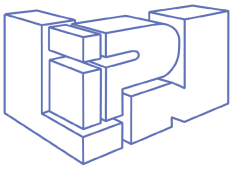
UNIVERSITÉ PARIS 13



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, Examinateur, 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. Quoi que 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 cité, mais qui d'une façon ou d'une autre m'ont aidé pendant ce parcours.

Table des matières

List of Algorithms

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Contents

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

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

Objectif de la thèse

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

Contributions

Contribution 1

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

Contribution 2

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

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

Contribution 3

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

Organisation du document

Dans ce document nous présentons dans le Chapitre 2 les concepts de base utilisés dans nos contributions à savoir les diagrammes états-transitions et les réseaux de Petri colorés. Le Chapitre 3 comportera les différents travaux de l'état de l'art concernant aspects que notre thèse traite, tels que la vérification des diagrammes états-transitions, les méthode de spécifications et l'intégration des contraintes temporelles dans les diagrammes états-transitions. Dans le Chapitre 4, nous présentons la méthode de spécification que nous utilisons et les différentes améliorations que nous apportons à cette dernière. Nous présentons également l'outil *EasySM* (qui implémente la méthode de spécification) avec les différentes améliorations apportées à ce dernier. Dans le chapitre suivant (Chapitre 5), nous présentons notre principale contribution concernant la traduction des diagrammes états-transitions vers les réseaux de Petri colorés. Ensuite nous présentons dans le Chapitre 6 une étude de cas avec son diagramme états-transitions correspondant et le réseau de Petri coloré résultat de l'application de notre traduction. La suite du document comportera le Chapitre 7 sur l'implémentation de notre traduction et le Chapitre 8 sur l'intégration des contraintes temporelles dans nos approches. Enfin, nous présentons une conclusion et des perspectives dans le Chapitre 9.

Chapitre 2

Concepts de base

Contents

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.

Diagrammes états-transitions UML (*SMD*)

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

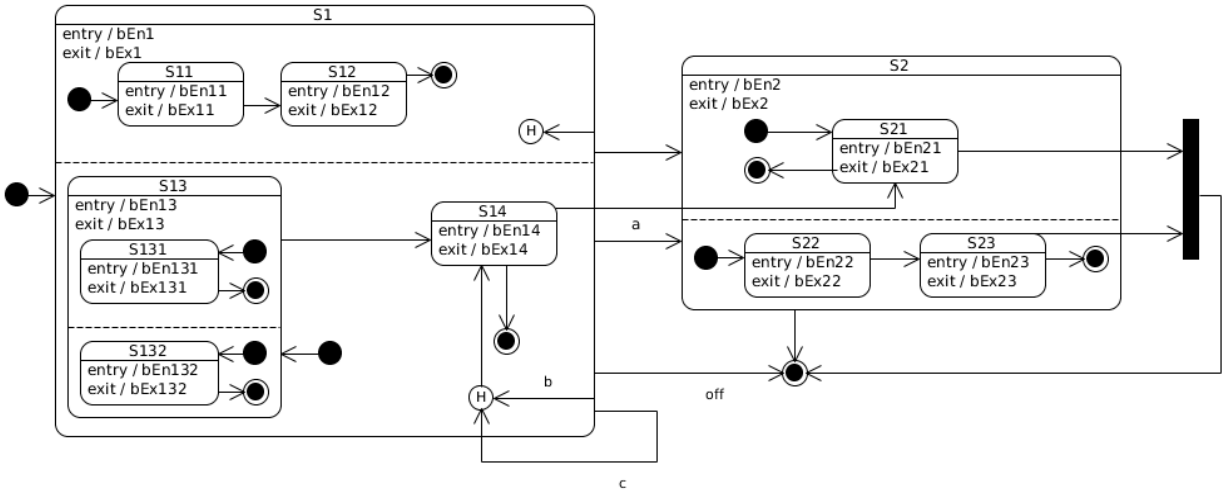


FIGURE 2.1 – Exemple d'un diagramme états-transitions

Diagrammes états-transitions : Rappel

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

États

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

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

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

- Ses *sous-états directs* sont l'ensemble des états immédiatement contenus dans cette région (par exemple, dans la Figure 2.1 les sous-états directs de la région inférieure de l'état `S1` sont `S13`, `S14` et l'état final de la région).
- Ses *sous-états indirects* sont des états obtenus par la fermeture transitive de la relation des sous-états (par exemple, dans la Figure 2.1 les sous-états indirects de la région inférieure de l'état `S1` sont `S13`, `S131`, `S132`, `S14` et les trois états finaux).

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

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

États finaux

Un état final (par exemple, l'état le plus à droite dans l'état `S2` de la Figure 2.1) est un état particulier dans lequel on entre lorsque l'exécution dans la région le contenant est terminée.

Comportements

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

Pseudo-états

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

Initial Un pseudo-état initial est le point de départ d'une région d'un état composite ou d'une machine à états. Chaque état composite (simple ou orthogonal) et chaque machine à états peut avoir un pseudo-état initial. Dans le cas où il n'y a pas de pseudo-état initial alors une possibilité est de considérer que le modèle (le diagramme états-transitions) est mal défini ("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` $\neq$ `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.

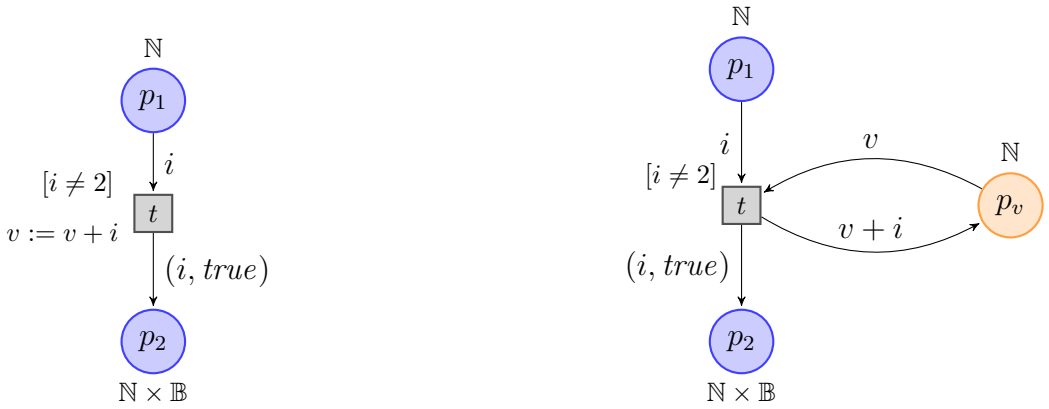
L'ordre d'exécution des comportements d'entrée pendant l'entrée dans un état à travers une transition est le suivant : si la destination de la transition est un état simple alors on exécute le comportement d'entrée de cet état. Si l'état destination est un état composite avec une seule région alors on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

Run-to-completion step

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



(a) Notations des variables globales

(b) La sémantique correspondante

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.

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

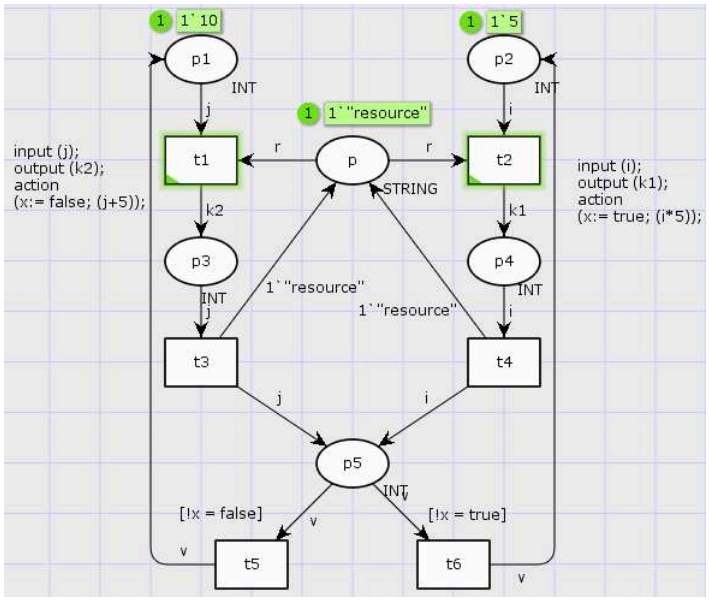


FIGURE 2.3 – Exemple de réseau de Petri coloré modélisé en utilisant CPN Tools

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

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

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

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

Marquage et marquage initial d'un *CPN*

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

- $\mathsf{Mark} = (\mathsf{nb\_1`value\_1} + + \mathsf{nb\_2`value\_2} + + ...\ \mathsf{nb\_k`value\_k})$.
- Chaque place peut contenir un ensemble de couleurs possibles de jetons. Pour un ensemble de jetons d'une couleur donnée on a la notation suivante : $(\mathsf{nb\_i`value\_i})$, où $\mathsf{nb\_i}$ est le nombre de jetons ayant la valeur $\mathsf{value\_i}$. En d'autres termes, une place peut contenir un nombre $\mathsf{nb\_1}$ de jetons ayant la valeur $\mathsf{value\_1}$, un nombre $\mathsf{nb\_2}$ de jetons ayant la valeur $\mathsf{value\_2}$, etc.

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

Franchissement des transitions et consommation/production des jetons

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

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

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

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

Exemple d'évolution du marquage d'un *CPN*

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

Afin de faire évoluer le marquage du réseau, la transition t1 sera franchie ; le résultat du franchissement de cette transition est montré dans la Figure 2.4. Nous remarquons que le marquage

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.

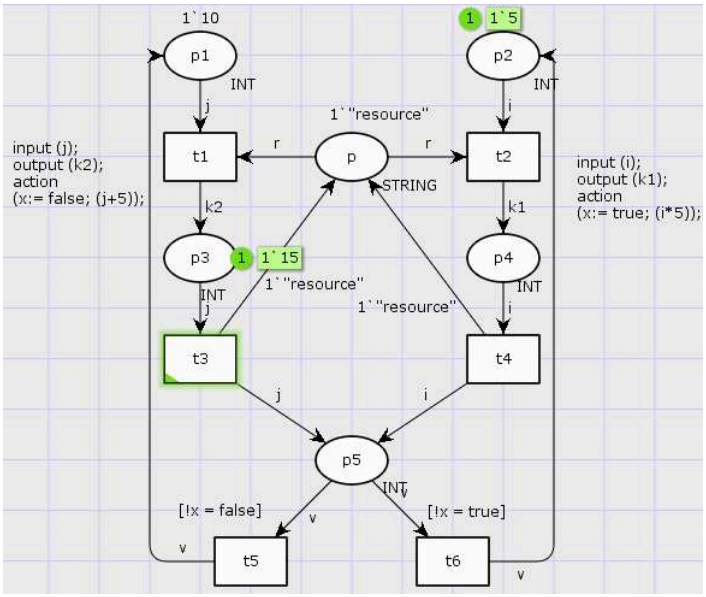


FIGURE 2.4 – Évolution du marquage : Franchissement de la transition t1

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.

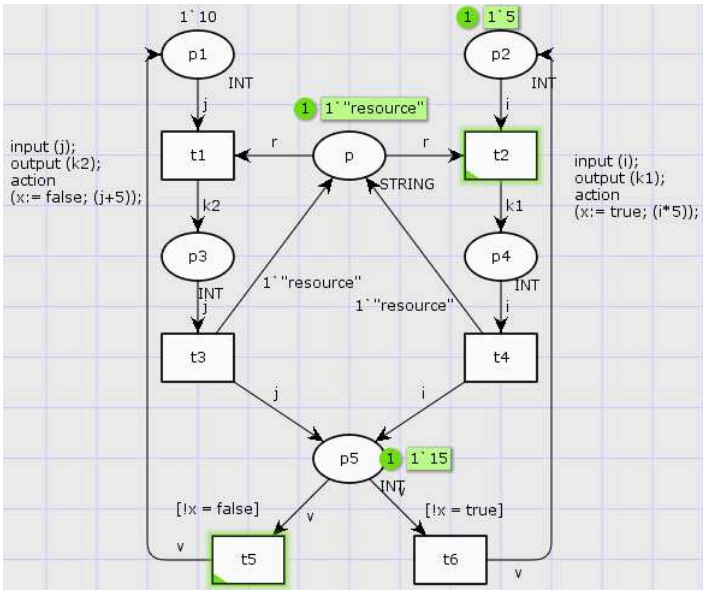


FIGURE 2.5 – Évolution du marquage : Franchissement de la transition t3

CPNTools

CPN Tools 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

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

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

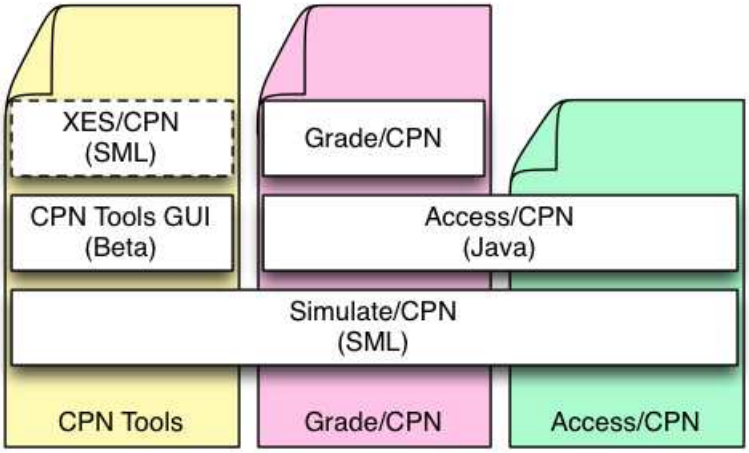


FIGURE 2.6 – L'architecture de CPN Tools

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

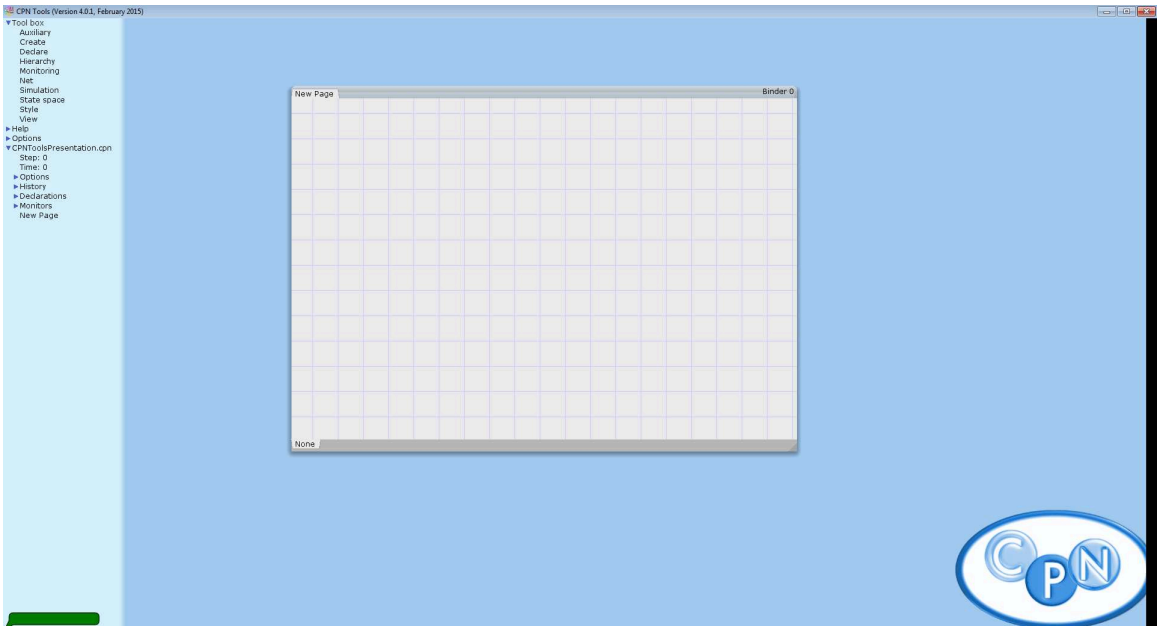


FIGURE 2.7 – Interface Standard de CPN tools

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

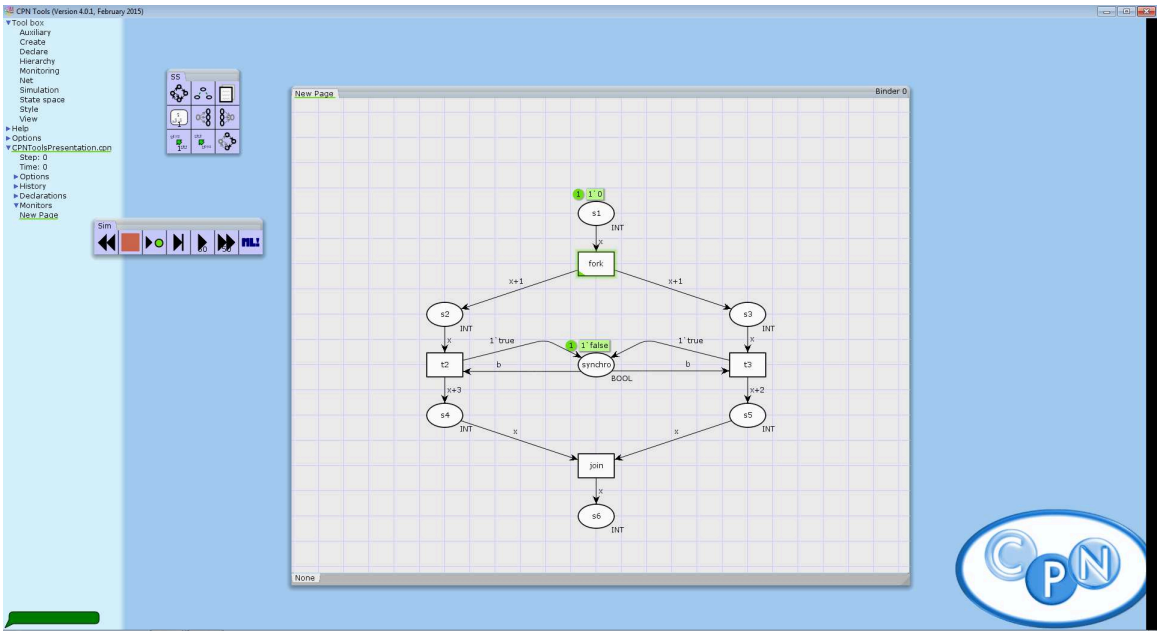


FIGURE 2.8 – Exemple dans CPN tools

Conclusion

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

Chapitre 3

État de l'art

Contents

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Autres approches de traductions Une approche proposée dans [?] qui permet de représenter les statechars en utilisant des structures de transitions. Cette approche prend en compte une version simplifiée des statecharts et ne considère pas les variables.

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

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

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

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

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

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

UML et temps

UML profile

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

La Figure 3.1 représente le méta-modèle utilisé pour la représentation du temps et des durées dans UML.

La syntaxe et la sémantique utilisées pour la représentation du temps dans UML sont présentées sous forme de grammaire comme le montre la Figure 3.2. Les deux éléments syntaxiques utilisés pour l'expression du temps dans UML sont `after` et `at`.

