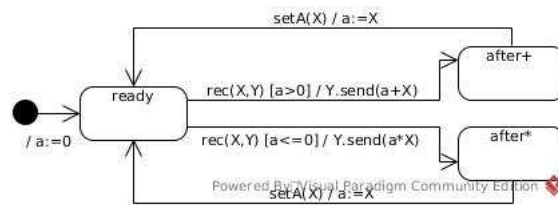


FIGURE A.1 – Exemple de l'article avec des états dupliqués

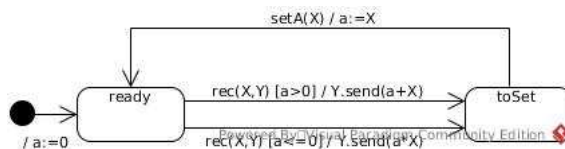


FIGURE A.2 – Exemple de l'article sans états dupliqués

- 1 **Exemple de l'inscription** Exemple utilisé pour l'étude du critère "de l'introduction de la
 2 hiérarchie et des états composites" dans un diagramme états-transitions. La Figure A.3 repré-
 3 sente le diagramme états-transition de l'inscription à un séminaire avec la présence de plusieurs
 4 transitions étiquetées de l'événement **cancelled** sortant de différents états et allant vers le même
 5 état (ici l'état final). La Figure A.4 représente le même diagramme mais avec l'introduction d'un
 6 état composite (**Ongoingregistration**) et la suppression des transitions étiquetées avec l'évé-
 7 nement **cancelled** et en les remplaçant par une transition (avec le même événement) allant de
 8 l'état composite **Ongoingregistration** vers l'état final.