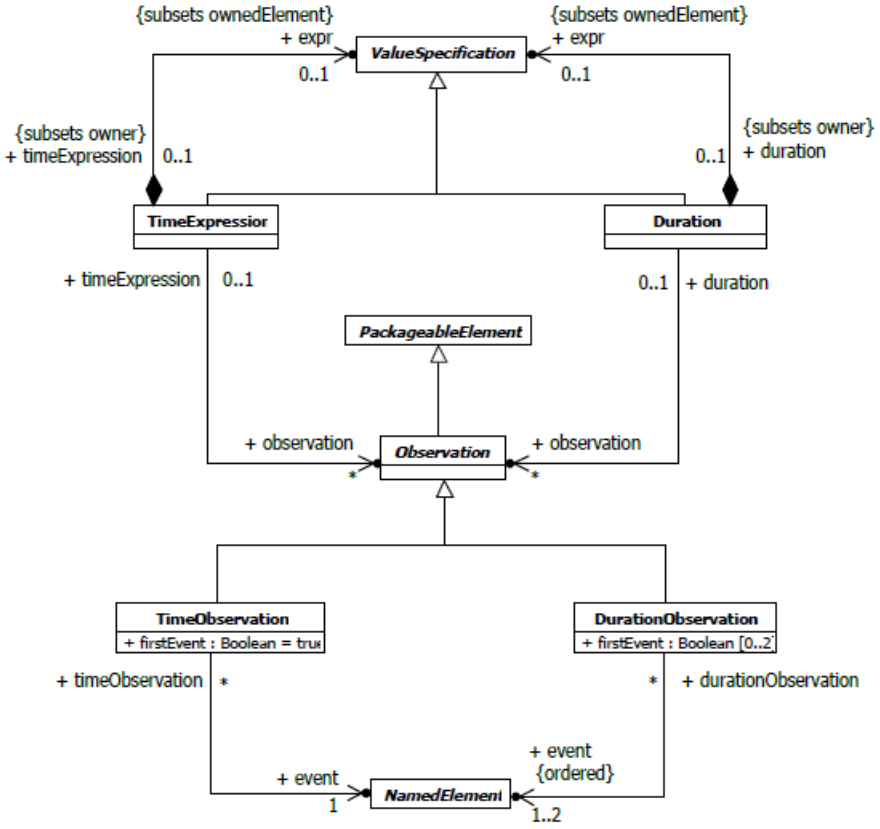


FIGURE 3.1 – Le temps et les durées dans UML

```
Grammaire
<time-event> ::= <relative-time-event> | <absolute-time-event>
<relative-time-event> ::= 'after' <time-expression>
<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.

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

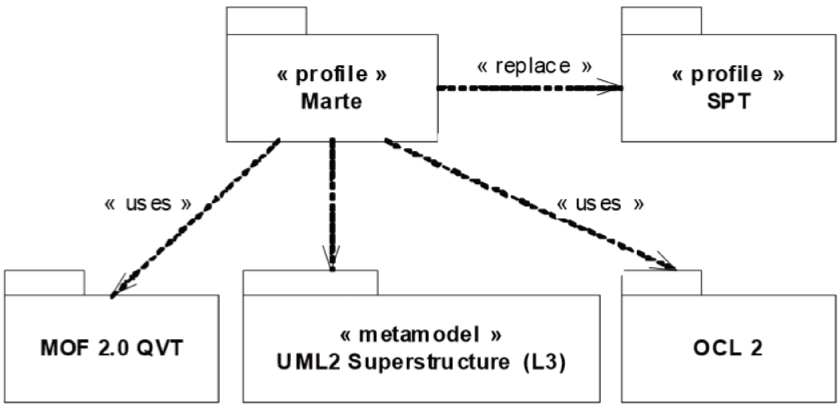


FIGURE 3.3 – MARTE et les standards de l'OMG

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

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

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

La Figure 3.4 montre l'architecture du profil MARTE.

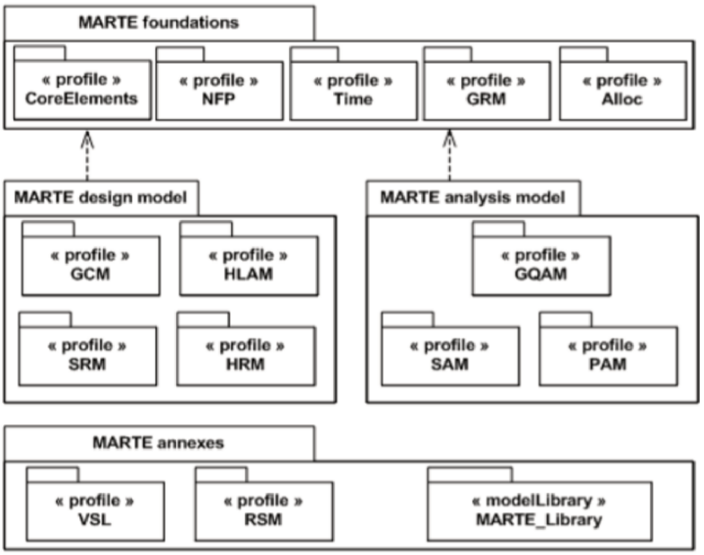


FIGURE 3.4 – L'architecture de MARTE.

Modélisation du temps dans MARTE

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

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

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

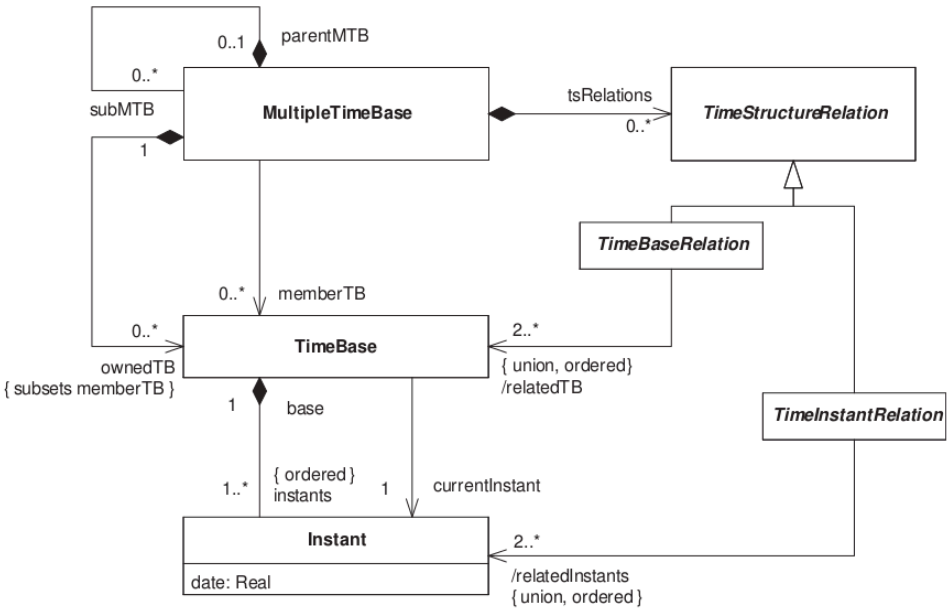


FIGURE 3.5 – La structure du temps dans MARTE ([?]).

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

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

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

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

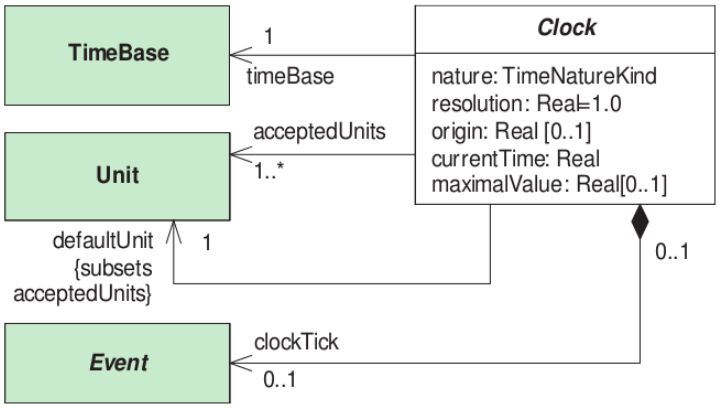


FIGURE 3.6 – L'horloge dans MARTE ([?]).

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

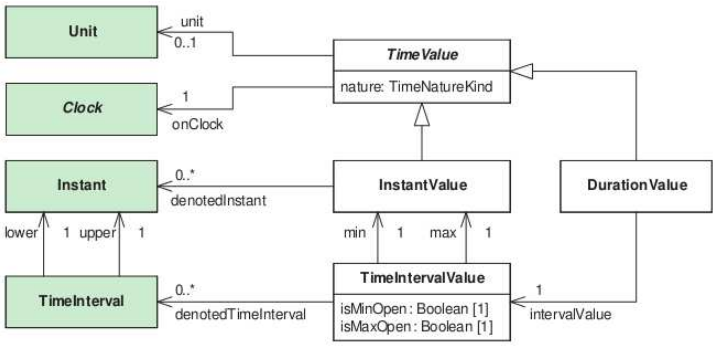


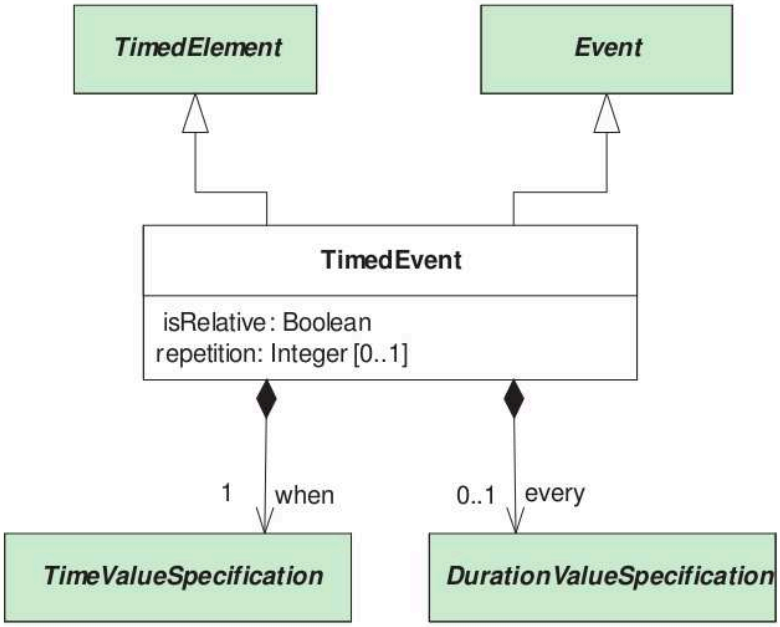
FIGURE 3.7 – TimeValue dans MARTE ([?]).

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

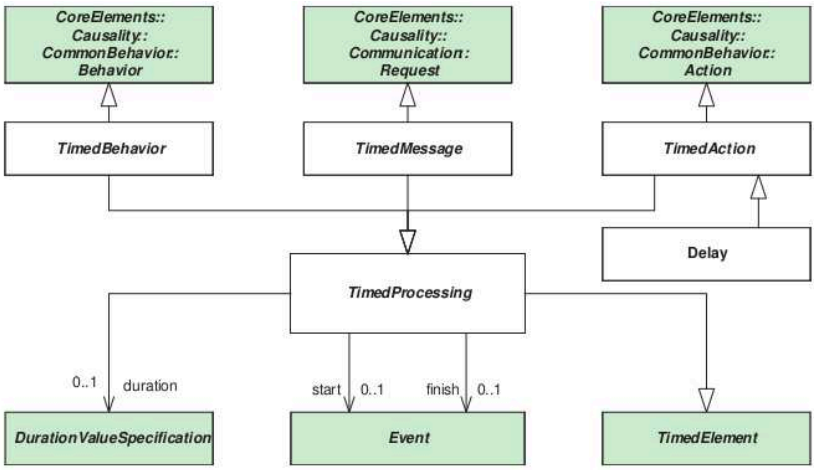
Les différents concepts décrits ci-dessus sont représentés dans MARTE par un ensemble de *stéréotypes*. Chaque *stéréotype* va permettre l'utilisation des différents concepts dans la modélisation du temps mais aussi de permettre à l'utilisateur de définir ses propres horloges en utilisant la bibliothèque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

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

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



(a) TimedEvent dans MARTE



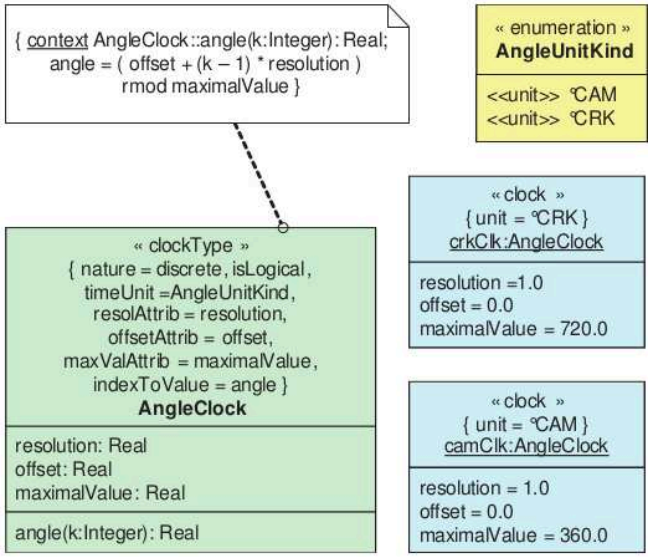
(b) TimedProcessing dans MARTE

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

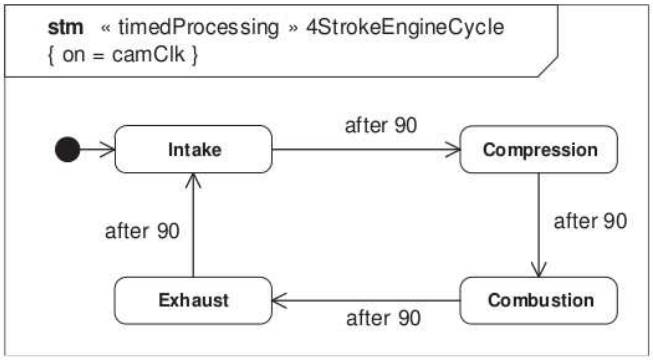
camClk et *crkClk* sont des instances de l'horloge principale avec des valeurs pour les attributs de bases.

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

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

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

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

Diagrammes états transitions UML avec le temps

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

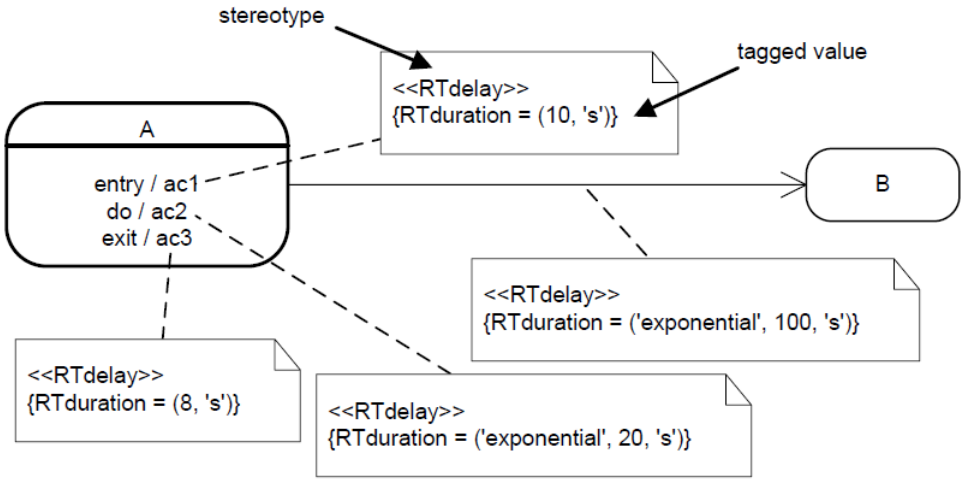


FIGURE 3.10 – Diagramme états transitions UML avec le temps

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

Réseaux de Petri stochastiques (SPN)

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

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

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

Transformation

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

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

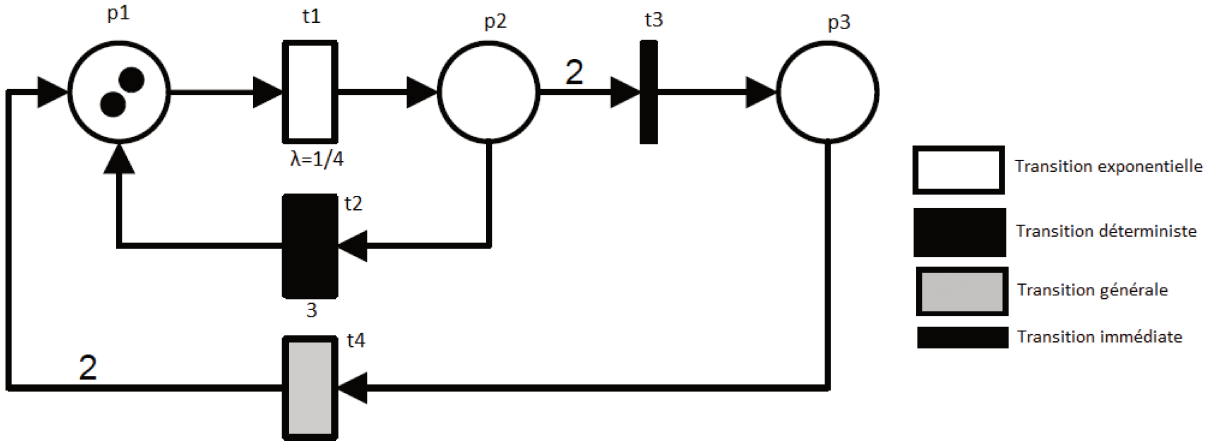


FIGURE 3.11 – Réseaux de Petri stochastiques

une transition (`t_do_A`), le comportement d'entrée par une transition (`t_ent_A`) et le comportement 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.

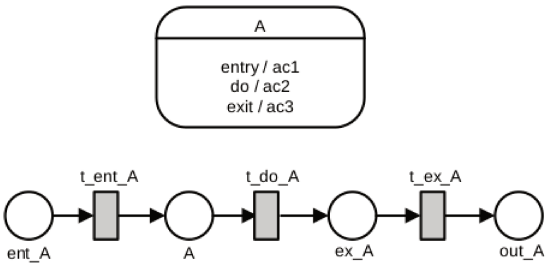


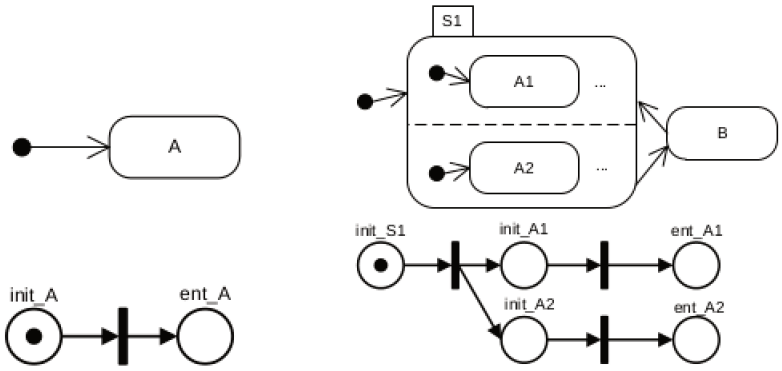
FIGURE 3.12 – Traduction d'un état

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

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

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

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

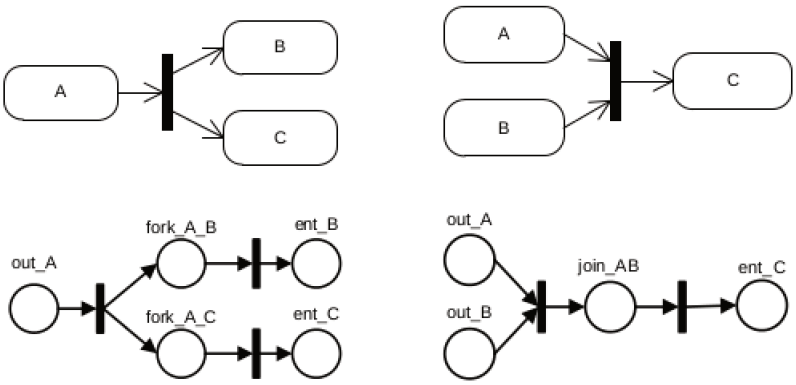


(a) Pseudo-état initial avec état simple

(b) Pseudo-état initial avec état orthogonal

FIGURE 3.13 – Traduction d'un pseudo-état initial

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 Join est représenté par une place et deux transitions. La Figure 3.14 montre la traduction d'un exemple contenant les pseudo-états Fork et Join.



(a) Pseudo-état Fork

(b) Pseudo-état Join

FIGURE 3.14 – Traduction des pseudo-états Fork et Join

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

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

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

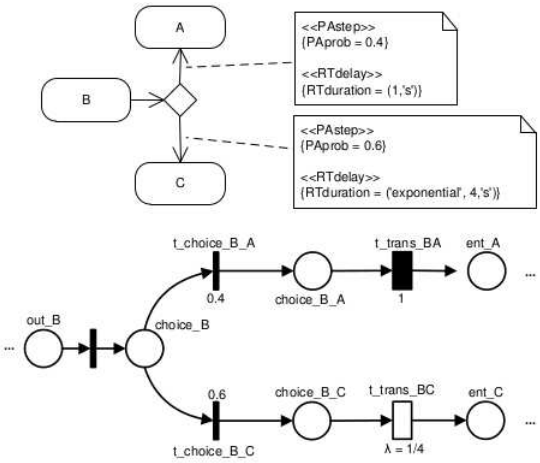


FIGURE 3.15 – Traduction d'un pseudo-état choice

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

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

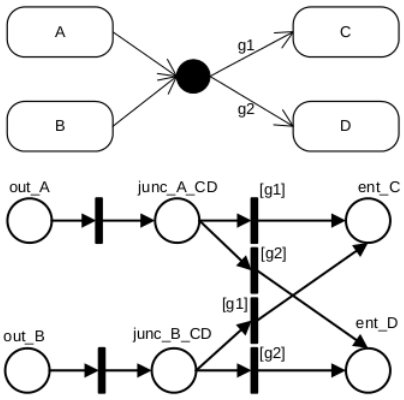


FIGURE 3.16 – Traduction d'un point de jonction

Notion du temps Le temps dans une transition UML définit le type de transition correspondante en réseaux de Petri stochastiques :

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

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

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

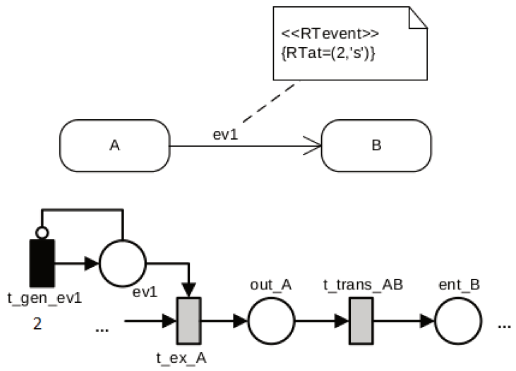


FIGURE 3.17 – Traduction d'une transition avec événement

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

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

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

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

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

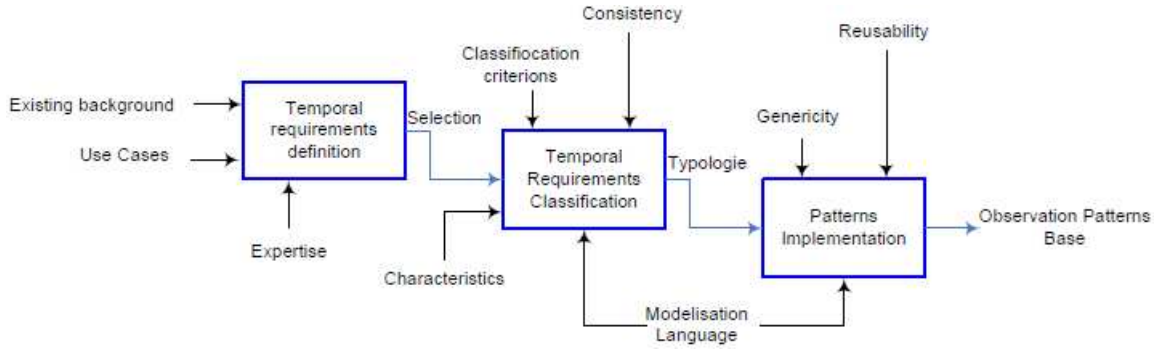


FIGURE 3.18 – Le processus général de l'utilisation des schémas [?]

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.

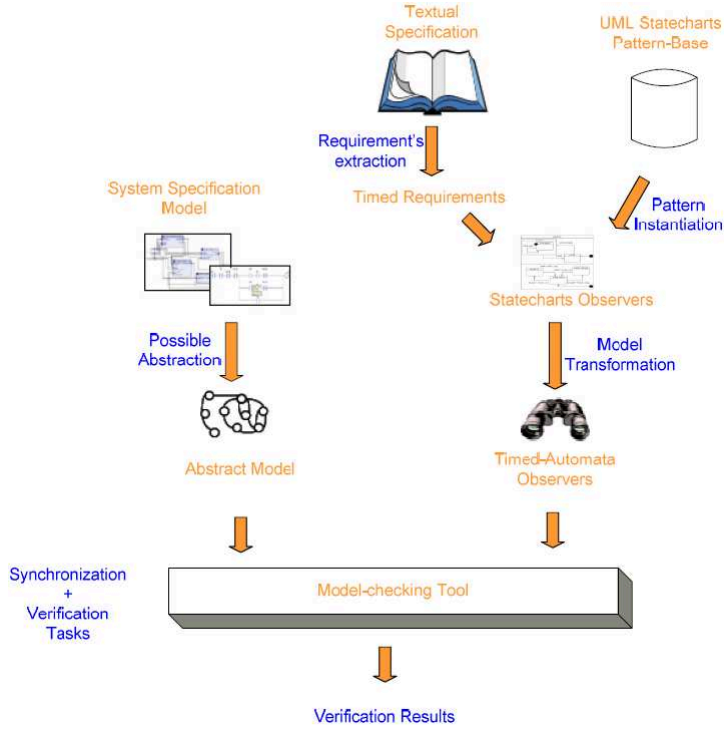


FIGURE 3.19 – Le processus de vérification [?]

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 séquencement 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 Tmin (R affirme que l'événement doit apparaître après un temps minimum Tmin)
- MaximumDelay : S $\models$ R est vrai si l'événement apparaît avant Tmax (R affirme que l'événement doit apparaître avant un temps maximum Tmax)
- Latency : S $\models$ R est vrai si l'événement apparaît dans l'intervalle temporel $]Tmin; Tmax[$
- 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

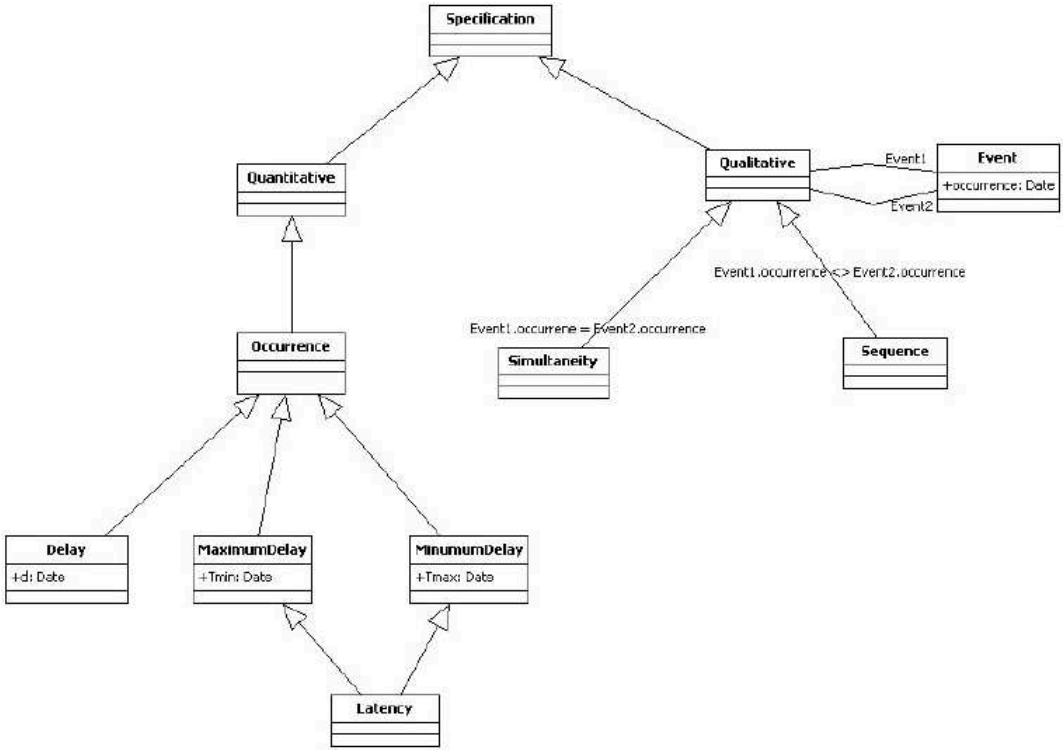


FIGURE 3.20 – Classification des contraintes de temps [?]

Schémas observateurs (patterns observateurs)

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

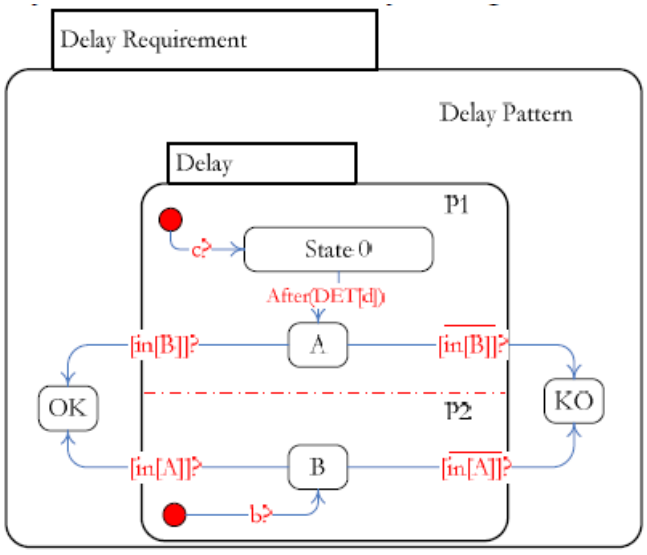


FIGURE 3.21 – Pattern de la contrainte "Delay" [?]

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

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

Transformation des SMD vers les automates temporisés (TA)

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

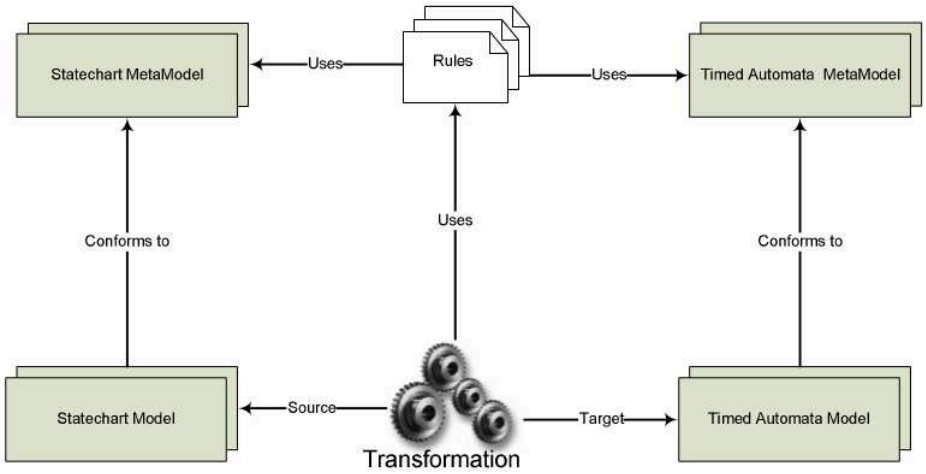


FIGURE 3.22 – Architecture de la transformation des SMD vers les TA [?]

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

Model checking pour les diagrammes états-transitions et de collaborations avec du temps

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

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

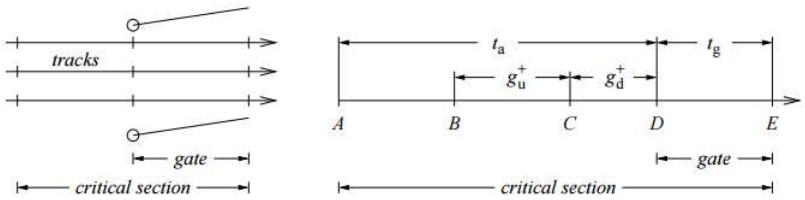


FIGURE 3.23 – Annotation temporelle de l'exemple du passage de train [?]

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

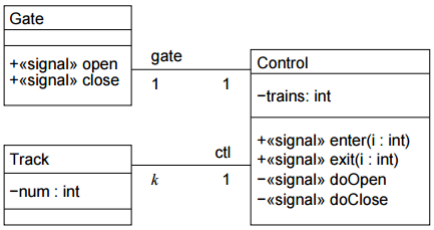


FIGURE 3.24 – Diagramme de classes de l'exemple du train [?]

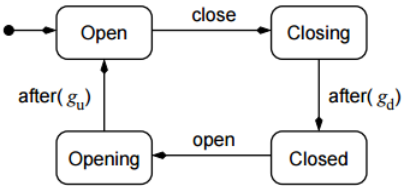


FIGURE 3.25 – Diagramme états-transitions de la barrière [?]

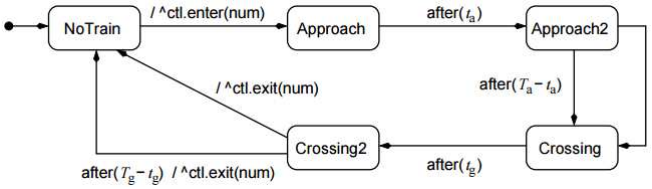


FIGURE 3.26 – Diagramme états-transitions du parcour du train [?]

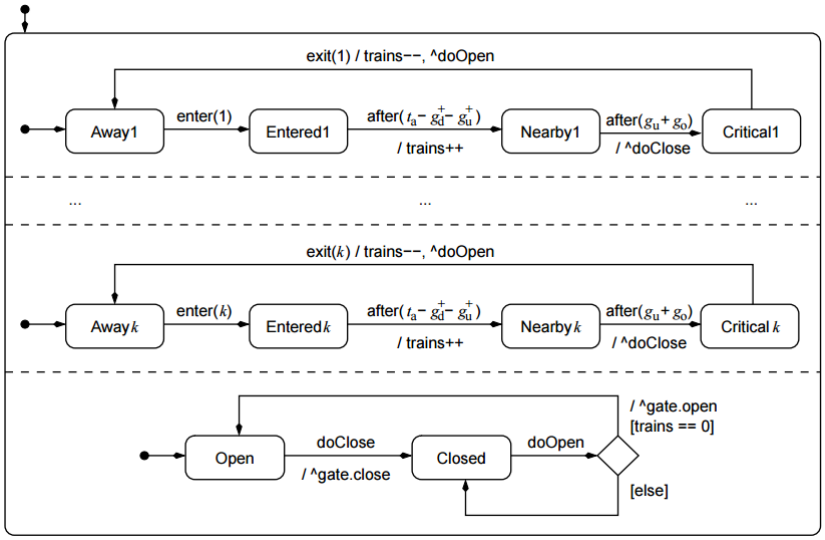


FIGURE 3.27 – Diagramme états-transitions du contrôleur [?]

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

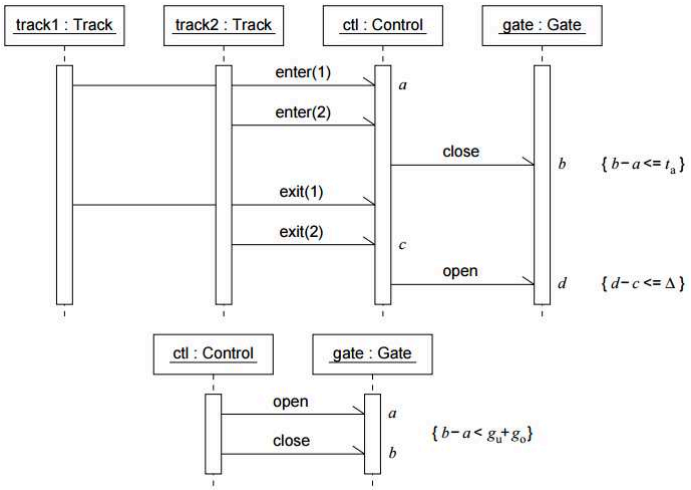


FIGURE 3.28 – Diagramme de séquence pour la propriété de sureté et d'utilité [?]

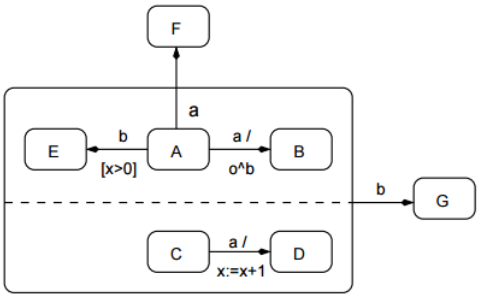


FIGURE 3.29 – Exemple diagramme états-transitions [?]

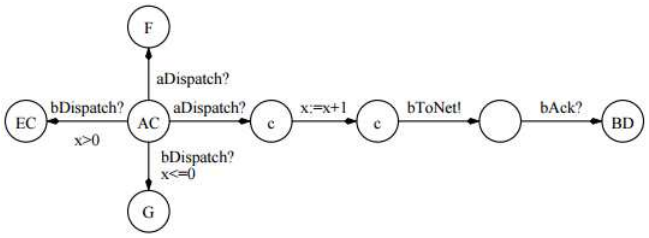


FIGURE 3.30 – Traduction de l'exemple de la Figure 3.29 [?]

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

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

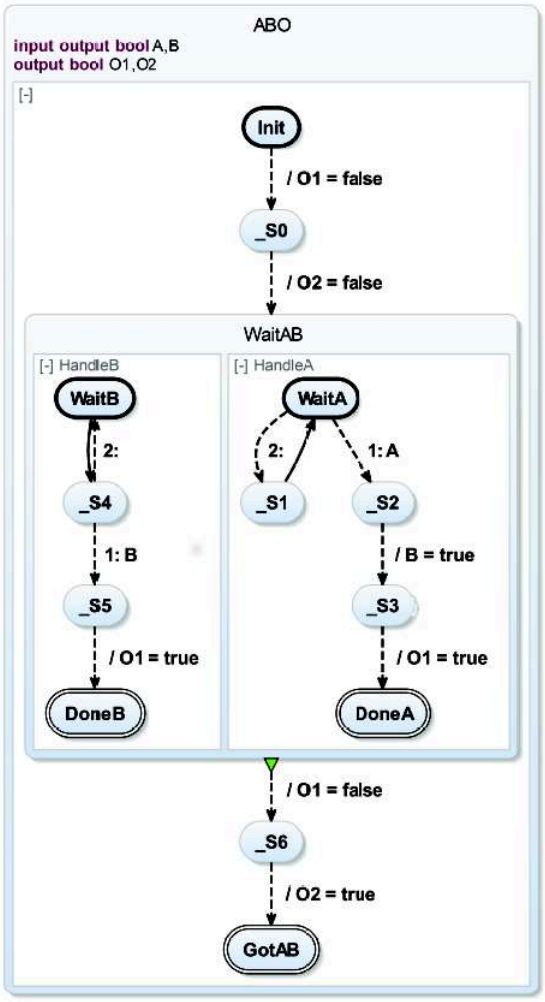


FIGURE 3.31 – Exemple de SCCharts [**?**]

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

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

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

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

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

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

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

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

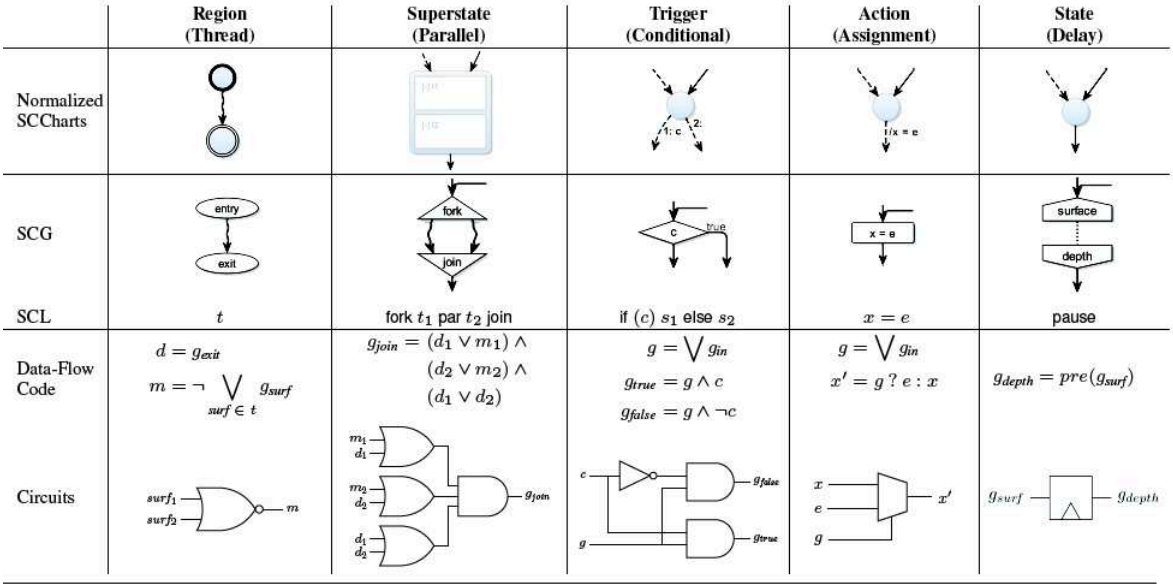


FIGURE 3.32 – Règles de transformations des SCCharts vers les SC graphs [?]

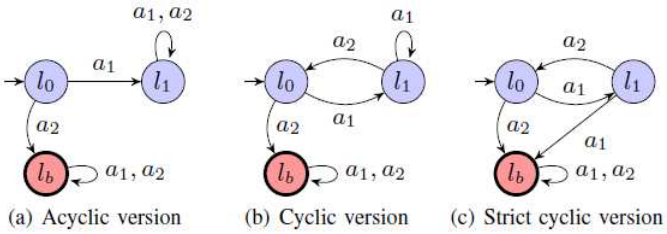


FIGURE 3.33 – Observateur automate temporisé pour la propriété de précédence[?]

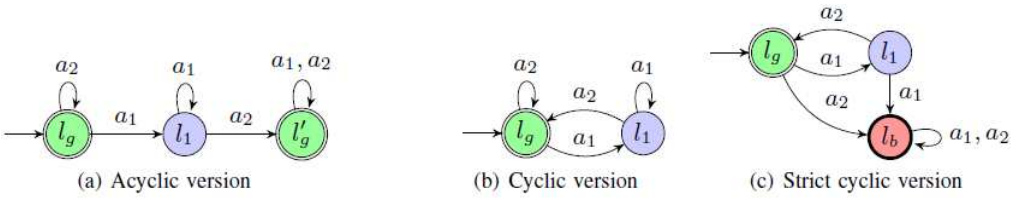


FIGURE 3.34 – Observateur automate temporisé pour la propriété de vivacité [?]

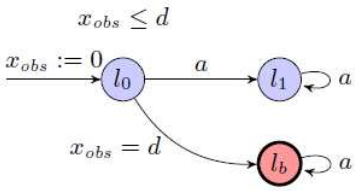


FIGURE 3.35 – Observateur automate temporisé pour la propriété de ponctualité [?]

Patterns pour la modélisation des systèmes temporisés

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

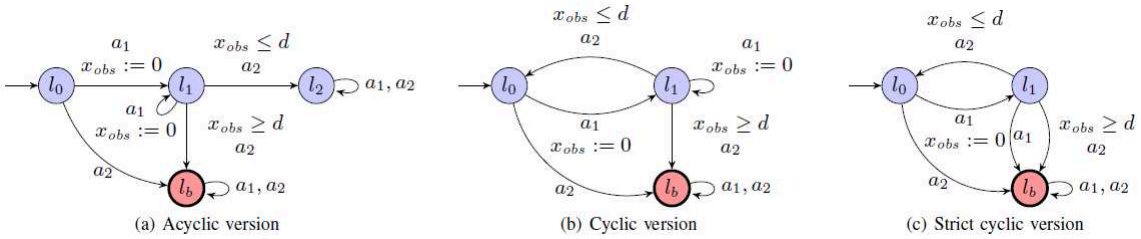


FIGURE 3.36 – Observateur automate temporisé pour la propriété de précédence avec un intervalle [?]

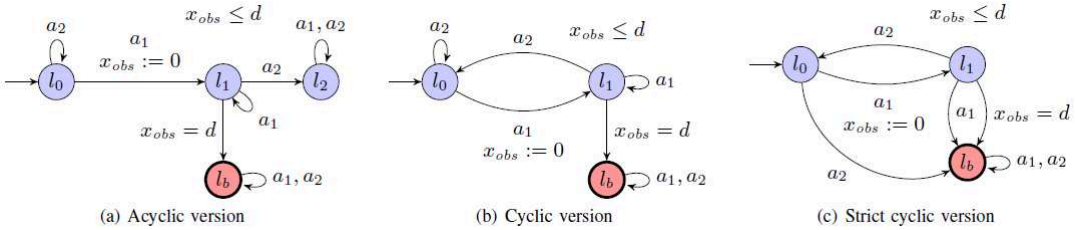


FIGURE 3.37 – Observateur automate temporisé pour la propriété de latence [?]

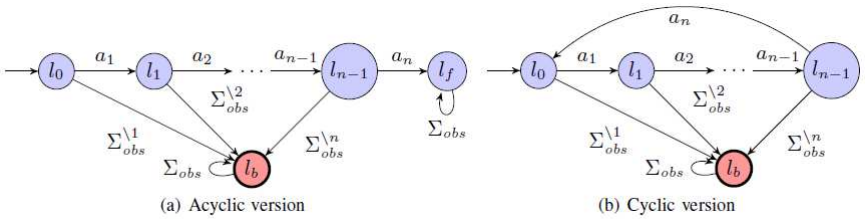


FIGURE 3.38 – Observateur automate temporisé pour la propriété de séquence [?]

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

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

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

Passage des contraintes de planning vers les automates temporisés

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

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

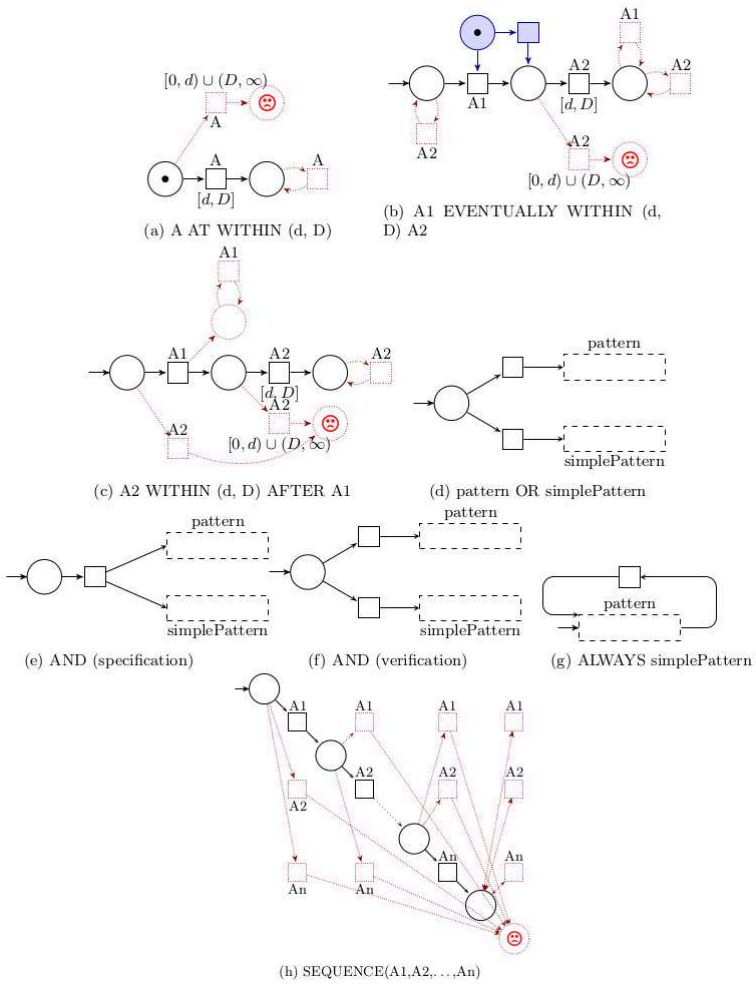


FIGURE 3.39 – Traduction des différents patterns vers les réseaux de Petri temporisés [?]

la Figure 3.40

Le profile UML hybride pour UML 2.0

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

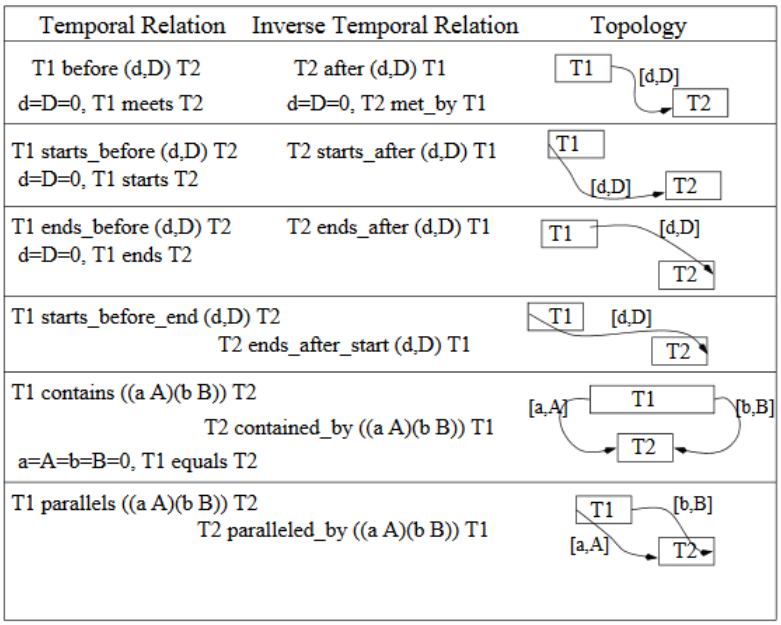


FIGURE 3.40 – Relations temporelles basées sur les intervalles [?]

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.

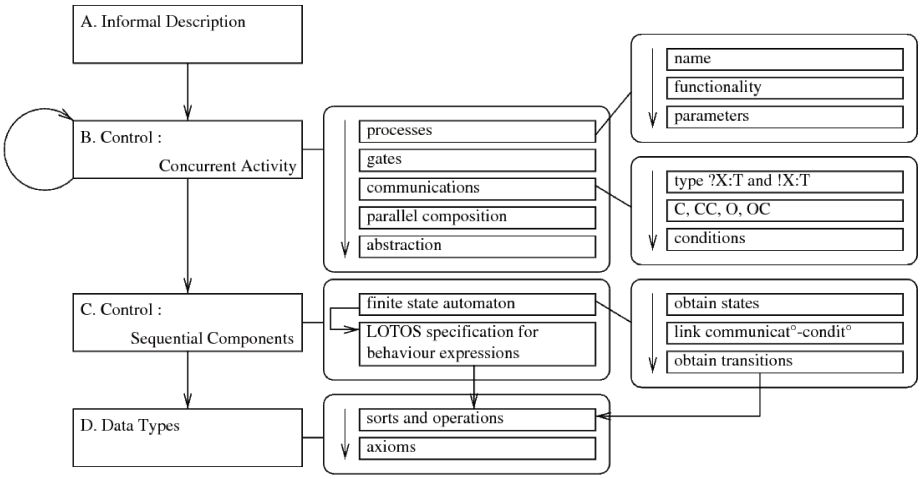


FIGURE 3.41 – La méthode pour la spécification *LOTOS*

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 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 ou 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-

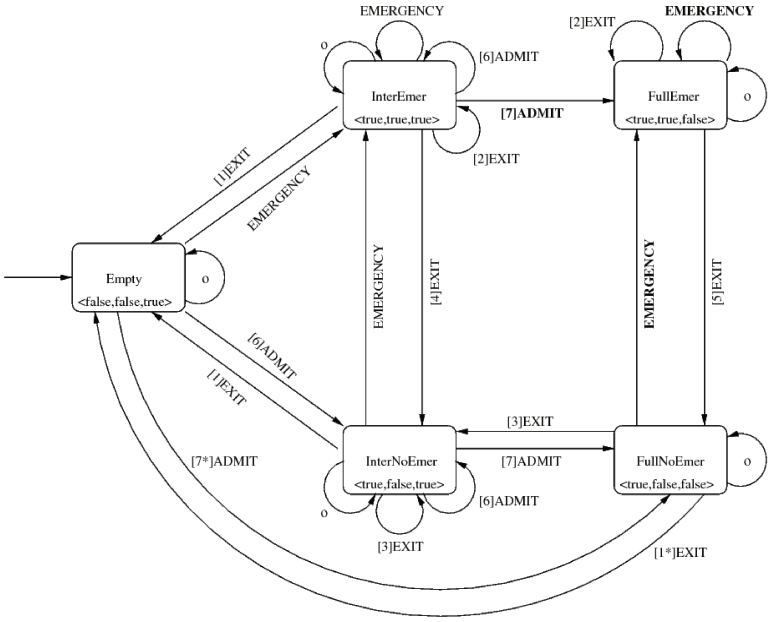


FIGURE 3.42 – L'automate de l'exemple de l'hôpital

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

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

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

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

UML statecharts mobile

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

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

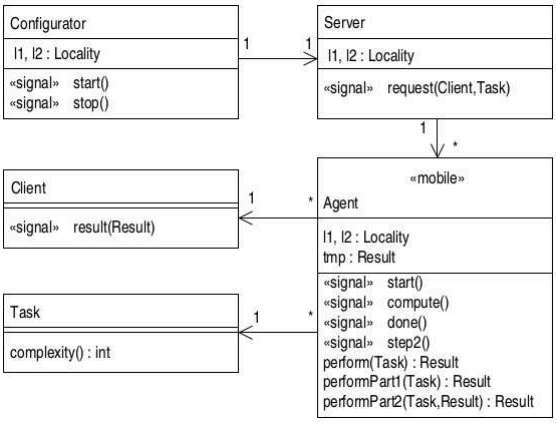


FIGURE 3.43 – Diagrammes de classes du service de réseau

Une approche de modélisation en utilisant les réseaux de Petri colorés

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

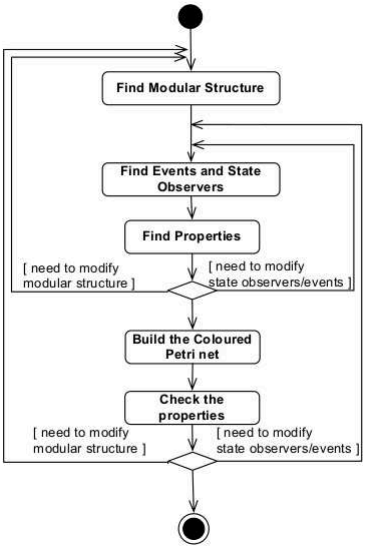


FIGURE 3.44 – Étapes de la méthode

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

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

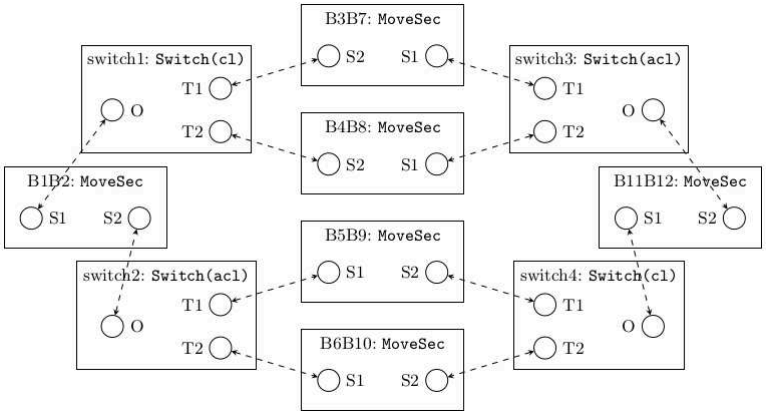


FIGURE 3.45 – Exemple du train électrique : réseau de Petri coloré

Conception des systèmes basés agents en utilisant UML

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

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

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

Les différentes étapes de la méthode sont les suivantes :

- Spécification de la fonctionnalité du système comme étant une organisation. La fonctionnalité du système est exprimée en terme de services fournis à l'utilisateur. Les services fournis à l'utilisateur peuvent être mappés sur les tâches sociales des agents membre de l'organisation.

- La conception du modèle de l'organisation est présentée en terme de modèle de rôles de règles sociales. Cela implique l'identification des positions et des fonctions qui doivent être réalisées par les agents pour leurs tâches individuelles mais aussi les règles sociales que les agents membres de l'organisation doivent suivre.

- L'identification des patterns d'interaction qui représentent les protocoles de communication entre différents rôles, et par conséquent la définition du modèle d'interaction.

- La conception du modèle de l'environnement est présentée en terme des ressources existantes ainsi que l'accès des protocoles à ces dernières.

Une traduction des statecharts vers le langage formel Promela est discutée avec une présentation d'une étude de cas sur la gestion des étudiants dans une université virtuelle. La modélisation de l'étude de cas est présentée sous forme de différents modules graphiques et différents diagrammes UML (tels que le diagramme de séquence, le diagramme états-transitions et le diagramme use case).

Conclusion

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

Dans le chapitre suivant (Chapitre 4) nous présentons une méthode de spécification en utilisant les diagrammes états-transitions et les différentes améliorations/extensions que nous proposons.

Chapitre 4

Méthode de spécification en utilisant les diagrammes états-transitions

Contents

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

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

Concepts utilisés dans la méthode de spécification

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

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

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

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

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

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

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

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

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

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

Invariants

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

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

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

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

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

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

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

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.

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.

5. Enfin, dans la dernière étape, le diagramme de classes de départ est mis à jour pour le 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.

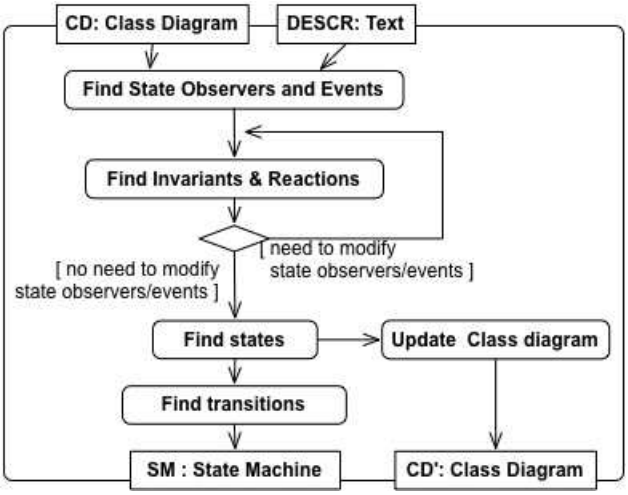


FIGURE 4.1 – Schéma des étapes de la méthode de spécification [?]

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

Détails de la méthode de spécification et améliorations

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

Vue générale de la méthode

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

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

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

Analyse du texte

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

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

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

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

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

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

Nous présentons dans ce qui suit la description textuelle de l'exemple de gestion de bibliothèque (exemple présenté dans [**?**]) ainsi que le diagramme de classes de départ (voir Figure 4.3).

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

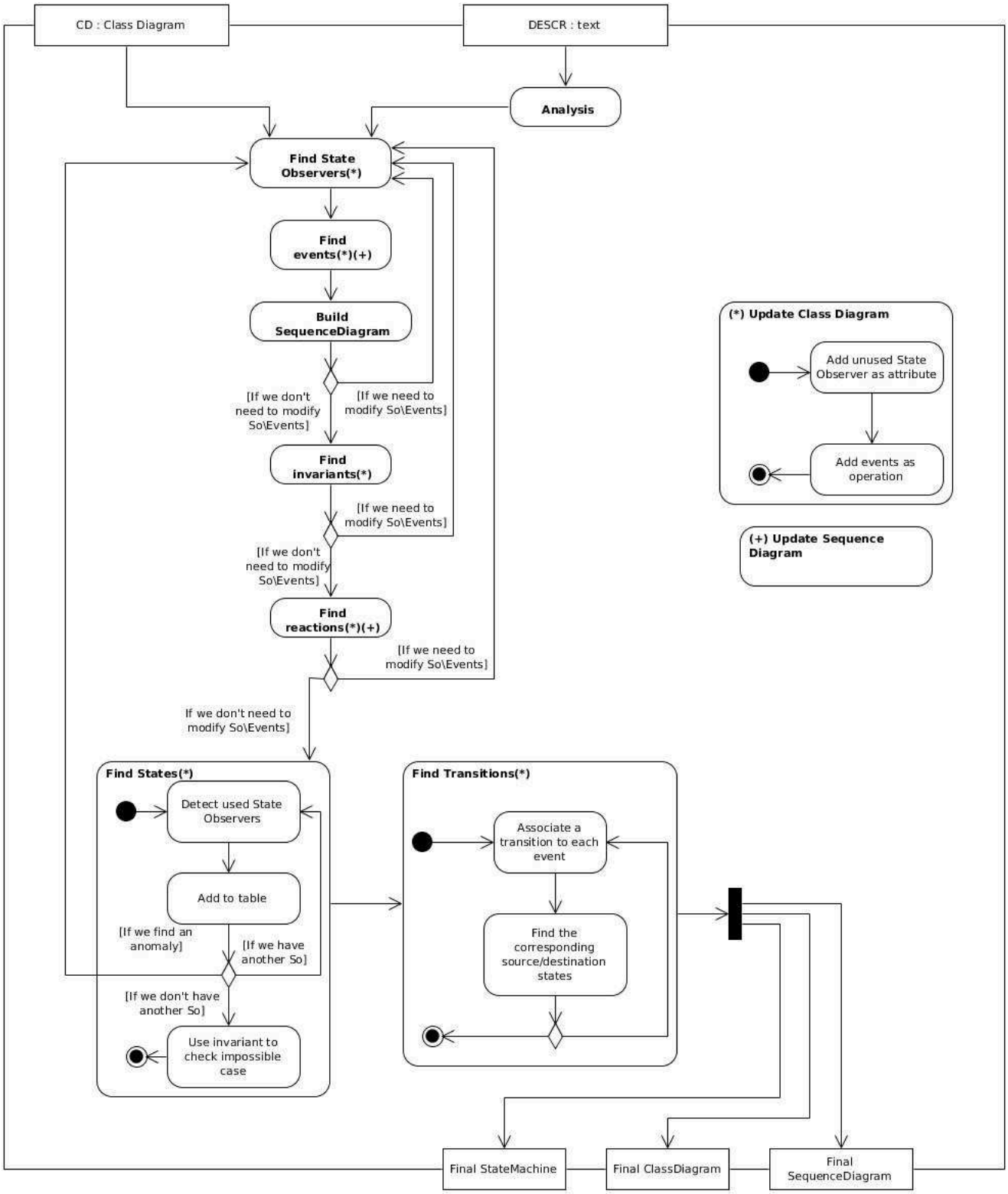


FIGURE 4.2 – Diagramme d'activité des étapes de la méthode

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

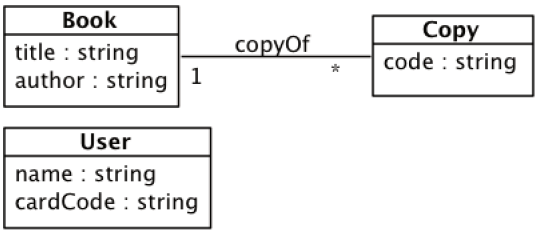


FIGURE 4.3 – Diagramme de classes de la gestion de bibliothèque

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

Recherche des observateurs d'état [?]

Le contenu de cette sous-section est une amélioration d'une étape existante dans la méthode de [?]. À la suite de l'analyse du texte, nous cherchons à extraire les observateurs d'état. Le 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

- Nous définissons un observateur d'état particulier (nommé *final* qui prend la valeur *true* dans le cas où l'exécution du système n'est pas en boucle et peut se terminer, et prend la valeur *false* dans le cas inverse.

- Le type des valeurs d'un observateur d'état peut être un type énuméré. Le travail effectué à ce stade permet d'affiner les valeurs pertinentes et le typage.

- **Remarque sur le type d'un observateur d'état :** Si un observateur d'état *obsI* peut prendre un ensemble infini de valeurs possibles, cet observateur ne peut être utilisé tel quel pour caractériser des états, car il y aurait une infinité d'états associés à ces valeurs. Cependant, ignorer cet observateur peut entraîner un manque d'information concernant les états. Afin de remédier à cela, nous rajoutons un nouvel observateur d'état *obsF* (de type énuméré prenant un nombre fini de valeurs) qui permet de combler l'absence de l'observateur d'état *obsI*, et qui sera utilisé pour caractériser les états. L'observateur d'état *obsF* permet de pallier à l'absence de *obsI*. L'observateur d'état *obsI* n'aura pas d'invariant dans la table des invariants et sera ajouté comme un attribut dans le diagramme de classes final. Dans le Tableau 4.2, nous présentons un ensemble d'observateurs d'état avec différents types. L'observateur d'état `code` peut prendre un ensemble infini de valeurs et par conséquent il est inutilisable dans la table des états. Cependant, son absence enlève 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
code	*String*	*ZeroOrOne*
inLib	*Boolean*	*One*
onLoan	*Boolean*	*One*
booked	*Boolean*	*One*
booker	*User*	*ZeroOrOne*
tagged	*Boolean*	*One*
final	*Boolean*	*One*

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
$event_1$
...
$event_k(type)$
created

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
$insertLib()$
$lend(User)$
$bringBack()$
$book()$
$putTag()$
$removeTag()$
$created()$

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

Construction du diagramme de séquence

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

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

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

Recherche des invariants des observateurs d'état [?]

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

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

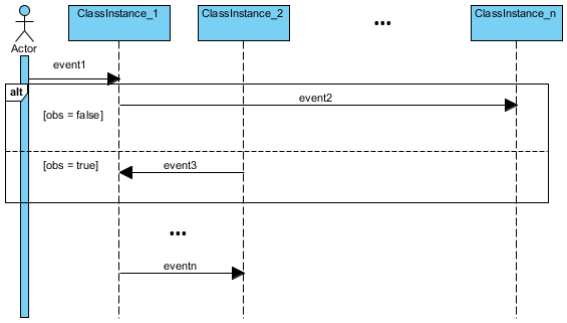


FIGURE 4.4 – Diagramme de séquence

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 séquencement 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.

Le Tableau 4.6 représente les invariants des observateurs d'état pour l'exemple de la gestion 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

Recherche des conditions / réactions des événements [?]

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

Événement	**Condition**	**Réaction**
$event_1$	$condition_1$	$reaction_1$
...	...	...
$event_k$	$condition_k$	$reaction_k$
created	/	*Initialisation*

TABLE 4.7 – Conditions/réactions des événements

Remarques

- Dans certains cas il est nécessaire de réinitialiser les valeurs des observateurs d'état (cela représente généralement l'état initial du système à l'absence d'un état final par exemple). Par conséquent, il faut chercher le moment pertinent pour le faire (c'est-à-dire dans la partie réaction de quel événement il faut faire cela).
- Dans le cas où il est nécessaire d'utiliser dans la partie réaction des événements les opérations de différentes classes, alors il est impératif d'établir des associations entre ces classes et la classe de contexte. Cette modification impliquera de mettre à jour le diagramme de classes avec les différentes associations établies. Les associations sont également essentielles 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 définissons 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).

obs_1	**...**	**obs_i**	**État**
$valeur_1$	...	$valeur_1$	nom_1
...	...	...	...
$valeur_k$	...	$valeur_k$	nom_k
$valeur_m$	...	$valeur_m$	*impossible*
...	...	...	...
$valeur_n$	...	$valeur_n$	*impossible*

TABLE 4.9 – États

Le Tableau 4.10 montre les différents états trouvés lors du traitement de l'exemple de la gestion de bibliothèque. Nous remarquerons qu'il y a un ensemble d'état possibles et un ensemble d'états impossibles (dont la lignes entre en contradiction avec l'un des invariants trouvés 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

Recherche des transitions [?]

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

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

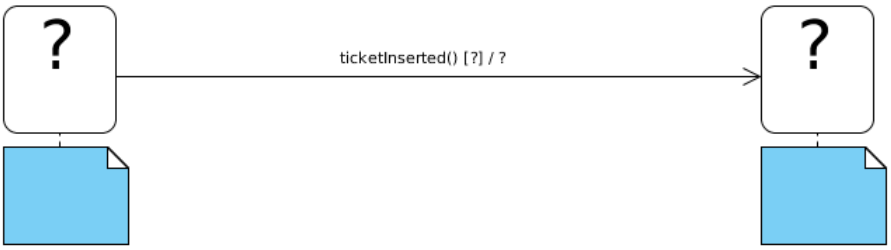


FIGURE 4.5 – Modèle de recherche d'une transition

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

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

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

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

Génération du diagramme états-transitions [?]

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

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

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

Vérification de la cohérence

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

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

- La première phase consiste à vérifier (d'une manière informelle) les invariants des observateurs d'état. En effet, il possible de vérifier si les valeurs des observateurs d'état utilisées dans l'invariant sont les bonnes et cela en nous basant sur la séquence de l'occurrence des événements dans le diagramme de séquence. À chaque occurrence d'un événement il y a généralement un changement d'état du système et par conséquent un changement de valeur de certains observateurs d'état.

- La seconde phase est la vérification des conditions/réactions de chaque événement. Pour chaque occurrence d'un événement il est possible de vérifier sa condition (réaction) en utilisant le diagramme de séquence. La condition modélise la situation dans laquelle se trouvait le système avant l'occurrence de l'événement. Par conséquent, dans le diagramme de séquence, cela est modélisé par les différents événements qui sont apparus avant l'événement en question, mais aussi par la valeur des observateurs d'état.

- La troisième phase consiste à vérifier si le scénario que le diagramme de séquence représente est le même que celui du diagramme états-transitions. Cela est pertinent car les deux diagrammes tracent les changements d'états et l'occurrence des événements lors du fonctionnement de notre système. Si l'application de la méthode est correcte et si les résultats des différentes étapes sont cohérents, alors les deux diagrammes seront identiques du point de vue du fonctionnement du système. Cette vérification est faite manuellement (sans l'utilisation d'outil).

Améliorations apportées à la méthode de spécification

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

- Nous intégrons dans un premier temps une phase d'analyse (basée sur le texte fourni par l'utilisateur) pour préciser le déroulement du fonctionnement du système mais aussi détecter les observateurs d'état / événements candidats.

- Nous séparons l'identification des observateurs de celle des événements afin d'avoir deux phases où chaque partie est traitée en profondeur.

- Nous ajoutons un diagramme de séquence (basé sur les observateurs d'état et les événements identifiés mais surtout sur l'analyse du texte) pour deux raisons : la première est qu'avoir un séquencement du fonctionnement du système peut aider à exprimer par la suite les invariants des observateurs d'état et les conditions / réactions des événements. La seconde raison est de pouvoir vérifier la cohérence entre le diagramme états-transitions obtenu par la méthode et le diagramme de séquence de départ.

- Nous séparons également la définition des invariants et les conditions / réactions en deux phases, cela pour à la fois approfondir le traitement de chaque partie mais également pour avoir la possibilité de revenir en arrière s'il y a une erreur ou une incohérence.