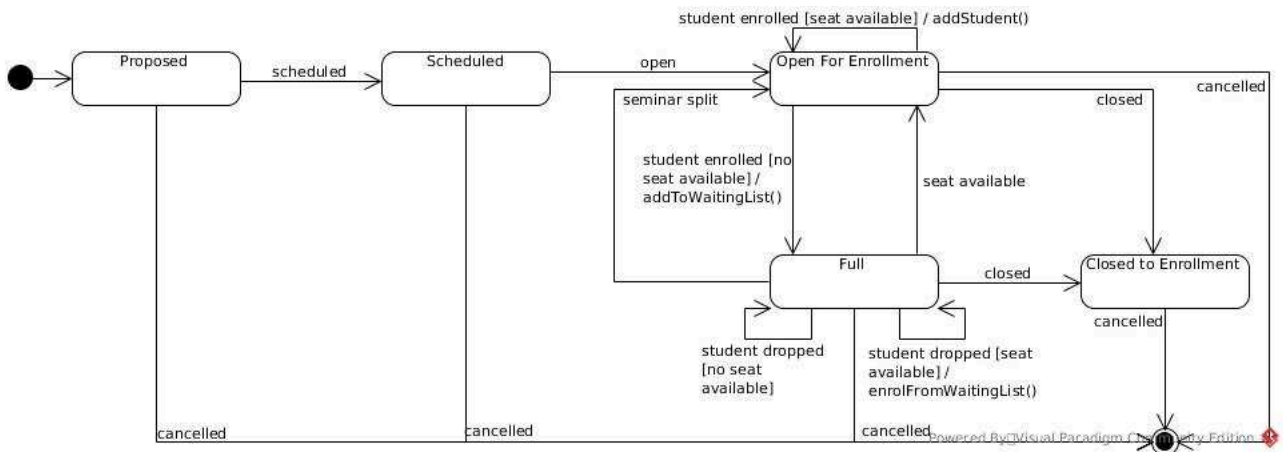


FIGURE A.3 – Exemple de l'inscription dans un séminaire sans hiérarchie

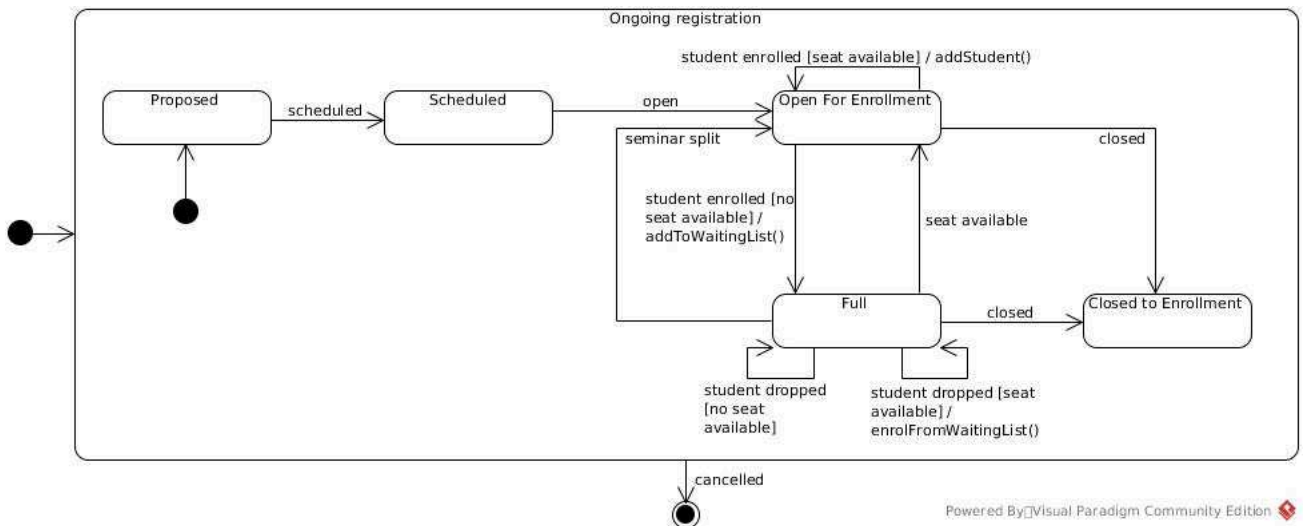


FIGURE A.4 – Exemple de l'inscription dans un séminaire avec hiérarchie

- 1 **Exemple du cycle de vie d'un objet** Exemple utilisé pour l'étude du critère "de l'uti-
 2 lisation événements multiples" dans un diagramme états-transitions. La Figure A.5 représente
 3 le diagramme états-transitions de la vie d'un objet avec la présence de plusieurs transitions sor-
 4 tant du même état, allant vers le même état mais avec des événements différents (*e.g.* de l'état
 5 **ExistsandReferenced** vers **ExistsandNotReferenced**). La Figure A.6 représente le même dia-
 6 gramme mais avec la factorisation des transitions sortant d'un seul état et allant vers le même
 7 état (*e.g.* la transition de l'état **DoesnotExistsandreferenced** vers l'état final).

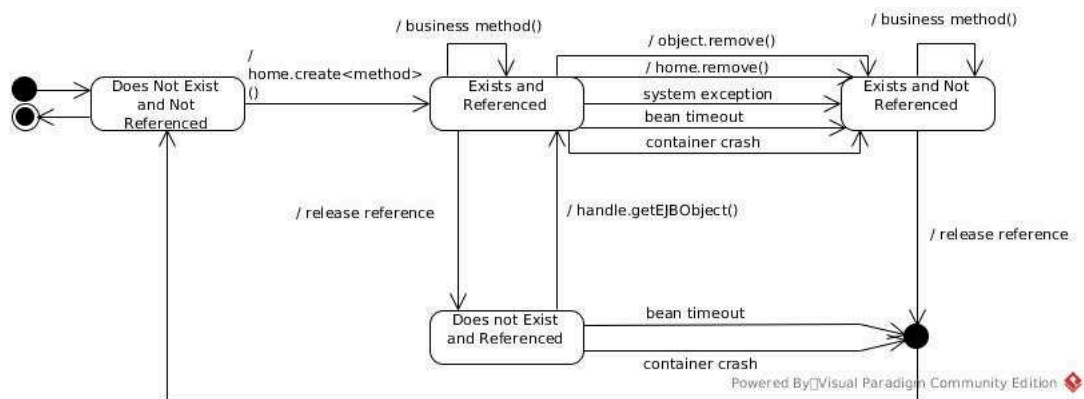


FIGURE A.5 – Exemple de la vie d'un objet avec des événements dans des transitions différentes

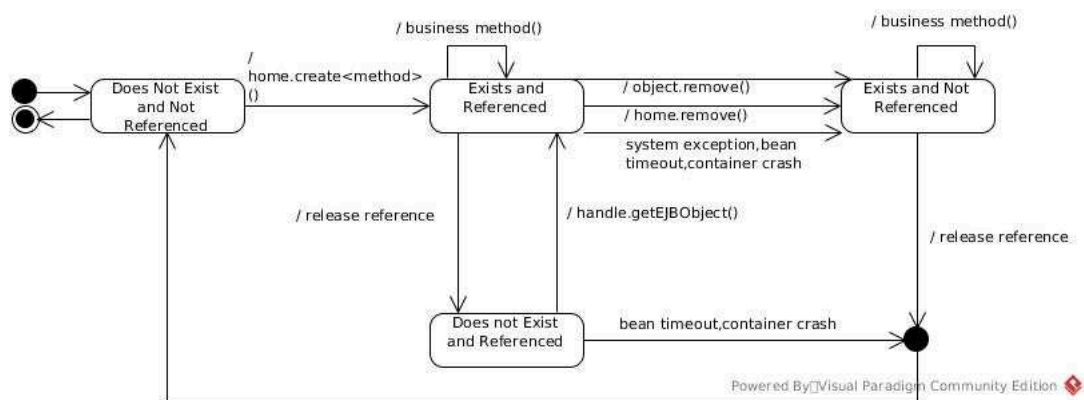


FIGURE A.6 – Exemple de la vie d'un objet avec des événements dans une seule transition