- Enfin, nous détaillons la manière de définir les états, les transitions et de faire la mise à jour du diagramme de classes qui apparaît à chaque étape. Afin de définir les états nous commençons par la sélection et l'ajout des observateurs d'état utilisés dans l'identification

des états, ensuite nous vérifions les invariants pour détecter les états impossibles. À chaque événement, nous associons une transition (avec un état source et un état cible) de telle manière que les états source et cible de chaque transition soient conformes aux conditions et réactions de l'événement. Concernant les transitions, nous dessinons les événements sous forme de transition (avec un état source et un état cible) et selon les conditions réactions 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 avions traité cet exemple pour acquérir une compréhension approfondie de la méthode présentée dans [**?**]. Dans les faits, cela a entraîné de nombreuses discussions avec les auteures, 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.

5. L'utilisateur insère sa carte bancaire : le contrôleur est informé de l'insertion de la carte bancaire par le lecteur de carte bancaire qui contacte la banque pour la vérification du paiement.

6. Si le paiement échoue alors la banque informe le contrôleur de l'échec du paiement qui ordonne au lecteur de carte bancaire d'éjecter la carte et l'utilisateur revient à l'état où il insert du ticket.

7. Si le paiement réussit alors la banque informe le contrôleur de la validité du paiement, ce dernier ordonne au lecteur de carte bancaire d'éjecter la carte bancaire et ordonne la mise en marche du moteur de la barrière.

8. Le détecteur de barrière informe le contrôleur que la barrière est en position haute alors le contrôleur ordonne l'arrêt du moteur de la barrière.

9. Le détecteur de présence de voiture informe le contrôleur du départ de la voiture alors le contrôleur ordonne la mise en marche du moteur de la barrière pour la descente.

10. Le détecteur de barrière informe le contrôleur que la barrière est en position basse alors le contrôleur ordonne l'arrêt du moteur de la barrière.

11. Revenir à l'état où une voiture arrive à la barrière.

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

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

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

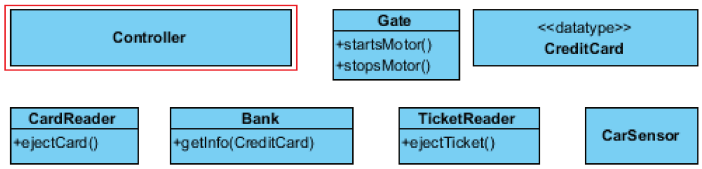


FIGURE 4.6 – Diagramme de classes de l'exemple du paiement du parking

Observateurs d'état et événements

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

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

Événement
`ticketInserted` ()
`ticketFailure` ()
`ticketSuccess` ()
`card_Inserted` (CreditCard)
`receivePaymentOk` (CreditCard)
`receivePaymentFail` (CreditCard)
`gateUp` ()
`gateDown` ()
`carArrives` ()
`carLeaves` ()

Observateur d'état	Type	Multiplicité
`ticketActive`	Bool	One
`ticketReadOk`	BoolS	One
`cardInfo`	CreditCard	ZeroOrOne
`cardKnown`	Bool	One
`paymentOk`	BoolS	One
`gateState`	GateS	One
`carAtGate`	Bool	One

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 tables 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 *note 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 avions 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.

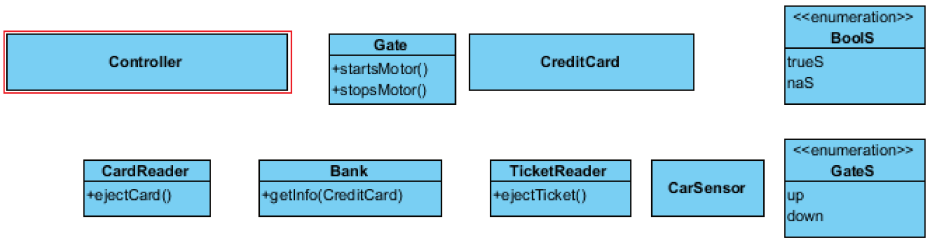


FIGURE 4.7 – Diagramme de classe de l'exemple du paiement du parking : mise à jour des nouvelles classes

Afin d'éclaircir l'usage des événements et des observateurs d'état, nous présentons dans ce qui suit l'explication de chaque événement/observateur d'état utilisé :

- `ticketInserted` () : l'événement modélise l'insertion du ticket (l'observateur d'état `ticketActive` reflète le fait qu'un ticket a été inséré et est en cours de traitement).
- `ticketFailure` () : l'événement modélise l'échec de la lecture du ticket.
- `ticketSuccess` () : l'événement modélise le succès de la lecture du ticket (la valeur *true* pour l'observateur d'état `ticketReadOk` reflète que la lecture a été bien effectuée).
- `card_Inserted` (CreditCard) : l'événement modélise l'insertion de la carte bancaire (l'observateur d'état `cardInfo` fournit les informations sur la carte bancaire).
- `receivePaymentFail` (CreditCard) : l'événement modélise l'échec du paiement.
- `receivePaymentOk` (CreditCard) : l'événement modélise la réussite du paiement (la valeur *true*S pour l'observateur d'état `paymentOk` indique que ce paiement a bien été effectué).
- `gateUp` () : l'événement modélise la détection de la position de la barrière en haut.
- `gateDown` () : l'événement modélise la détection de la position de la barrière en bas.
- `carArrives` () : l'événement modélise la détection de l'arrivée d'une voiture (qui est observée par `carAtGate`).
- `carLeaves` () : l'événement modélise la détection du départ d'une voiture.

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

Les noms choisis pour les différents événements et opérations sont utilisés dans le diagramme de séquence Figure 4.8.

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

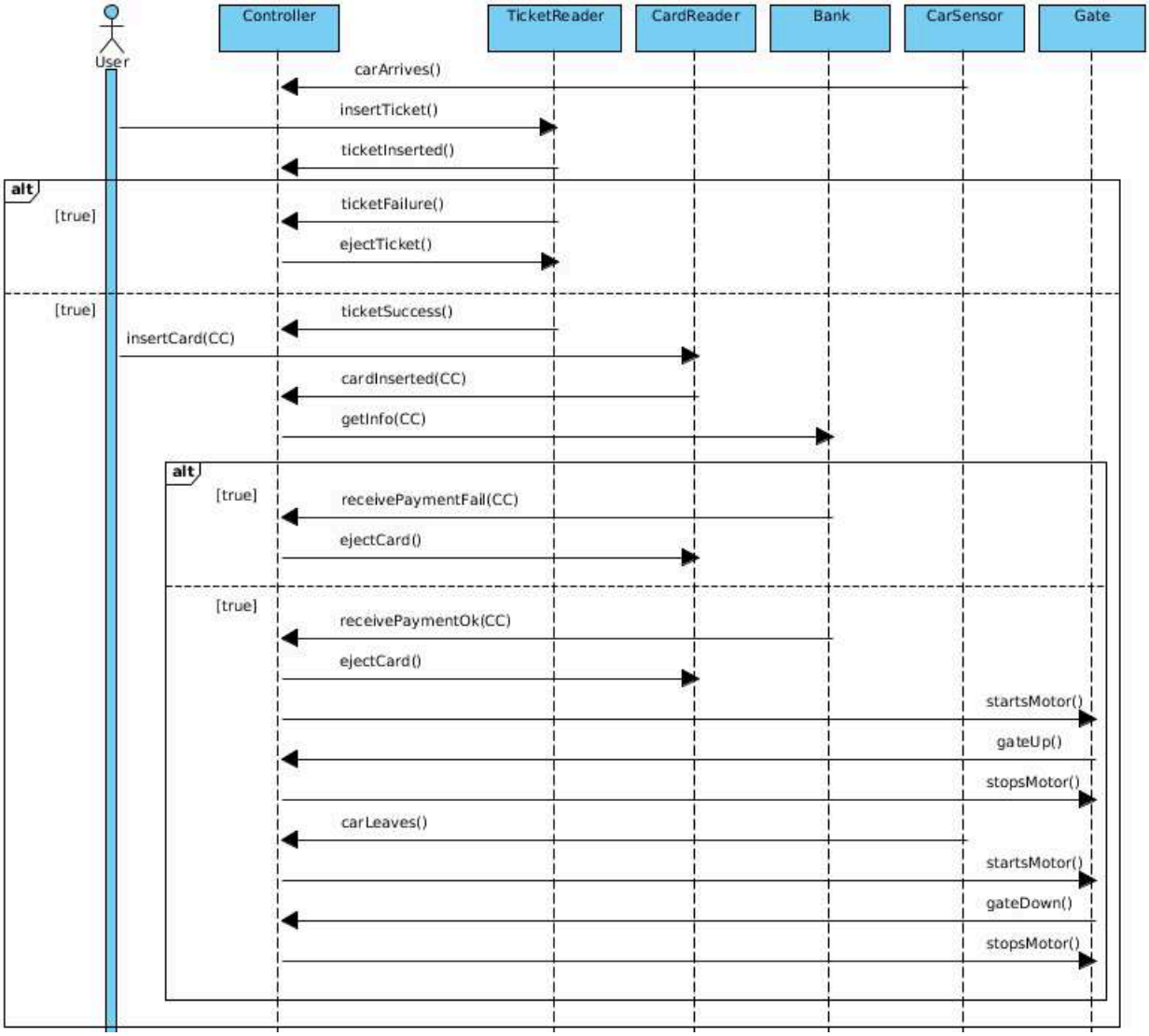


FIGURE 4.8 – Diagramme de séquence du déroulement du système

Invariants et réactions

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

Invariants

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

Initialement la barrière est vide et il n'y a aucune voiture. Cela nous permet de définir la valeur initiale de chaque observateur d'état (extrait dans l'étape précédente) :
`carAtGate` := *false* ; `ticketActive` := *false* ; `ticketReadOk` := BoolS.naS ;
`cardKnown` := *false* ; `paymentOk` := BoolS.naS ; `gateState` := GateS.down.
Ces valeurs initiales changent dans le cas où le système change d'état en réaction à un événement.

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

CAG	TA	TRO	CK	PO	GS
false	*false*	*BoolS.naS*	*false*	*BoolS.naS*	*GateS.down*

Invariant de `carAtGate`

À la réception de l'événement `carArrives`, la valeur de l'observateur d'état `carAtGate` change en *true* car la voiture est présente au niveau de la barrière. Nous définissons l'invariant de `carAtGate` quand sa valeur est *false*, car il y a beaucoup de situations où sa valeur est égale à *true* (moins pertinent). En partant de ce raisonnement, `carArrives` a la valeur *false* si seulement si notre système est à son état initial (c'est à dire chaque observateur d'état est à sa valeur initiale). Par conséquent, nous définissons l'invariant de l'observateur d'état `carAtGate` comme suit :
`carAtGate` = *false* ⇒ `ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` = GateS.down.

CAG	TA	TRO	CK	PO	GS
false	*false*	*BoolS.naS*	*false*	*BoolS.naS*	*GateS.down*

Invariant de `ticketActive`

Une fois à la barrière, l'utilisateur insère son ticket ; cette situation est représentée par l'occurrence de l'événement `ticketInserted`. En nous basant sur le diagramme de séquence, l'insertion du ticket vient après l'arrivée de la voiture à la barrière, cela implique que `ticketActive` est à *true* uniquement si `carAtGate` = *true*. Cependant caractériser la situation où le ticket est inséré uniquement avec la présence de la voiture est nécessaire mais pas suffisant, car la présence de la voiture caractérise également d'autres situations (par exemple, l'insertion de la carte bancaire, la validation du paiement, etc). Pour cette raison, nous allons reprendre l'ensemble des observateurs d'état avec leurs valeurs initiales à l'exception de `carAtGate`. L'invariant est le suivant :
`ticketActive` = *false* ⇒ `carAtGate` = *true* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` = GateS.down.

CAG	TA	TRO	CK	PO	GS
true	*false*	*BoolS.naS*	*false*	*BoolS.naS*	*GateS.down*

Invariant de `ticketReadOk`

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

afin que la lecture réussisse il faut que la voiture soit au niveau de la barrière, le ticket soit inséré et le reste du système à son état initial (voir le diagramme de séquence). Cela nous permet de définir l'invariant suivant :
`ticketReadOk` = BoolS.*true*S ⇒ `carAtGate` = *true* and `ticketActive` = *true* and `cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` = GateS.down.

CAG	TA	TRO	CK	PO	GS
true	*true*	*BoolS.trueS*	*false*	*BoolS.naS*	*GateS.down*

Invariant de `cardKnown`

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

CAG	TA	TRO	CK	PO	GS
true	*false*	*BoolS.naS*	*true*	*BoolS.naS*	*GateS.down*

Invariant de `paymentOk`

À l'insertion de la carte bancaire ou bien dans le cas où le paiement est refusé, l'événement `receivePaymentFail` apparaît et par conséquent la carte bancaire est éjectée et on revient à un état précédent, ou bien le paiement est accepté et dans ce cas là l'événement `receivePaymentOk` apparaît. L'occurrence de l'événement `receivePaymentOk` permet de changer l'état du système et donc la valeur de l'observateur d'état `paymentOk` (qui aura *true*). Si le paiement est accepté alors la voiture est au niveau de la barrière et le reste des observateurs d'état sont à leur état initial (la même chose pour `cardKnown`, car la carte bancaire est éjectée). L'invariant de l'observateur d'état `paymentOk` est le suivant :
`paymentOk` = *true* ⇒ `carAtGate` = *true* and `ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `gateState` = GateS.down.

CAG	TA	TRO	CK	PO	GS
true	*false*	*BoolS.naS*	*false*	*BoolS.trueS*	*GateS.down*

Invariant de `gateState`

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

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

L'observateur `carAtGate` n'apparaît pas dans l'invariant de l'observateur d'état `gateState` car il y a une situation où `carAtGate` est égale à *true* quand la barrière est en position haute, et 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
-	*false*	*BoolS.naS*	*false*	*BoolS.naS*	*GateS.up*

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

Une première remarque concerne l'invariant de l'observateur d'état `ticketActive`. Comment caractériser la condition ou la situation du système où l'observateur d'état a une valeur *true*, c'est-à-dire la situation où le ticket est inséré ? Un premier raisonnement serait que, pour le ticket soit inséré, il est nécessaire que la voiture soit au niveau de la barrière, c'est-à-dire `ticketActive` = *true* ⇒ `carAtGate` = *true*. Cependant, la présence de la voiture n'est pas suffisante pour caractériser la situation où le ticket est inséré. En effet, la présence de la voiture est nécessaire également pour les invariants des autres observateurs d'état (par exemple, la validation du paiement, etc). Par conséquent, il est nécessaire, lors de la définition de l'invariant de `ticketActive`, de prendre en compte la valeur des autres observateurs d'état au moment de l'insertion du ticket. Cela permet de définir l'invariant suivant (qui permettra de caractériser la situation où le ticket est inséré) :
`ticketActive` = *true* ⇒ `carAtGate` = *true* and `ticketReadOk` = BoolS.naS and
`cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` = GateS.down.

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

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

Événements

Pour chaque événement les couples condition/réaction sont présentés dans les Tableaux 4.13 et 4.14. Les conditions pour qu'un événement puisse avoir lieu sont exprimées à l'aide de valeurs prises par certains observateurs d'état, de même que les réactions après l'occurrence d'un événement.

State Observer	Invariant
`ticketActive`	`ticketActive` = *true* ⇒ (`carAtGate` = *true* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` = GateS.down)
`ticketReadOk`	`ticketReadOk` = BoolS.*true*S ⇒ (`carAtGate` = *true* and `ticketActive` = *false* and `cardKnown` = *false* and `paymentOk` = BoolS.naS and `gateState` =GateS.down)
`cardInfo`	
`cardKnown`	`cardKnown` = *true* ⇒ (`carAtGate` = *true* and `ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `paymentOk` = BoolS.naS and `gateState` = GateS.down)
`paymentOk`	`paymentOk` = BoolS.*true*S ⇒ (`carAtGate` = *true* and `ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `gateState` = GateS.down)
`gateState`	`gateState` = GateS.up ⇒ (`ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `paymentOk` = BoolS.naS and `cardKnown` = *false*)
`carAtGate`	`carAtGate` = *false* ⇒ (`ticketActive` = *false* and `ticketReadOk` = BoolS.naS and `cardKnown` = *false* and `paymentOk` = BoolS.naS)
final	*final*= *false*

TABLE 4.12 – Observateurs d'état et invariants

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

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

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

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

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

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

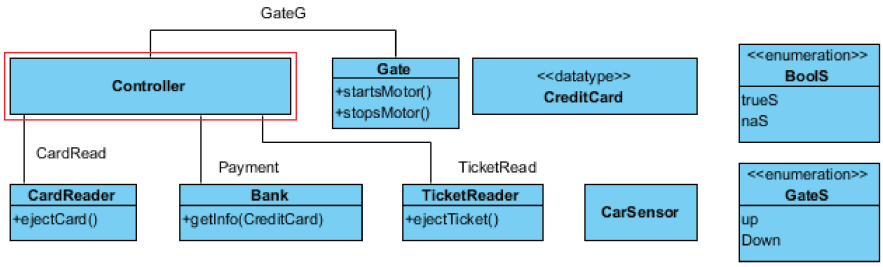


FIGURE 4.9 – Diagramme de classe de l'exemple du paiement du parking : mise à jour des classes avec les associations

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

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

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

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

Les états et les transitions. En nous basant sur les résultats des phases précédentes, dans cette étape nous trouvons les différents états et transitions qui constituent le diagramme d'états-transitions final. Les états sont résumés dans le Tableau 4.15.

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

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

TABLE 4.15 – Les états du diagramme états-transitions

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

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

Une remarque concerne l'événement *created* dans le Tableau 4.13 et l'état *init* dans Tableau 4.15. En effet, l'événement *created* correspond à la situation initiale du système (tel que chaque observateur d'état a une valeur initiale). Les valeurs de la ligne de l'état *init* correspondent aux valeurs des observateurs d'état de l'état cible de *created* et donc cet état sera l'état initial.

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

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

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

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

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

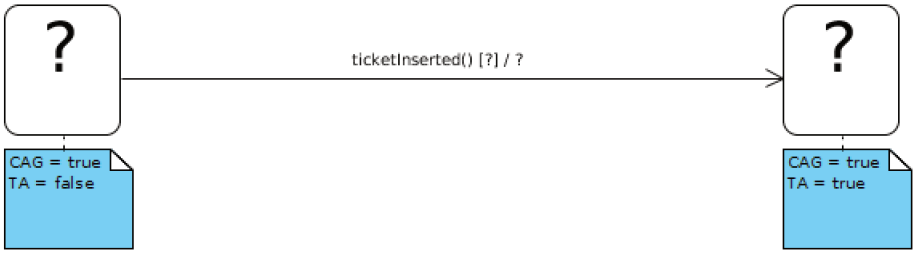


FIGURE 4.10 – La première étape pour trouver la transition de l'événement `ticketInserted`

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

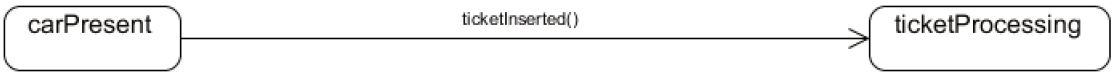


FIGURE 4.11 – La seconde étape pour trouver la transition de l'événement `ticketInserted`

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

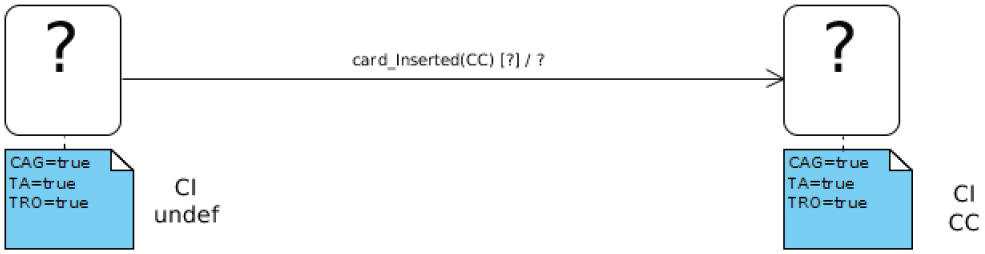


FIGURE 4.12 – La seconde étape pour trouver la transition de l'événement `card_Inserted`

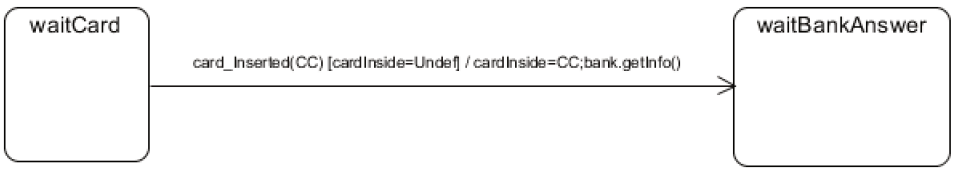


FIGURE 4.13 – La seconde étape pour trouver la transition de l'événement `card_Inserted`

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

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

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

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

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

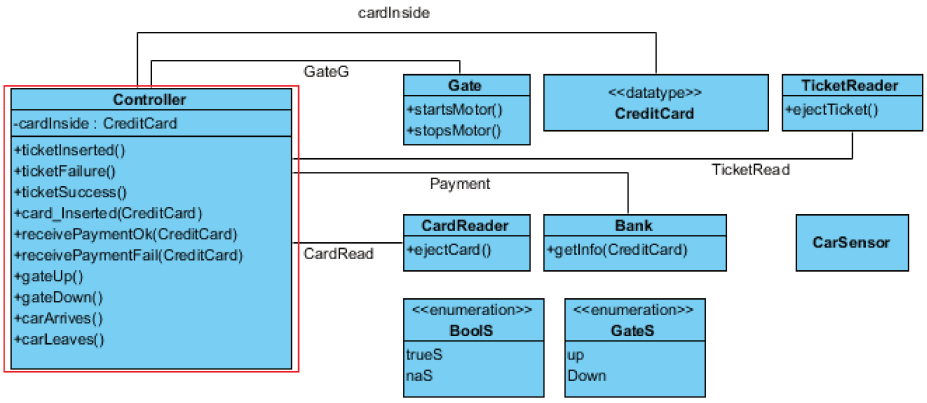


FIGURE 4.14 – Diagramme de classes final de l'exemple du paiement du parking

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

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

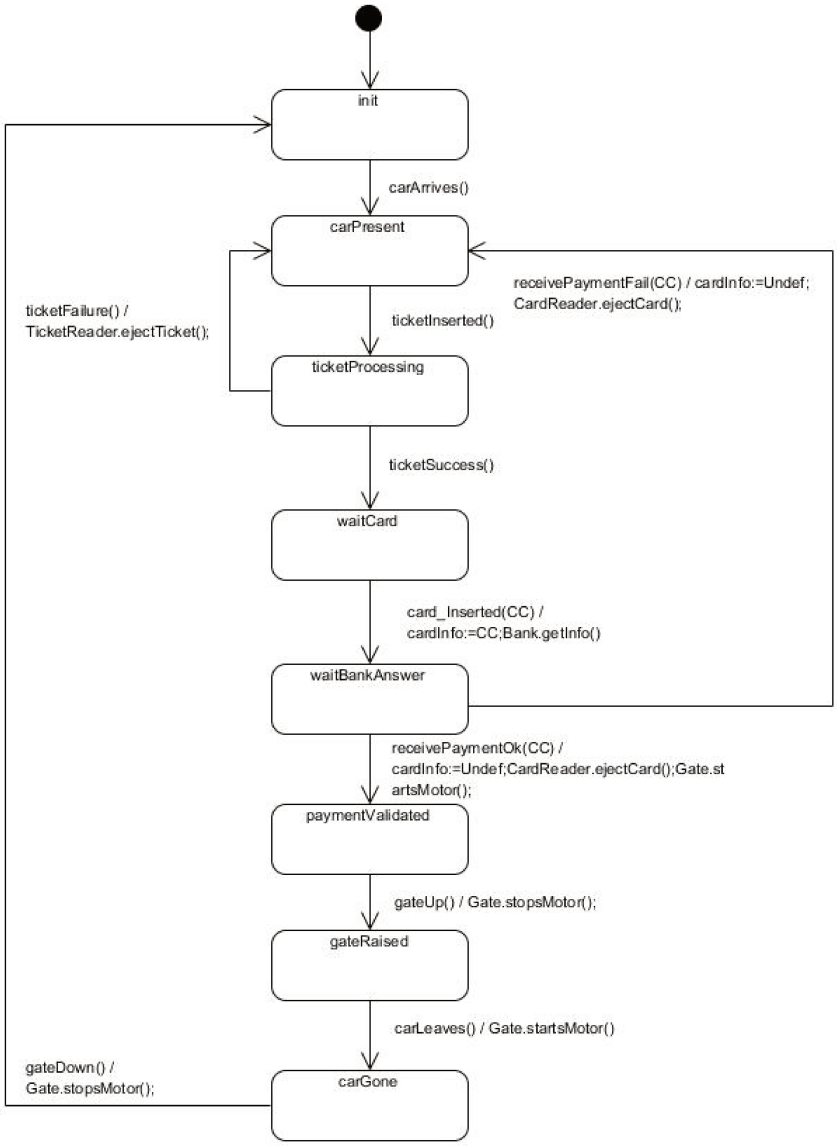


FIGURE 4.15 – Diagramme états-transitions de l'exemple du paiement du parking

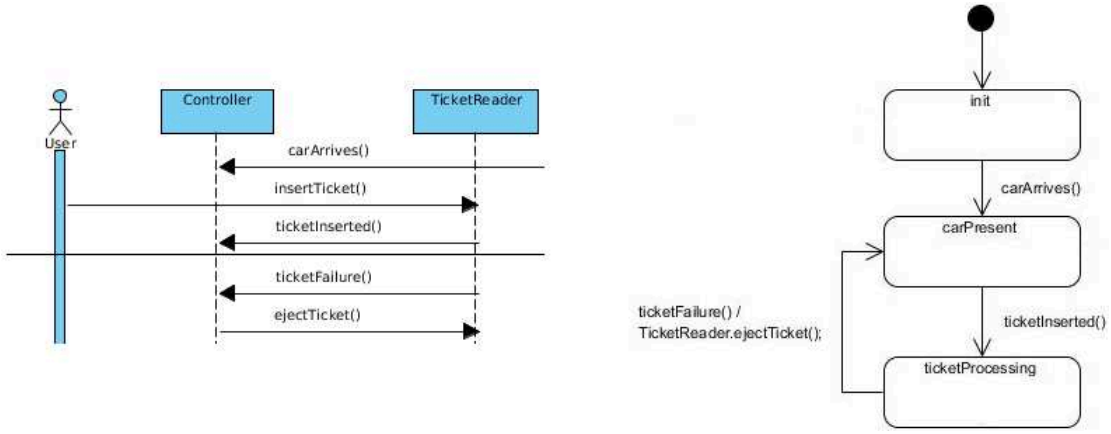


FIGURE 4.16 – Diagramme de séquence et états-transitions : Comparaison

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 séquencement 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 séquencement 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 séquencement d'occurrence des événements, nous constatons que le séquencement 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 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.

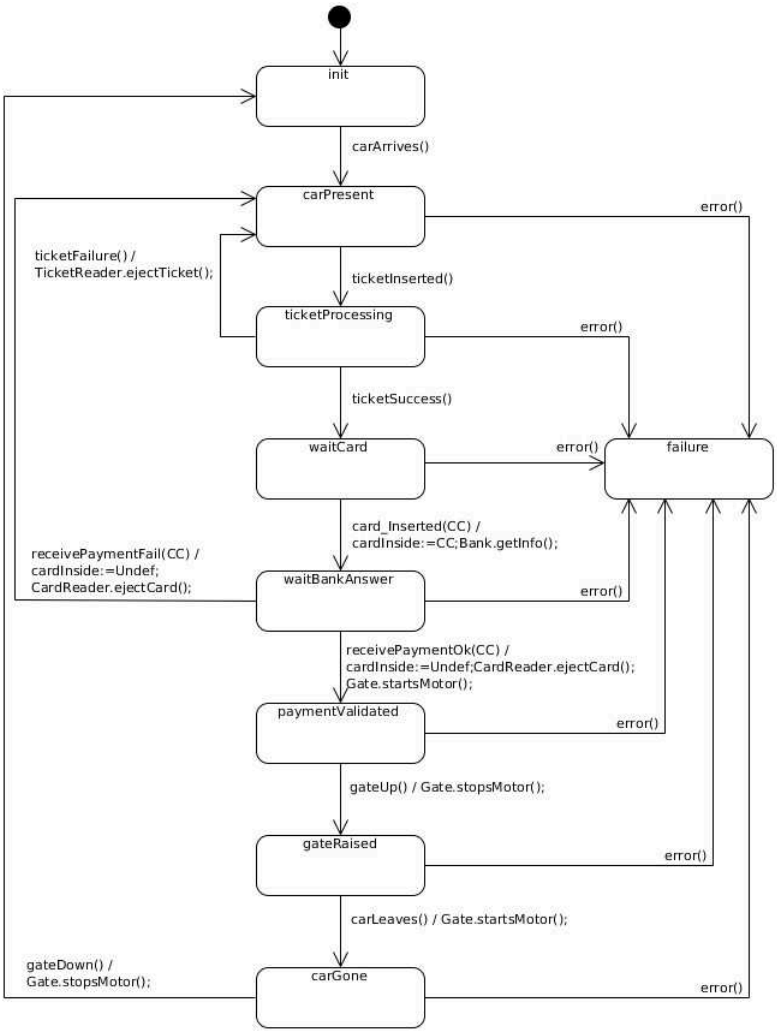


FIGURE 4.17 – Diagramme états-transitions de la barrière de péage : nouvelle version

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

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

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

Condition

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

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

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

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

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

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

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

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

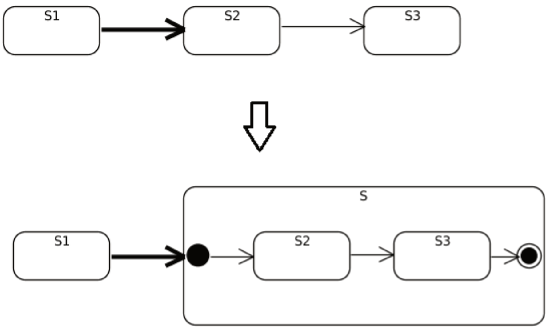


FIGURE 4.18 – Exemple de transition vers état simple dans un état composite

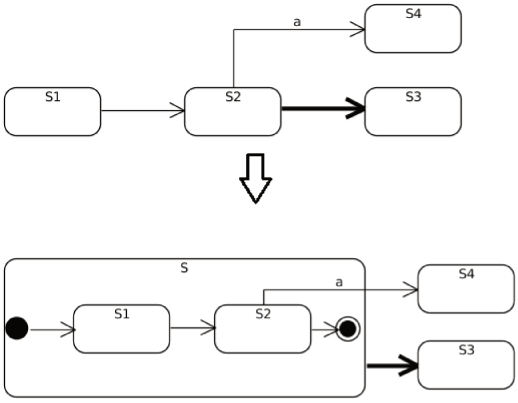


FIGURE 4.19 – Exemple de transition (avec ou sans événement) originaire d'un état simple dans un état composite

Exemple La Figure 4.21 montre le nouveau diagramme états-transitions avec introduction de la hiérarchie dans le cas du paiement et de deux autres états composites. Dans la Figure 4.21 il y a un état composite global `Success` qui modélisera la partie de notre système avec les cas de succès. Cet état permet d'éviter d'avoir des transitions avec l'événement `error` issues de tous les états. 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

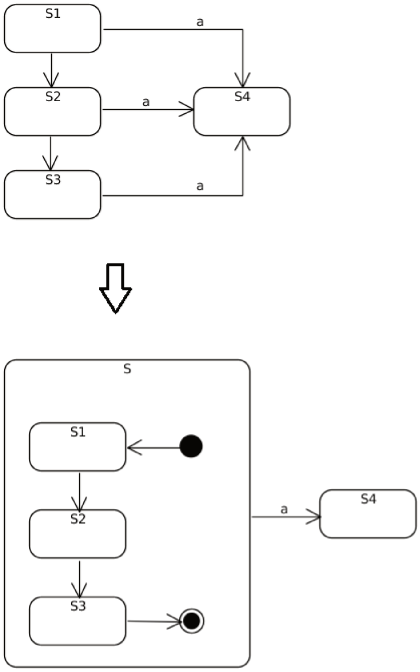


FIGURE 4.20 – Exemple d'application de la règle 3 sur un diagramme états-transitions

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

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

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

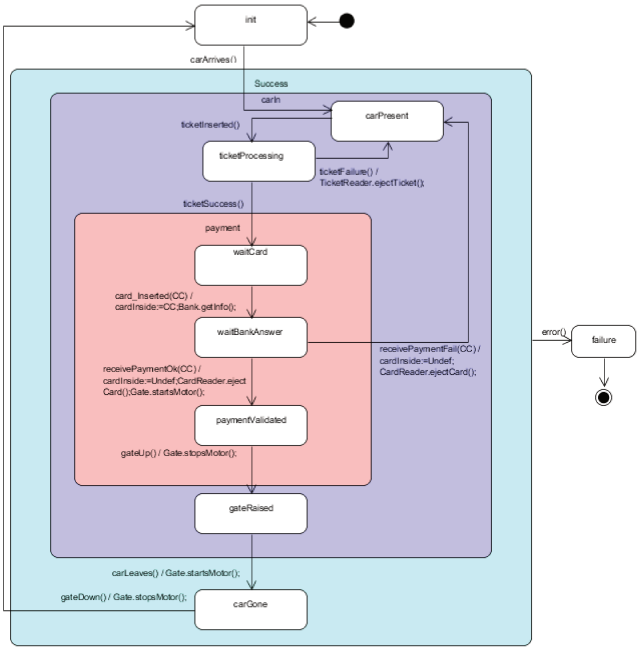


FIGURE 4.21 – Diagramme états-transitions de la barrière de péage avec hiérarchie

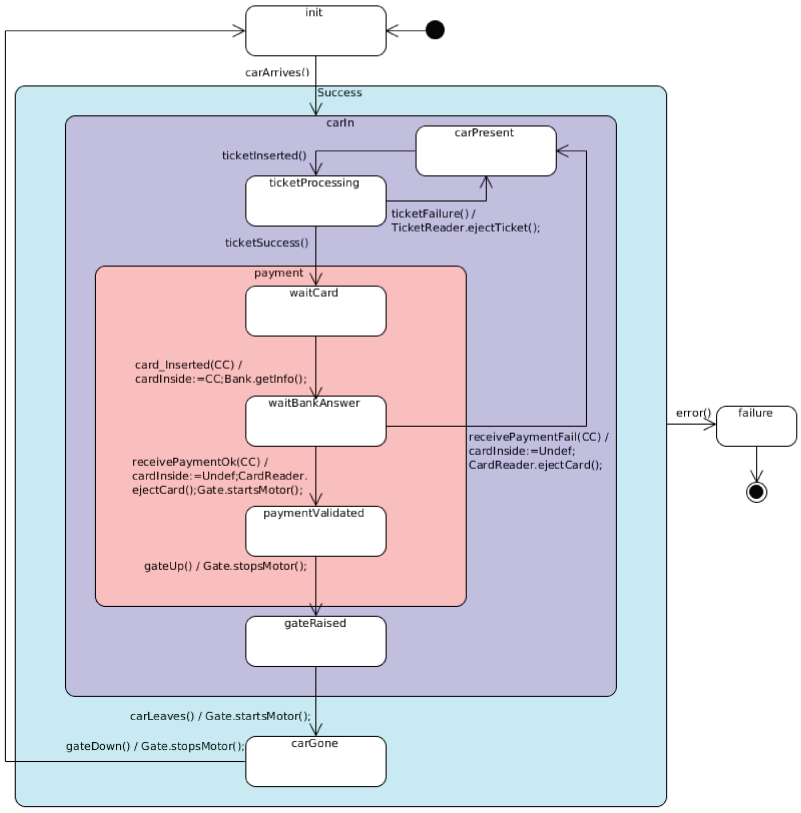


FIGURE 4.22 – Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)

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

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

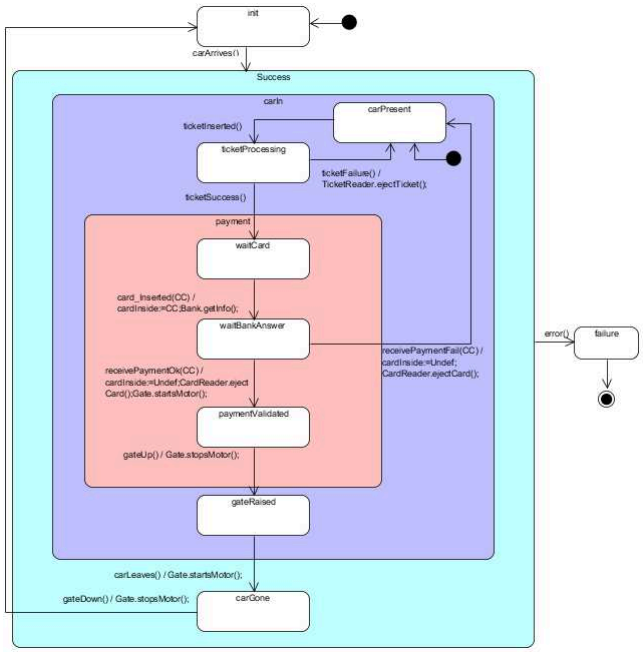


FIGURE 4.23 – Diagramme états-transitions de la barrière de péage avec hiérarchie (version erronée)

Qualité du diagramme états-transitions

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

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

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

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

Métriques

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

Il existes différents facteurs de qualités et métriques qui permettent l'évaluation d'un diagramme états-transitions. Le travail de [**?**] regroupe des métriques de la littérature le plus utilisées parmi eux :

Métriques basiques

- *États* : le nombre d'états d'un diagramme états-transitions
- *Actions (activités) des états* : le nombre d'activités définit pour les états dans un diagramme états-transitions
- *Transitions* : nombre de transitions dans un diagramme états-transitions
- *Gardes des transitions* : nombre de transitions avec une garde dans un diagramme états-transitions
- *événements des transitions* : nombre des événements des transitions dans un diagramme états-transitions

Métriques complexes

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

Smells

- *Pseudo-état initial et état final*
- *État sans nom*
- *Pas de transitions entrantes pour un état*
- *Pas de transitions sortantes pour un état*
- *Pas de gardes dans un pseudo-état choice*

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

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

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

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

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

Critères de qualité

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

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

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

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

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

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

Améliorations

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

Suppression de la duplication La présence des mêmes [1] transitions en sortie allant vers le même état permet de détecter les états dupliqués et ainsi de les supprimer. Nous distinguons alors deux cas : le cas où les états dupliqués ont les mêmes transitions en entrée et le cas où les états dupliqués ont des transitions en entrée différentes.

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

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

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

Introduction des transitions avec plusieurs événements Pour chaque ensemble de transitions ayant la même source et la même destination avec des événements, nous remplaçons cet ensemble par une seule transition ayant plusieurs événements. Cette situation est envisageable uniquement dans le cas ou le regroupement des événements dans une seule transition ne complique 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

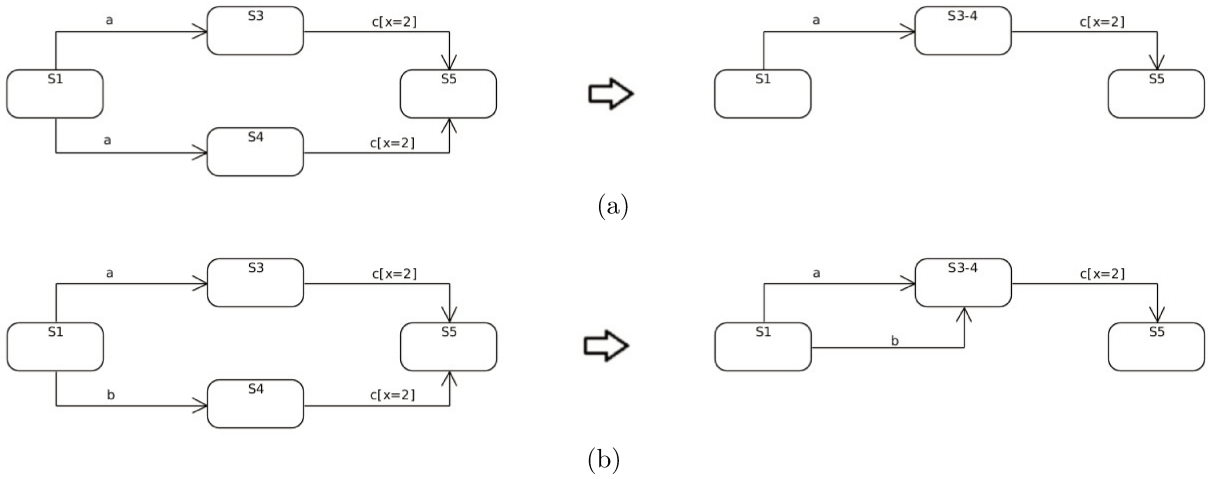


FIGURE 4.24 – Exemple de diagramme états-transitions avec et sans amélioration de la duplication

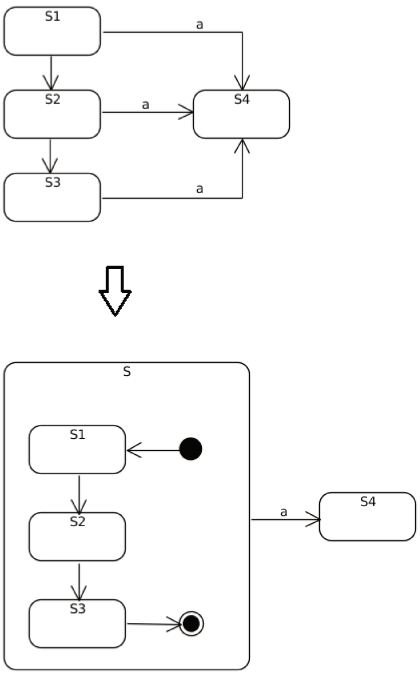


FIGURE 4.25 – Exemple d'introduction de la hiérarchie

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

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

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

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

Pseudo-états initiaux et états finaux Pour chaque état composite ayant un ensemble de transitions (avec ou sans événements) en entrée (l'ensemble doit contenir au moins une transition), nous rajoutons un pseudo-état initial allant vers son état initial. Dans la Figure 4.27 nous 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

Pour chaque état composite ayant un ensemble de transitions sans événements en sortie, nous rajoutons un état final lié au dernier état qui sera visité dans l'état composite. Dans la 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

Expérimentation

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

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

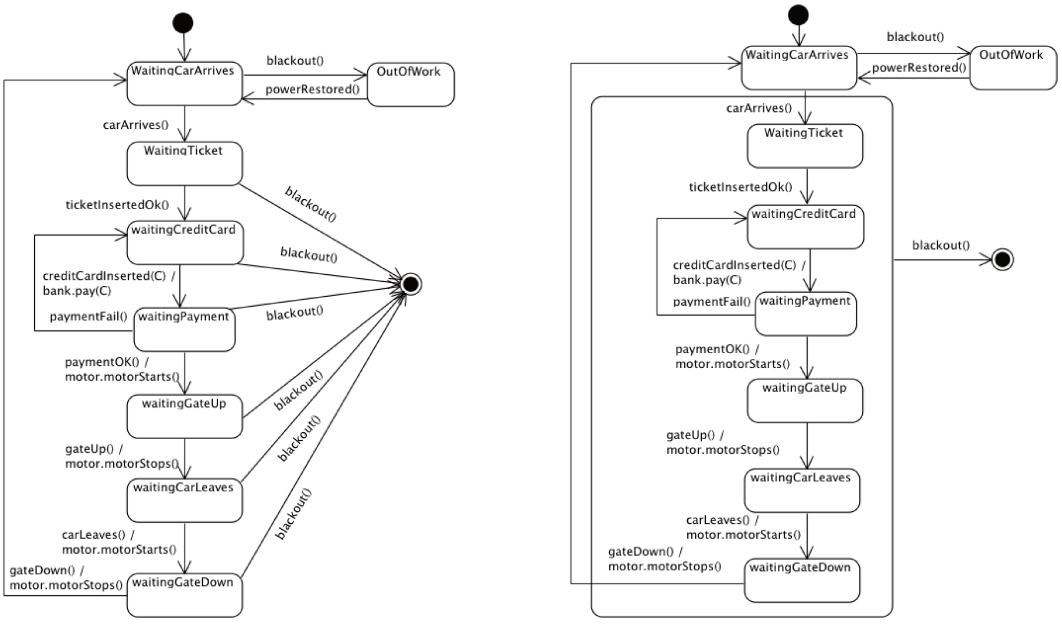
TABLE 4.18 – Exemples utilisés dans l'expérimentation

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 seul 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.



(a) Sans hiérarchie (b) Avec hiérarchie

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

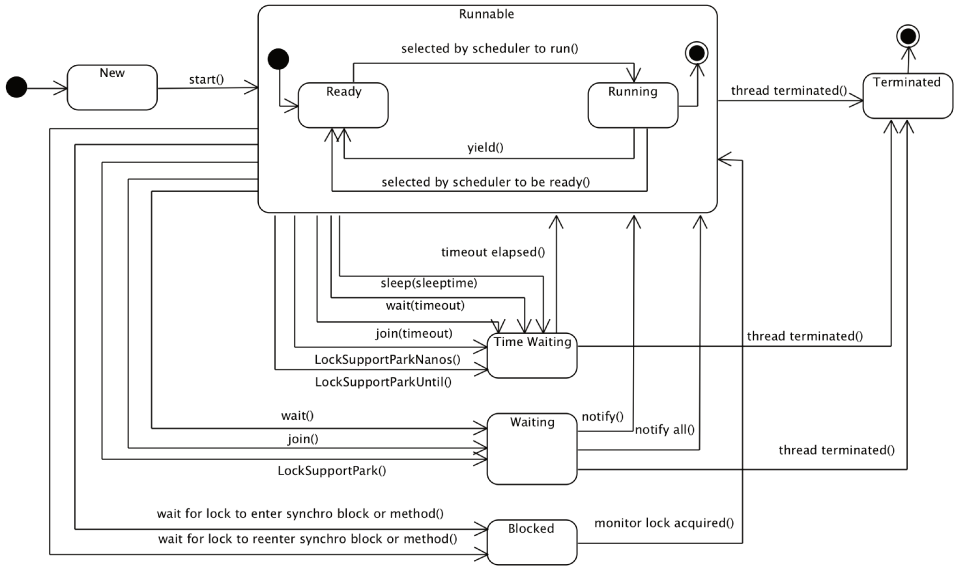


FIGURE 4.30 – Exemple du thread avec des événements dans des transitions différentes

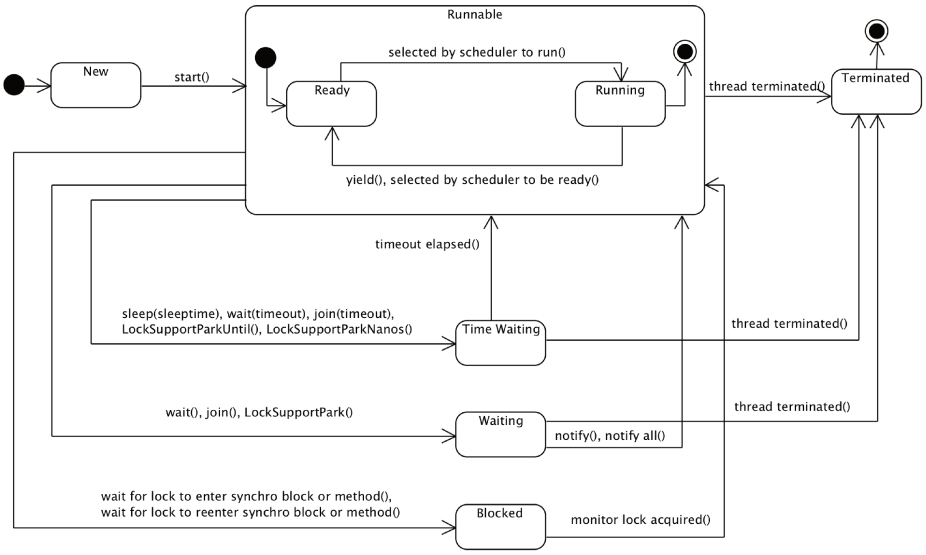


FIGURE 4.31 – Exemple du thread avec des événements dans une seule transition

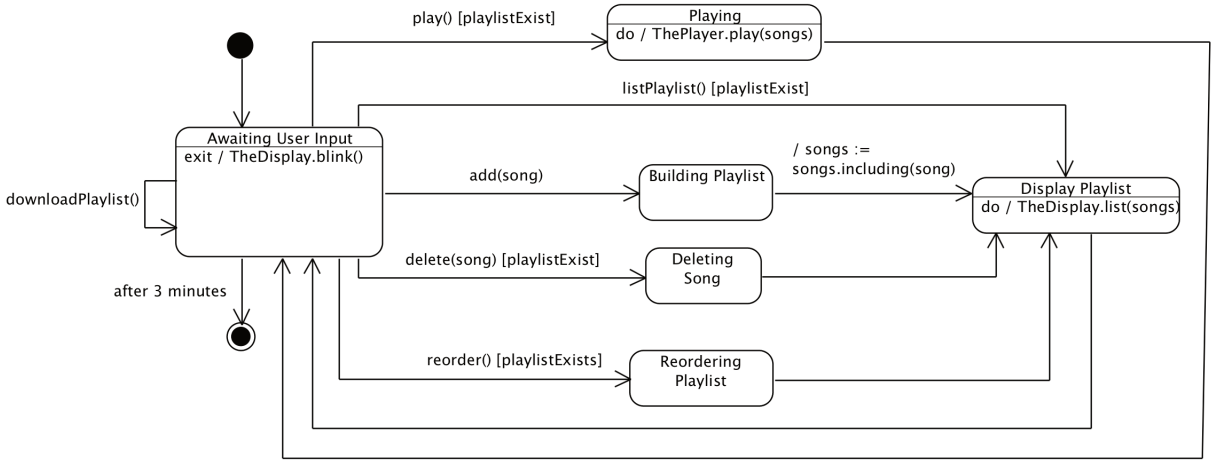


FIGURE 4.32 – Exemple de la playlist avec des états dupliqués

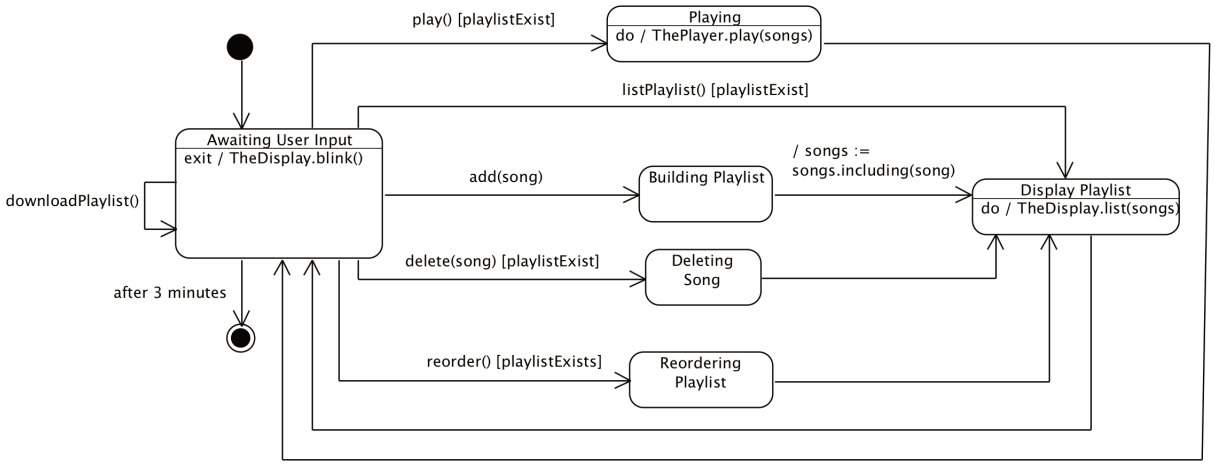


FIGURE 4.33 – Exemple de la playlist sans états dupliqués

Outil support de la méthode : *EasySM*

EasySM : première version

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

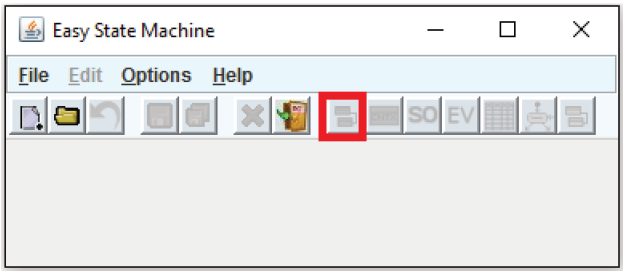


FIGURE 4.34 – Interface de base de *EasySM*

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

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

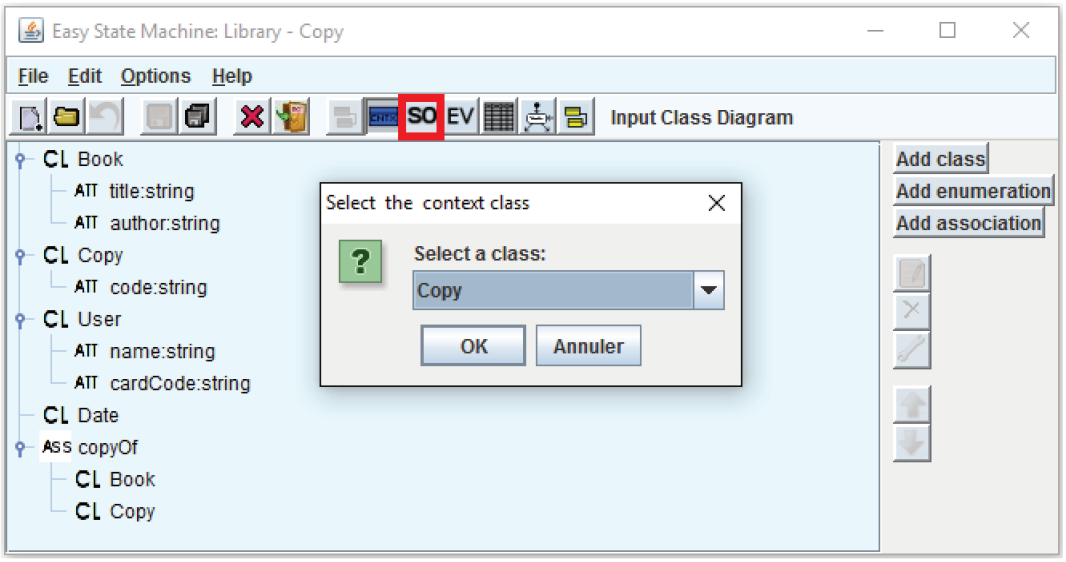


FIGURE 4.35 – Édition du diagramme de classes et de la classe de contexte dans *EasySM*

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

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

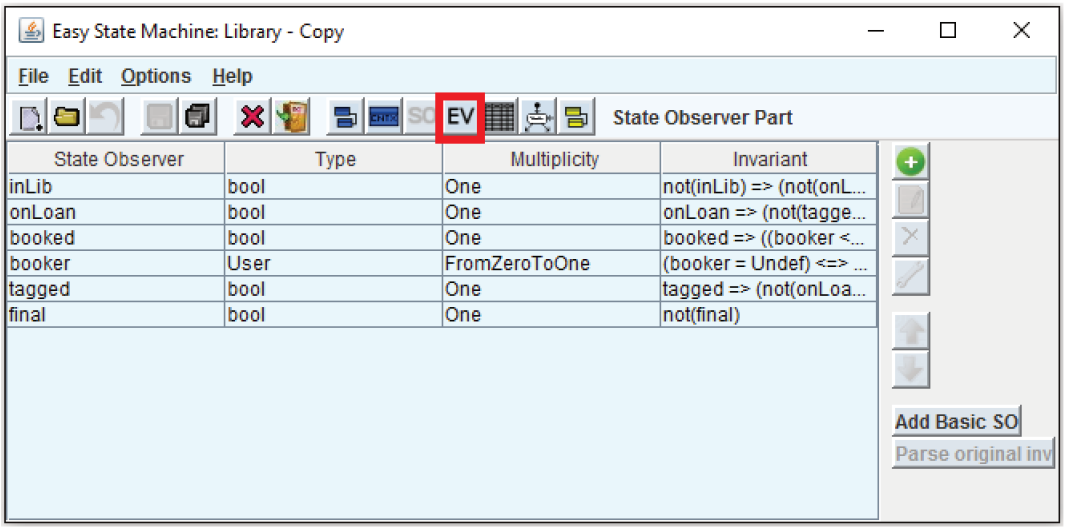


FIGURE 4.36 – Édition des observateurs d'états dans *EasySM*

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

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

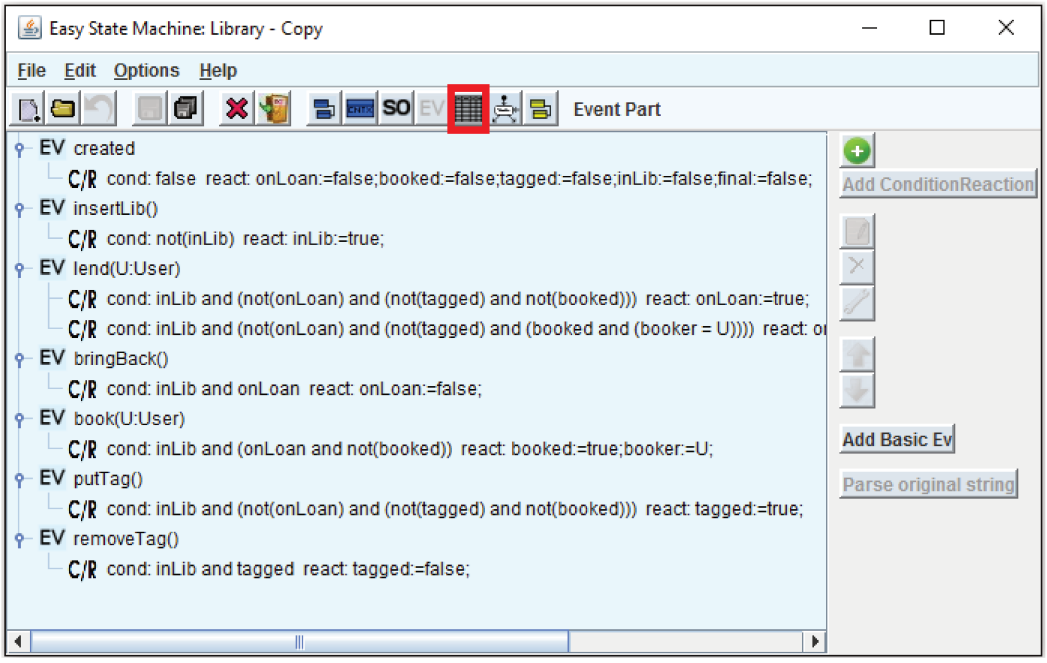


FIGURE 4.37 – Édition des événements dans *EasySM*

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

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

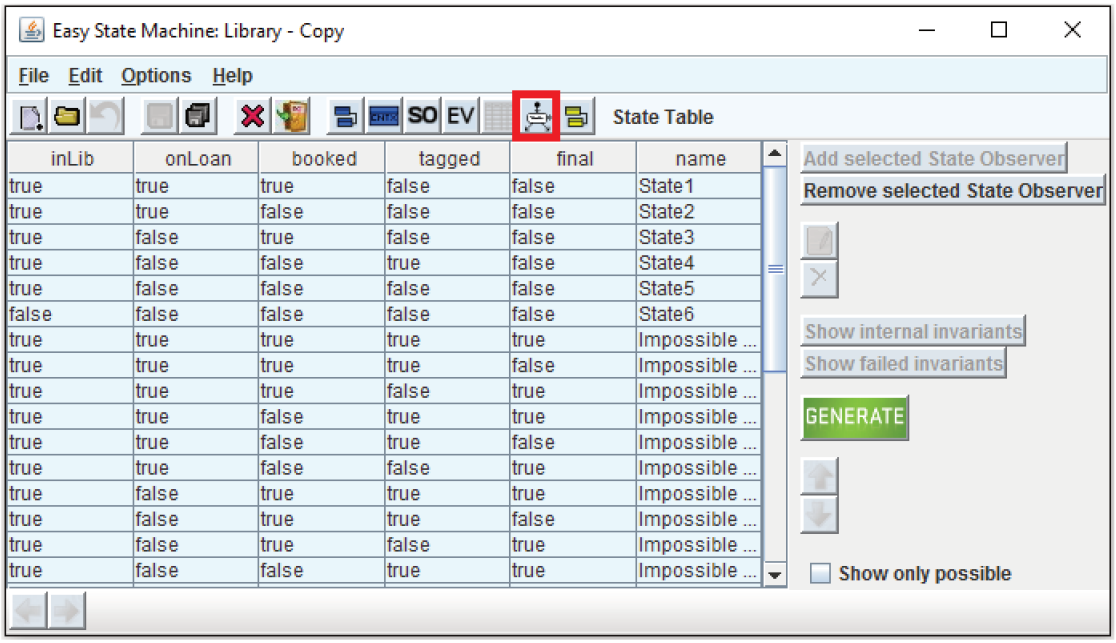
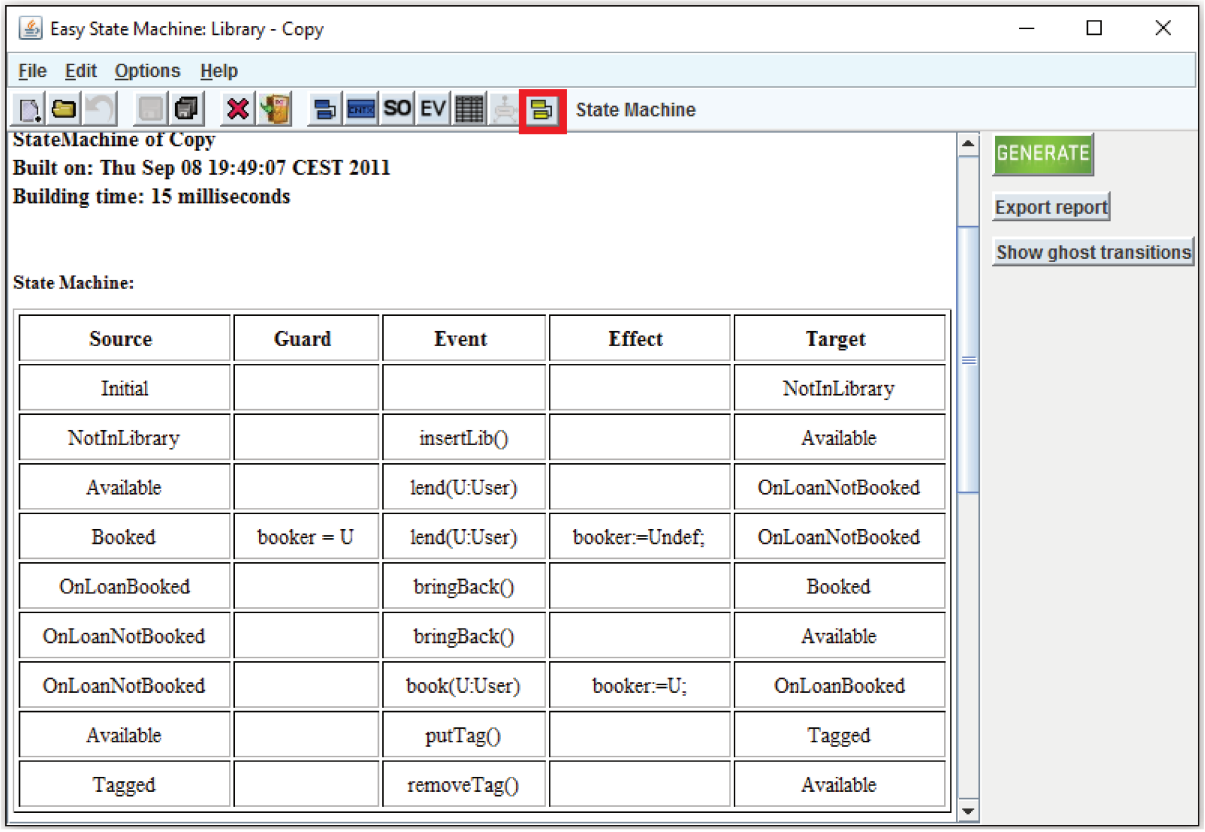


FIGURE 4.38 – Édition de la table des états dans *EasySM*

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



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*

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

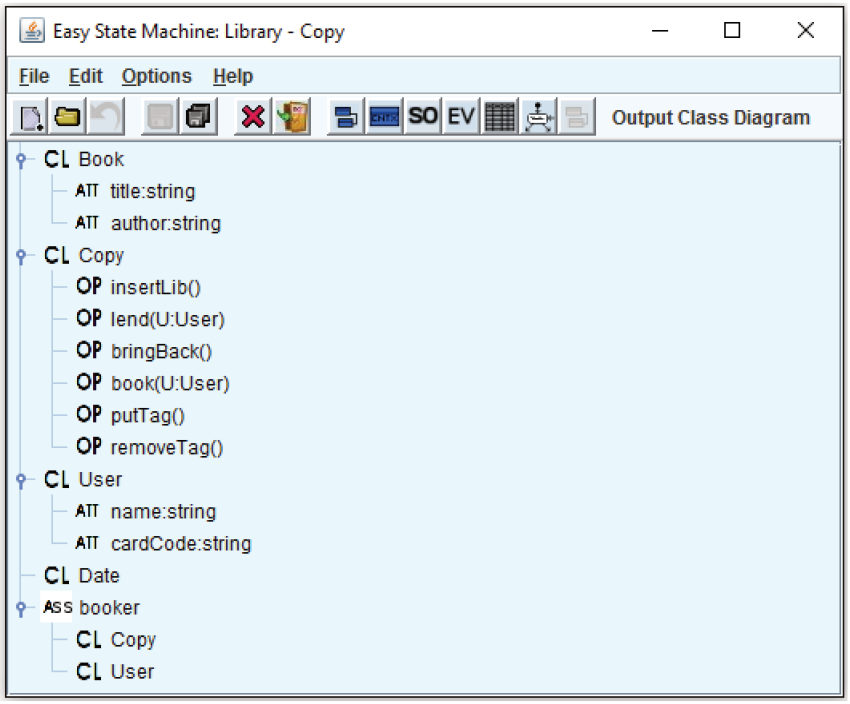


FIGURE 4.40 – Mise à jour du diagramme de classes dans *EasySM*

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 contenants 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.

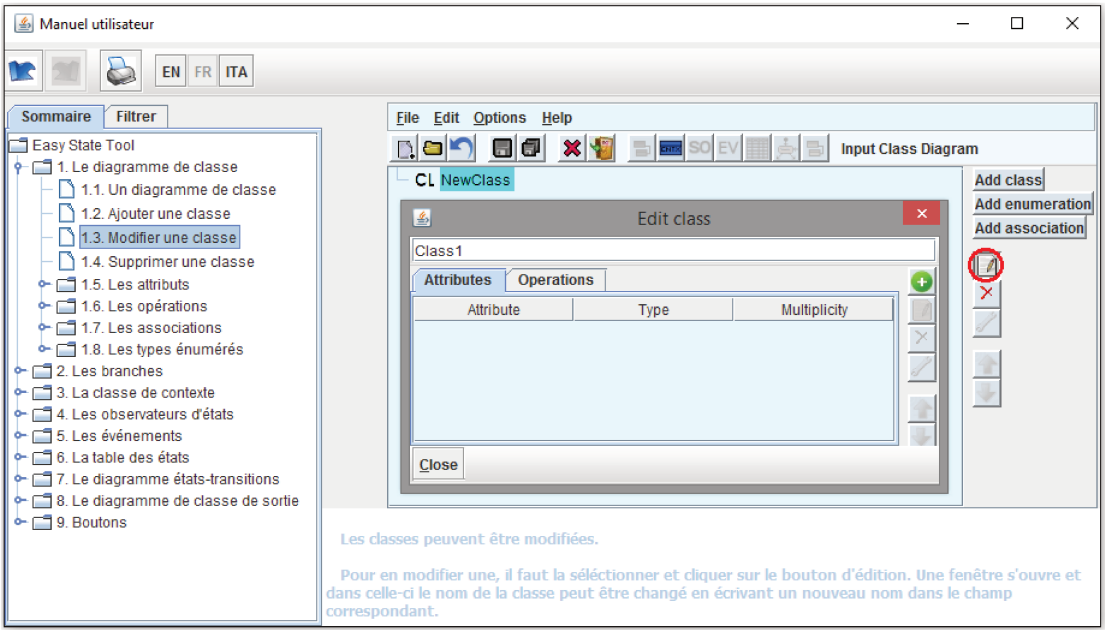


FIGURE 4.41 – Manuel utilisateur dans*EasySM*

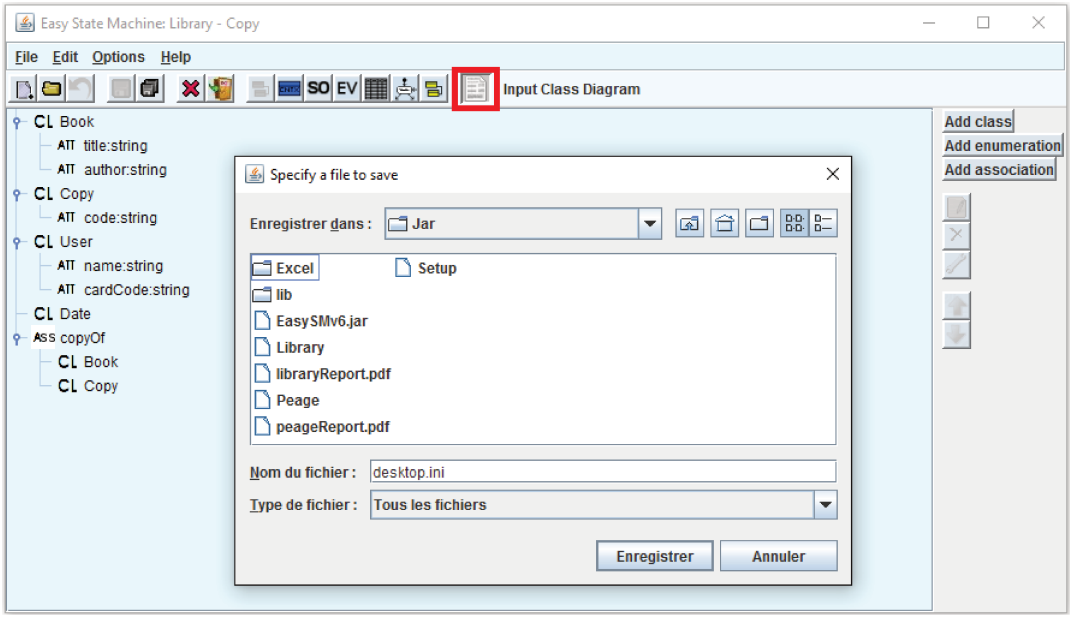


FIGURE 4.42 – Le bouton *Export Rapport* dans *EasySM*

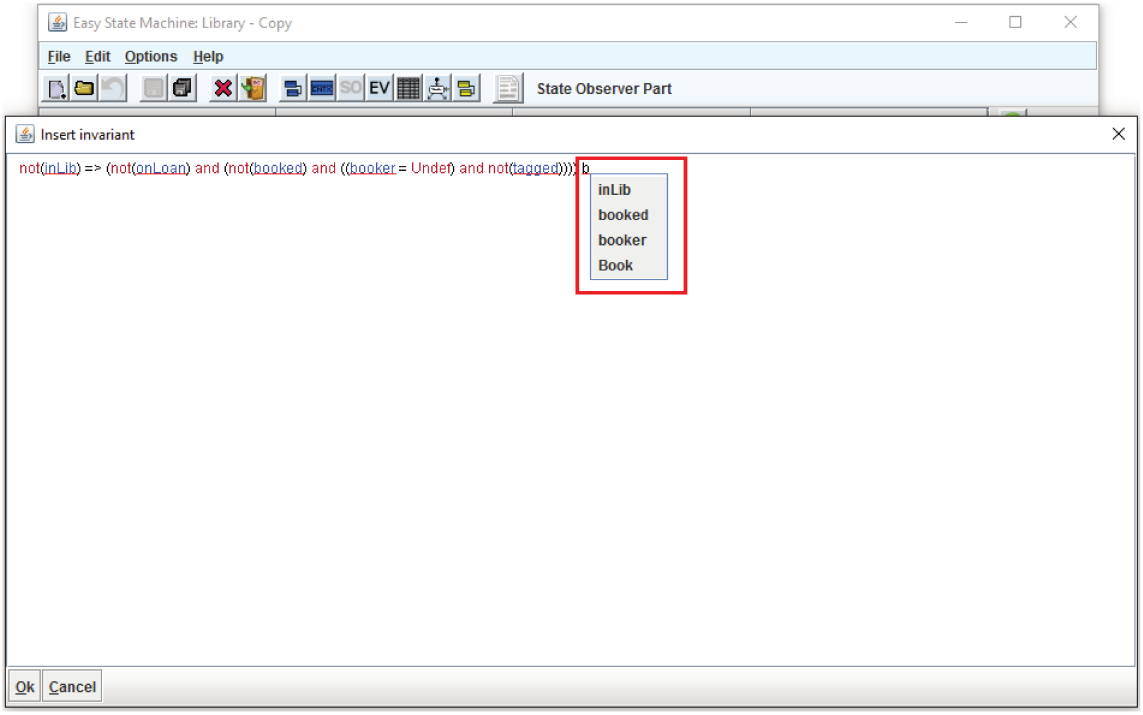


FIGURE 4.43 – Menu déroulant des propositions de types dans *EasySM*

Conclusion

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

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

Chapitre 5

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

Contents

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.

Formalisation et hypothèses

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

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

Notez que nous discuterons de comment supprimer les hypothèses ainsi que comment prendre en compte le reste des éléments dans la section Section 5.8.

États Nous prenons en compte deux types d'états : simples et composites. Les états submachine ne sont pas considérés dans notre traduction.

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

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

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

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

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

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

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

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

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

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

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

Nous réutilisons le concept de [**?**] d'état niveau (ou *level state* noté par *sLevel*). L'état niveau d'une transition allant de $\mathbf{s_1}$ à $\mathbf{s_2}$ est l'état le plus intérieur dans la structure hiérarchique du diagramme états-transitions qui contient la transition. Par exemple, l'état niveau de la transition 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*
$$SMD = (\mathcal{S}, \mathcal{B}, \mathcal{E}, \mathcal{V}, \mathcal{P}, \mathcal{N}, \mathcal{X}, \mathcal{D}, SubStates, \mathcal{T}),\ où$$

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. $\mathcal{V} = \{V_1, \ldots, V_{N_V}\}$ *est l'ensemble des variables* N_V *(pour certaines valeurs* $N_V \in \mathbb{N}$*),*
5. $\mathcal{P} : \mathcal{S} \to Pr$ *associe à chaque état une seule propriété dans* $Pr = \{isSimpleNotFinal, isComposite, isFinal, isHistory\}$*,*
6. $\mathcal{N} : \mathcal{S} \to ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ *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. $\mathcal{X} : \mathcal{S} \to ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ *associe pour chaque état un comportement de sortie,*
8. $\mathcal{D} : \mathcal{S} \to ((\mathcal{B} \cup \{none\}) \times List(\mathcal{F}))$ *associe pour chaque état un comportement do,*
9. $SubStates : \mathcal{S} \to 2^{\mathcal{S}}$ *associe pour chaque état un ensemble de sous-états directs,*
10. $\mathcal{T}$ *est l'ensemble des transitions de la forme* $\mathbf{t} = (\mathbf{S}_1, \mathrm{e}, g, (b, f), sLevel, \mathbf{S}_2)$*, 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* $\mathbf{S}_2$*) ;*

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

Notez que, dans la plus part des cas, *sLevel* de `t` est $LCA(\texttt{S1}, \texttt{S2})$, tel que "The operation LCA(s1,s2) returns the Region that is the least common ancestor of Vertices s1 and s2, based on the StateMachine containment hierarchy." ([?, p.356]). Un exemple où la fonction *LCA* ne couvre pas tous les cas est la transition entre l'état `S1` et son pseudo-état histoire `H` dans la Figure 2.1. En effet, la fonction $LCA(\texttt{S1}, \texttt{H})$ renvoie `S1`, tandis que le résultat doit être la machine à états `S`. La fonction *sLevel* pour la même transition (dans notre traduction) renvoie `S` (la machine à états), car *sLevel* permet de capturer le cas d'une transition externe.

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

Schéma général

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

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

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

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

Variables 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*.

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

- `S1_a_S2_1` pour la combinaison <`S11` , `S131`>
- `S1_a_S2_2` pour la combinaison <`S11` , `S132`>
- `S1_a_S2_3` pour la combinaison <`S11` , `S13_1F`>
- `S1_a_S2_4` pour la combinaison <`S11` , `S13_2F`>
- `S1_a_S2_5` pour la combinaison <`S11` , `S14`>
- `S1_a_S2_6` pour la combinaison <`S11` , `S1_2F`>
- `S1_a_S2_7` pour la combinaison <`S12` , `S131`>
- `S1_a_S2_8` pour la combinaison <`S12` , `S132`>
- `S1_a_S2_9` pour la combinaison <`S12` , `S13_1F`>
- `S1_a_S2_10` pour la combinaison <`S12` , `S13_2F`>
- `S1_a_S2_11` pour la combinaison <`S12` , `S14`>
- `S1_a_S2_12` pour la combinaison <`S12` , `S1_2F`>
- `S1_a_S2_13` pour la combinaison <`S1_1F` , `S131`>
- `S1_a_S2_14` pour la combinaison <`S1_1F` , `S132`>
- `S1_a_S2_15` pour la combinaison <`S1_1F` , `S13_1F`>
- `S1_a_S2_16` pour la combinaison <`S1_1F` , `S13_2F`>
- `S1_a_S2_17` pour la combinaison <`S1_1F` , `S14`>
- `S1_a_S2_18` pour la combinaison <`S1_1F` , `S1_2F`>

Soit `t` une transition *SMD*, notons par *combinations*(**t**) la fonction qui calcule toutes les combinaisons possibles des états sources. Chaque combinaison est traduite par une transition *CPN*. Par exemple, considérons la transition de `S1` vers `S2` étiquetée par l'événement `b` dans la Figure 5.6a. Les états `S11` et `S12` forment une combinaison ; par conséquent, une transition *CPN* est créée et sera notée `S1_b_S2_1` dans la Figure 5.6b (`1` dénote que c'est la première combinaison). Dans le cas d'une transition *SMD* sans événement, nous la traduisons en une seule transition *CPN*, ce qui permet d'éviter l'explosion combinatoire dans ce cas : en effet, afin de sortir d'un état composite (avec une ou plusieurs régions) en utilisant une transition de terminaison, il est nécessaire que chaque région soit dans son état final, ce qui implique une seule 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`	place	s_iF
	Dans un état composite `s`	place	sF
	État final racine	place	F
État simple `s`	-	place	s
Pseudo-état histoire	Dans une seule région	place	Sh
	Dans plusieurs régions	place	S_ih
Comportement `do` d'un état `s`	-	transition	s_do
Transition sans événement	Une seule transition entre `S1` & `S2`	transition	S1_S2
	Plusieurs transitions entre `S1` & `S2`	transition	S1_S2_i
Transition avec événement `e`	Une seule transition entre `S1` & `S2`	transition	S1_e_S2_j
	Plusieurs transitions entre `S1` & `S2`	transition	S1_e_S2_i_j
Transition vers un pseudo-état histoire	Sans événement	transition	S1_H_S2
	Avec événement `e`	transition	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 $\mathbf{t}$, nous utilisons la fonction *combinations*($\mathbf{t}$) afin d'engendrer les combinaisons possibles, et nous ajoutons les transitions correspondantes (voir Section 5.3). Rappelons que la fonction *combinations*($\mathbf{t}$) 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*

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

Troisième étape La dernière étape (ligne 19–ligne 26) traite les transitions ayant comme cible un pseudo-état histoire. Rappelons que nous excluons la possibilité d'avoir une transition fork qui cible un pseudo-état histoire, par conséquent l'état cible est nécessairement unique. En revanche, la source de la transition peut être multiple, et un ou plusieurs état(s) source(s) peut être un état composite, ce qui nécessite d'énumérer les différentes combinaisons des états sources (ligne 20). Ensuite, pour chaque pseudo-état histoire cible ("$\mathbf{s_2}$"), c'est-à-dire que pour chaque état non final dans la région du pseudo-état histoire, nous ajoutons une transition dédiée (ligne 22). Cette transition *CPN* aura une garde (avec la fonction *DecorateH*()), cela implique qu'elle peut être franchie uniquement si $\mathbf{s_2}$ est le dernier état visité dans cette région. Cette transition a comme source les places encodant tous les états simples de la combinaison considérée (ligne 24), et comme cible les places encodant tous les états initiaux de la cible du pseudo-état histoire (ligne 26) ; en effet, il est possible d'avoir plusieurs places dans le cas où la cible du pseudo-état histoire est un état composite.

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

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

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

si s2 est le dernier état visité dans la région qui contient le pseudo-état histoire. En second, pour chaque région S qui contient un pseudo-état histoire, pour chaque sous-état direct s1 de S, pour chaque sous-état simple indirect de s1, nous ajoutons le code suivant à la transition *CPN* qui conduit à la place encodant cet état simple (le sous-état indirect de s1) : Sh := s1. Cela permet d'enregistrer le dernier état visité dans la région S qui est s1. Une exception est quand s1 est un état final, dans cette situation, le code de la transition n'est pas Sh := s1, mais plutôt Sh := si, où si est l'état destination du pseudo-état initial de la région. Cela est en accord avec le fait que, si nous quittons un état composite à travers son état final, et nous rentrons à travers son pseudo-état histoire, alors l'entrée sera par son entrée par défaut, c'est à dire le pseudo-état initial. Nous donnerons des exemples de la gestion des pseudo-états histoire dans la Section 5.7.6.

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

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

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

// Step 1: Simple states and their do behaviours

foreach *simple state* $\mathbf{s} \in \mathcal{S}$ **do**
  Add (s)
  if $\mathbf{s}$ *has a do behaviour* **then**
    Add (s) ⟲ s_do

// Step 2: Transitions

foreach *transition* $\mathbf{t} = (\mathbf{S}_1, \mathrm{e}, g, (b, f), \mathit{sLevel}, \mathbf{S}_2) \in \mathcal{T}$ *with no history pseudostate within* $\mathbf{S}_2$ **do**
  if $\mathrm{e} \neq \mathit{noEvent}$ **then**
    foreach $c \in \mathit{combinations}(\mathbf{t})$ **do**
      Add [S1_e_S2_i_j] // jth combination of the ith transition from $\mathbf{S}_1$ to $\mathbf{S}_2$ with e
      foreach *simple state* $\mathbf{s} \in c$ **do**
        Add (s) → [S1_e_S2_i_j]
      foreach *simple state* $\mathbf{s} \in \mathit{init}^*(\mathbf{S}_2)$ **do**
        Add [S1_e_S2_i_j] → (s)
  else
    Add [S1_S2_i] // ith transition without event from $\mathbf{S}_1$ to $\mathbf{S}_2$
    foreach *simple state* $\mathbf{s} \in \mathit{final}^*(\mathbf{S}_1)$ **do**
      Add (s) → [S1_S2_i]
    foreach *simple state* $\mathbf{s} \in \mathit{init}^*(\mathbf{S}_2)$ **do**
      Add [S1_S2_i] → (s)

// Step 3: History states

foreach *transition* $\mathbf{t} = (\mathbf{S}_1, \mathrm{e}, g, (b, f), \mathit{sLevel}, \mathtt{H}) \in \mathcal{T}$ **do**
  foreach $c \in \mathit{combinations}(\mathbf{t})$ **do**
    foreach *state* $\mathbf{s_2}$ *such that* $\mathbf{s_2}$ *is a non-final direct substate of* $\mathbf{r}(\mathtt{H})$ **do**
      Add [S1_He_s2_i_j] // jth combination of the ith transition from $\mathbf{S}_1$ to H with e
      foreach *state* $\mathbf{s} \in c$ **do**
        Add (s) → [S1_He_s2_i_j]
      foreach *simple state* $\mathbf{s} \in \mathit{init}^*(\mathbf{s_2})$ **do**
        Add [S1_He_s2_i_j] → (s)

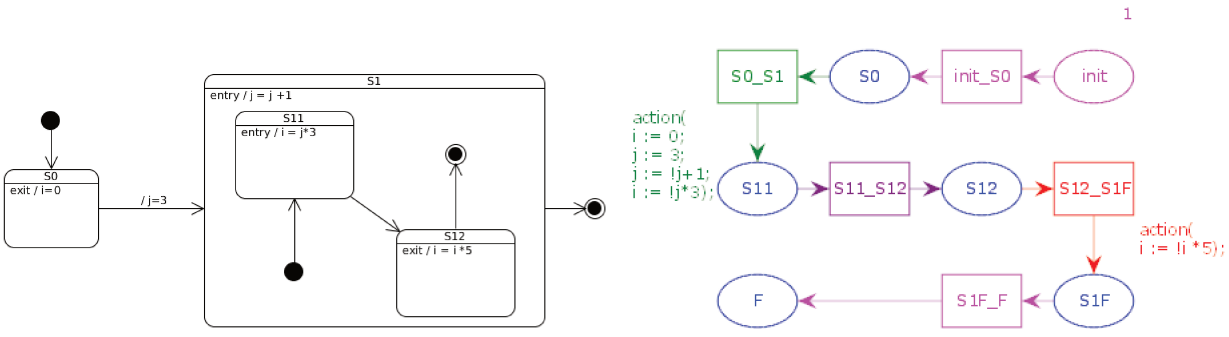
Add (init) → [init_S] // S is pointed by the initial pseudostate of *SMD*
foreach $\mathbf{s} \in \mathit{init}^*(\mathit{SMD})$ **do**
  Add [init_S] → (s)
AddGuards() ; *DecorateH*() ; *AddBehaviours*() ;

L'ajout des comportements au code des transitions

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

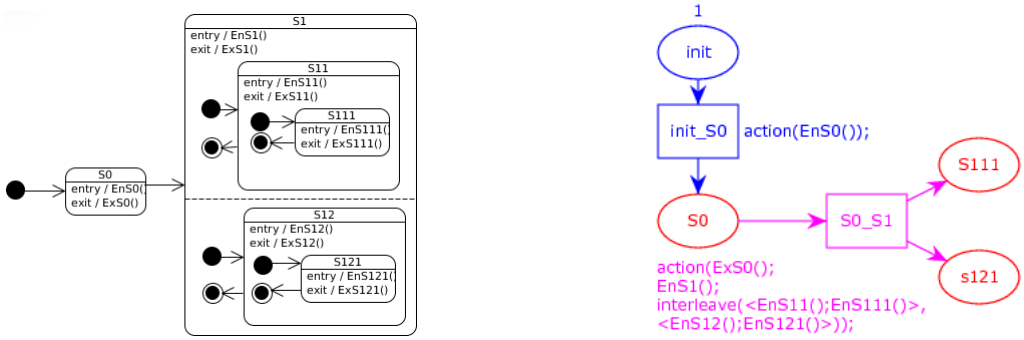
Comportement Do Chaque comportement est encodé avec une transition *CPN* liée avec la place qui encode l'état correspondant (ligne 4 de l'Algorithme 1). Par conséquent, la fonction *AddBehaviours*() ajoutera à la partie code de cette transition l'équivalent de la fonction f. Par exemple, le comportement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

Comportement d'entrée et de sortie Une difficulté majeure dans la formalisation des *SMD* est de traiter correctement les comportements d'entrée et de sortie lors du franchissement des transitions *SMD*. La fonction *AddBehaviours*() est responsable de la décoration de la partie code des transitions *CPN* afin de prendre en compte tous les comportements d'entrée et de sortie pendant le franchissement d'une transition *SMD*. Rappelons que, pendant le franchissement d'une transition *SMD*, les comportements de sortie doivent être exécutés en premier, venant des états les plus intérieurs de l'état source (ou des états source si la transition est un "fork") jusqu'à l'état qui contient la transition (nommé ici par "état niveau" ou "level state" *sLevel* dans la Définition 2), le comportement de la transition sera ensuite exécuté. Enfin, les comportements d'entrée sont exécutés venant de l'état niveau jusqu'à l'état le plus interne ("innermost") de l'état(s) destination de la transition. Le mécanisme est relativement simple en l'absence de concurrence, c'est-à-dire que dans le cas où un état composite n'est pas orthogonal. Soit la transition de l'état `S0` vers l'état `S1` dans le *SMD* de la Figure 5.1a, pendant le franchissement de cette transition, le comportement de sortie de `S0` sera exécuté en premier (c'est-à-dire que $\mathtt{i} = 0$ doit être exécuté). Le comportement associé à la transition est ensuite exécuté ($\mathtt{j} = 3$), enfin le comportement d'entrée de l'état `S1` est exécuté (c'est-à-dire que $\mathtt{j} = 1$), suivi par le comportement de `S11` et `S12` (c'est-à-dire que $\mathtt{i} = \mathtt{i} + 1$ et $\mathtt{i} = 3$). En correspondance à l'exécution des comportements en franchissant 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*) : $\mathtt{i} := 0; \mathtt{i} := 3; \mathtt{j} := 1;$ *interleave*$(< \mathtt{i} :=!\mathtt{i} + 1 >, < \mathtt{i} := 3 >)$.



(a) SMD (b) CPN

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



(a) SMD (b) CPN

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

Cependant, l'exécution des comportement est plus compliquée dans les états composites orthogonaux. Soit la transition de `S0` vers `S1` dans le diagramme états-transitions de la Figure 5.2a. À la sortie de l'état `S0` et une fois que le comportement de la transition est exécuté, le comportement d'entrée de l'état `S1` sera exécuté. Ensuite, les comportements des états `S11` et `S12` seront exécutés d'une manière concurrente, ce qui est représenté par :

1. `exS0()` ; `enS1()` ; `enS11()` ; `enS111()` ; `enS12()` ; `enS121()`
2. `exS0()` ; `enS1()` ; `enS11()` ; `enS12()` ; `enS111()` ; `enS121()`
3. `exS0()` ; `enS1()` ; `enS11()` ; `enS12()` ; `enS121()` ; `enS111()`
4. `exS0()` ; `enS1()` ; `enS12()` ; `enS11()` ; `enS111()` ; `enS121()`
5. `exS0()` ; `enS1()` ; `enS12()` ; `enS11()` ; `enS121()` ; `enS111()`
6. `exS0()` ; `enS1()` ; `enS12()` ; `enS121()` ; `enS11()` ; `enS111()`

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

Une solution simple est d'énumérer toutes ces possibilités et de créer une transition *CPN* pour chaque séquence de comportements. Au lieu de cela, nous proposons une fonction *interleave*() qui est utilisée afin de modéliser tous les entrelacements, et cela en gardant notre représentation compacte. Soit une liste de comportements (d'entrée ou de sortie) $\texttt{exS1}, \ldots, \texttt{exSn}$, nous notons $< \texttt{exS1}, \ldots, \texttt{exSn} >$ la séquence des comportements exécutée dans cet ordre. Soit la liste de plusieurs séquences $seq_1, \ldots seq_n$, notre fonction $interleave(seq_1, \ldots seq_n)$ considère tous les entrelacements entre ces séquences de comportements. Autrement dit, dans une séquence donnée seq_i, les comportements sont exécutés d'une façon séquentielle ; mais l'ordre entre les comportements de seq_i et celui des autres séquences peut être arbitraire. En fait, cette fonction *interleave*() est du "sucre syntaxique" pour une liste de transitions d'un *CPN*, chaque transition contenant un entrelacement particulier. L'utilisation de cette fonction permet une vision plus claire et propre de nos figures [1].

1. Bien que nous utilisions cette fonction dans les captures d'écrans de CPN Tools, cette fonction

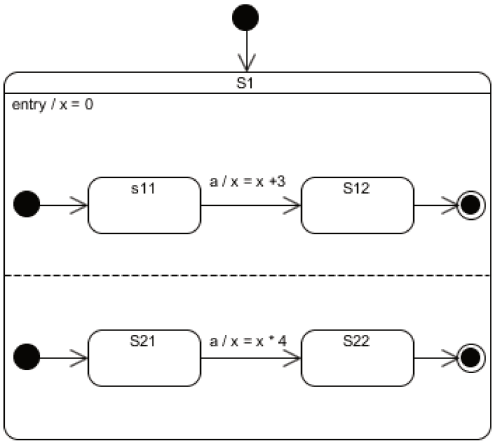


FIGURE 5.3 – Exemple avec modification concurrente des variables

Par exemple, l'entrelacement des comportements au-dessus peuvent être représentés comme suit : $\texttt{exS0}(); \texttt{enS1}(); \textit{interleave}(< \texttt{enS11}(); \texttt{enS111}() >, < \texttt{enS12}(); \texttt{enS121}() >)$

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

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

Application de l'algorithme à des cas simples

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.

États simples et comportements “Do”

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

(a) SMD (b) CPN

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

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

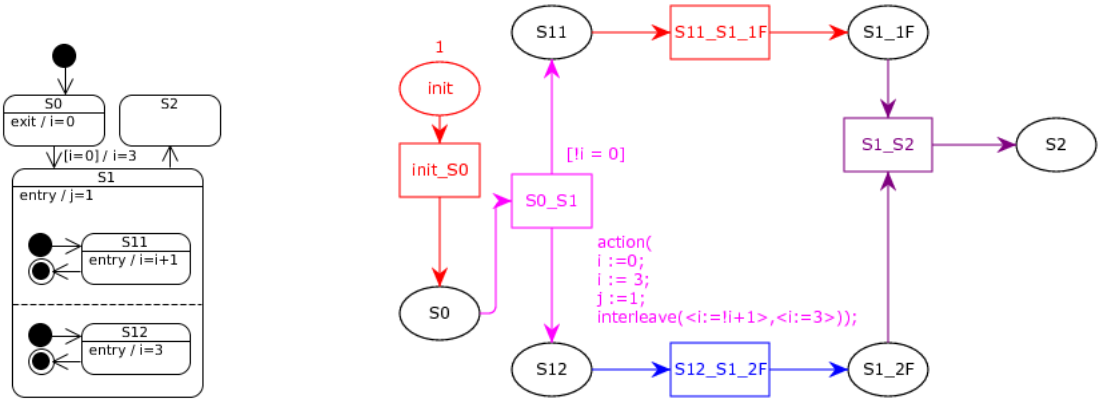
Transitions sans événements avec régions orthogonales

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

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

Transitions avec événement et régions orthogonales

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

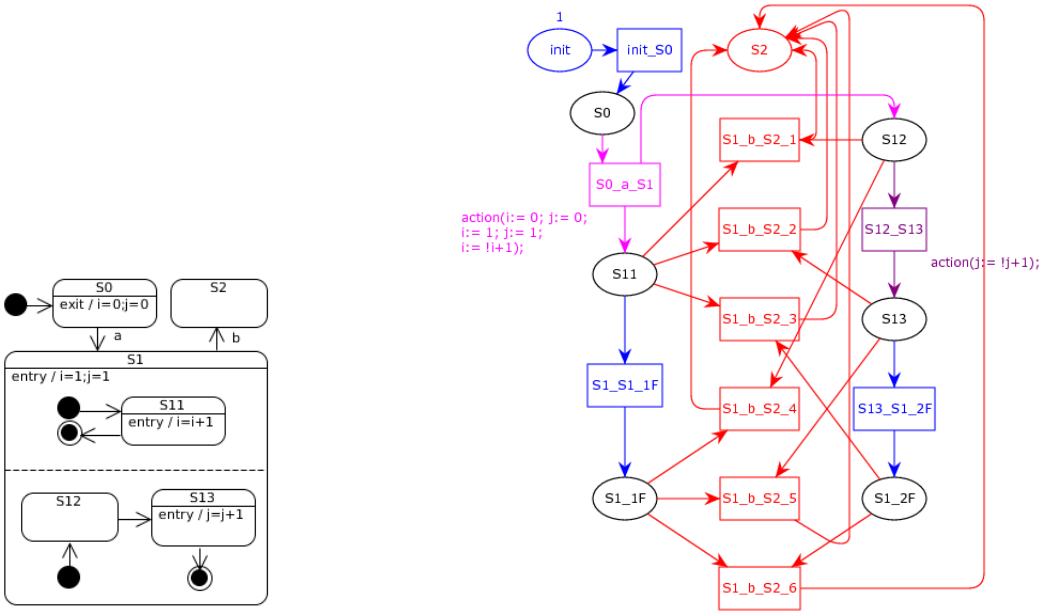


(a) SMD (b) CPN

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

Sortir de l'état `S1` à travers la transition étiquetée avec l'événement `b`, résulte aux combinaisons suivantes (le système peut être dans l'une de ces combinaisons) :

$$\{(\texttt{S11}, \texttt{S12}), (\texttt{S11}, \texttt{S13}), (\texttt{S11}, \texttt{S1\_2F}), (\texttt{S1\_1F}, \texttt{S12}), (\texttt{S1\_1F}, \texttt{S13}), (\texttt{S1\_1F}, \texttt{S1\_2F})\}.$$



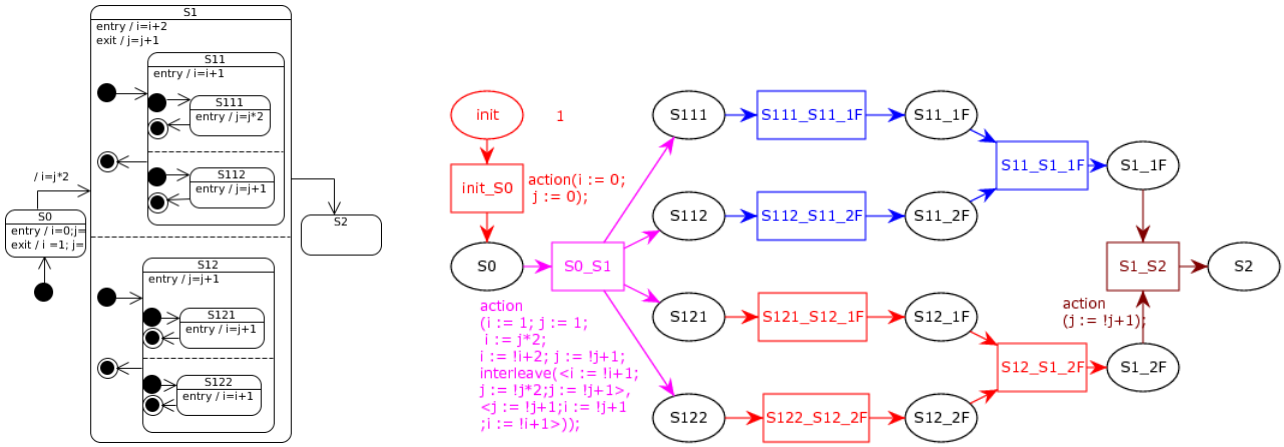
(a) SMD (b) CPN

FIGURE 5.6 – Exemple : transitions avec événements et régions orthogonales

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

Hiérarchie des régions orthogonales

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



(a) SMD (b) CPN

FIGURE 5.7 – Exemple : hiérarchie des régions orthogonales

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

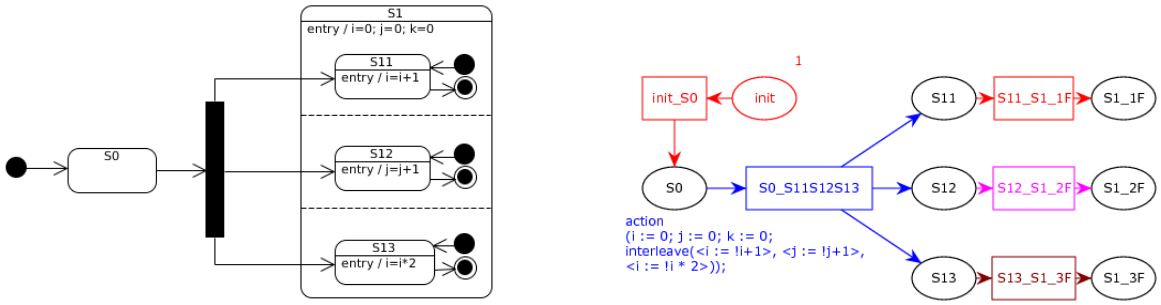
Pseudo-état fork

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

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

Pseudo-état histoire

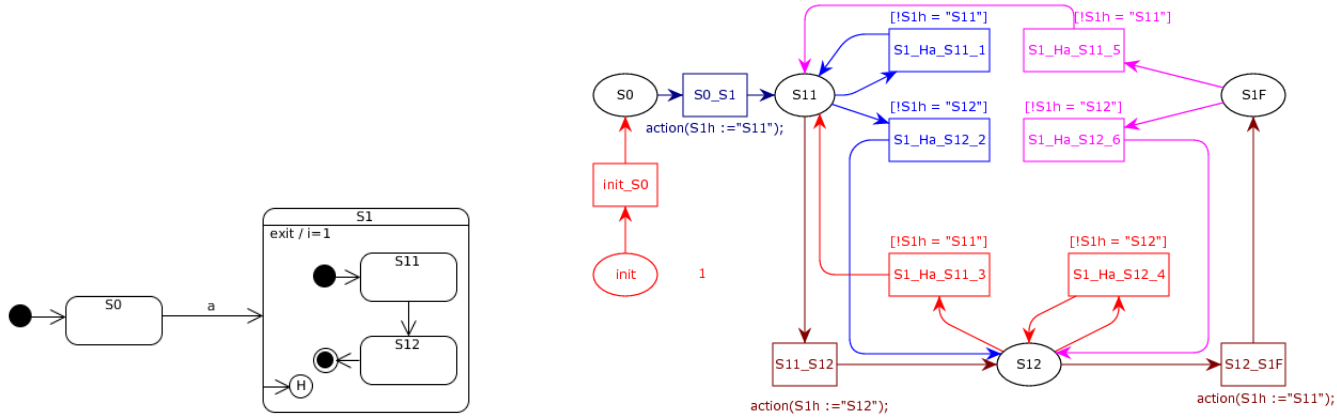
Situation considérée Nous considérons un pseudo-état histoire dans la Figure 5.9a. Rappelons que seulement les états composites peuvent avoir un pseudo-état histoire. La transition du



(a) SMD (b) CPN

FIGURE 5.8 – Exemple : pseudo-état fork

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



(a) SMD (b) CPN

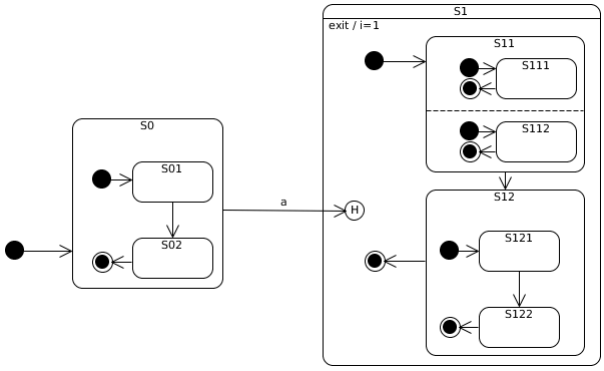
FIGURE 5.9 – Exemple : pseudo-état histoire

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

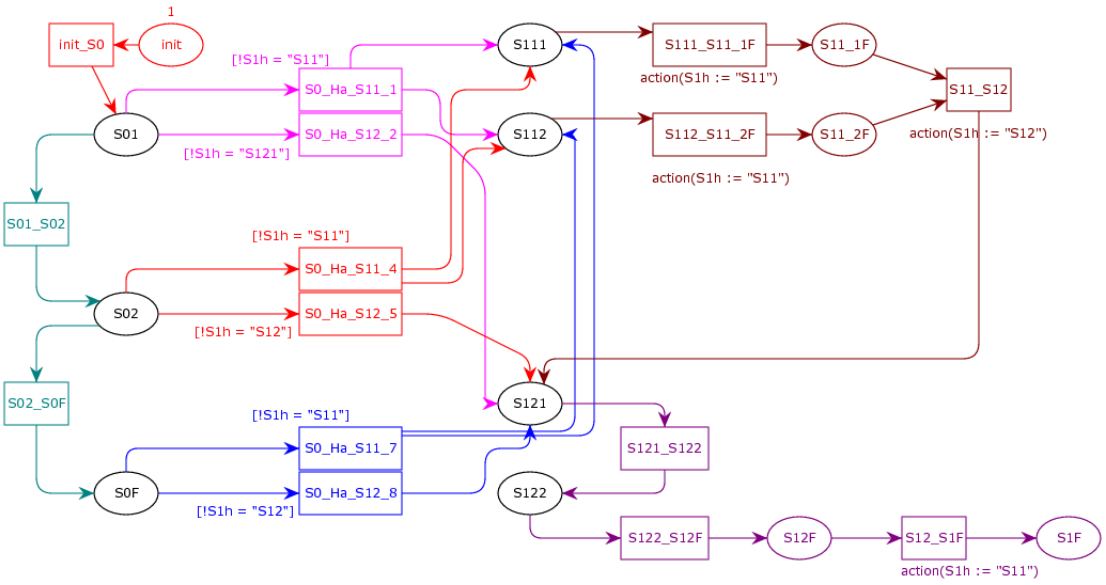
Dans la Figure 5.9b, la variable `S1h` est mise à ajour dans chaque transition qui conduit à la place CPN modélisant un sous-état de l'état `S1` (par exemple, le segment de code `S1h` := "`S11`" de la transition `S0_S1`).

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

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

Au-delà de notre traduction

Supprimer les hypothèses

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

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

Modélisation des expressions des comportements Rappelons que, bien que nous modélisons la modification des variables dans les comportements d'entrée et de sortie, nous ne prenons pas en compte dans notre traduction les expressions des comportements (par exemple, le comportement d'entrée de l'expression "`fade in`" dans l'état `PLAYING` dans la Figure 6.1) Cela nous empêche de vérifier par exemple des propriétés telles que "quand une piste est jouée (le comportement do `play track`), alors "fade in" (le comportement d'entrée `fade in`) est nécessairement apparu". Afin de considérer de telles propriétés, une variable globale booléenne fade in peut être ajoutée, qui sera mise à vrai lors de l'entrée dans la place correspondante à l'état `PLAYING`, et à faux lors de la sortie de cette la place. Par conséquent, vérifier cette propriété consiste à vérifier si la variable booléenne fade in est égale à vrai avant que la variable encodant le comportement do `play track` ne soit modifiée.

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

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

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

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

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

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

Extension de la syntaxe

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

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

Transitions internes Les transitions internes peuvent être exécutées dans le cas où leurs événements sont dispatchées et leurs gardes qui doivent être satisfaites. L'exécution des transitions 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 problème 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 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

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

Gestion des événements

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

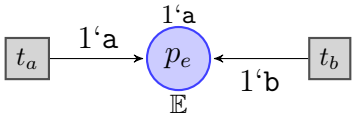


FIGURE 5.11 – Modélisation de la place des événements et de l'arrivée aléatoire des événements

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

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

Gestion des priorités

Aperçu

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

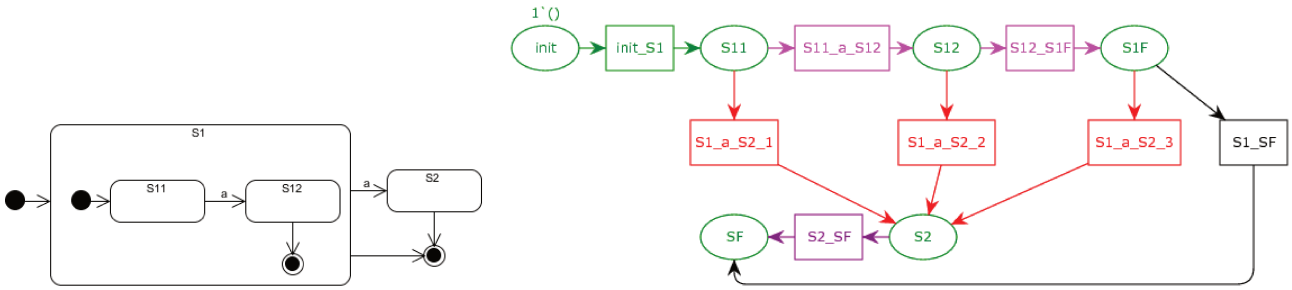
Afin de savoir si certaines transitions peuvent être franchies (ce qui est nécessaire afin de gérer les priorités), nous aurons besoin de tester la présence du jeton dans certaines places du *CPN*. Une solution aurait pu être la lecture de l'information sur les arcs de lecture. Cependant, afin de rendre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

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

Fusion des transitions

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

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



(a) SMD (b) CPN selon l'Algorithme 1

FIGURE 5.12 – Exemple avec priorités

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

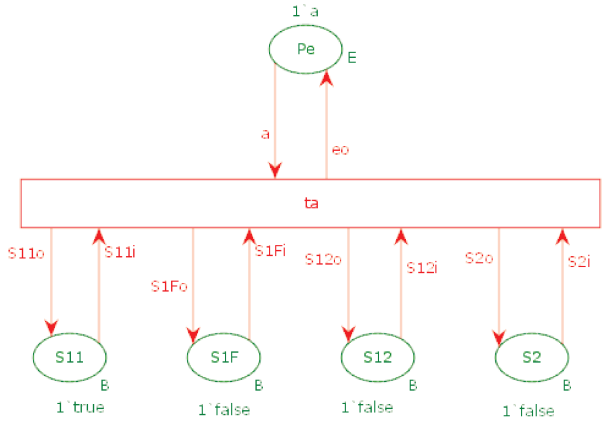


FIGURE 5.13 – Un exemple d'une transition fusionnée pour l'événement `a`

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

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

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

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

Algorithme de gestion des priorités

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

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

Algorithm 2: Gestion des priorités de franchissement

1 Add (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 (Pe)
    `// Add the transition corresponding to the fusion, and link` **Pe** `and` **t_***e*
4 Add (Pe) ⟷ *ei*/*eo* [t_e]
    `// Step 2`
5 **foreach** *transition* $\mathbf{t} = (\mathbf{S}_1, \mathrm{e}_1, g, (b, f), sLevel, \mathbf{S}_2) \in \mathcal{T}$ *such that* $\mathrm{e}_1 = \mathrm{e}$ **do**
6 Delete [t]
7 **foreach** *simple state* $\mathbf{s_1} \in \mathbf{S}_1$ **do**
8 Add (s1) ⟷ $\mathbf{s_1}i$/$\mathbf{s_1}o$ [t_e]
9 **foreach** *simple state* $\mathbf{s_2} \in \mathbf{S}_2$ **do**
10 Add (s2) ⟷ $\mathbf{s_2}i$/$\mathbf{s_2}o$ [t_e]
11 **foreach** *completion transition* $(\mathbf{S}'_1, noEvent, g', (b', f'), sLevel', \mathbf{S}'_2) \in \mathcal{T}$ **do**
12 **foreach** *simple state* $\mathbf{s} \in \mathbf{S}'_1$ **do**
13 Add (s) ⟷ *si*/*so* [t_e]
14 *AddCodeSegments*() ;

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

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

Détection et gestion des completion events

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

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

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

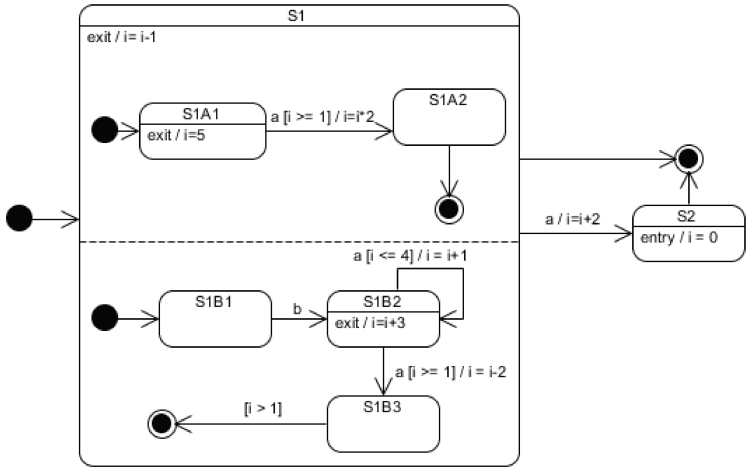


FIGURE 5.14 – Exemple avec des priorités et de la concurrence

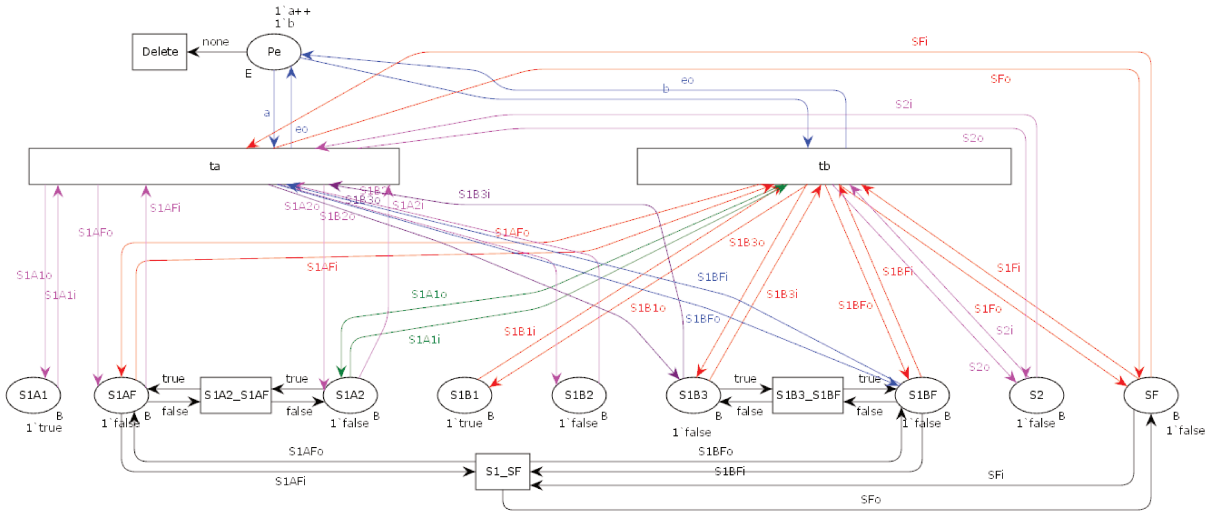


FIGURE 5.15 – Exemple avec des priorités et de la concurrence : traduction

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

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

```
function completionEventEnabled()
    S12i or S1Fi or S2i or SFi
```

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

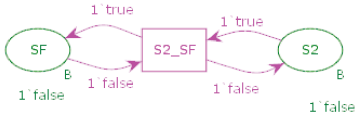


FIGURE 5.16 – Modélisation d'une transition de terminaison

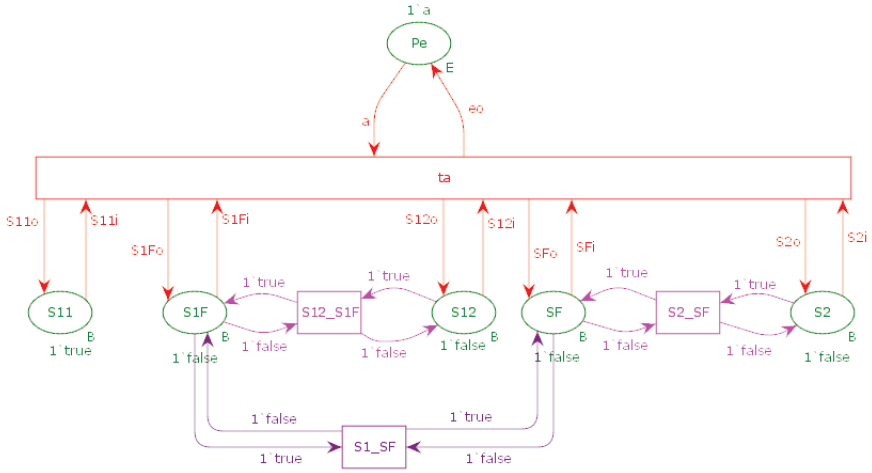


FIGURE 5.17 – Exemple avec les priorités : traduction complète sans code

```
input(S11i,S12i, S1Fi, S2i, SFi);
output(eo,S11o,S12o,S1Fo,S2o, SFo);
action
(if S12i=true  orelse S1Fi=true   orelse S2i=true orelse 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);
```

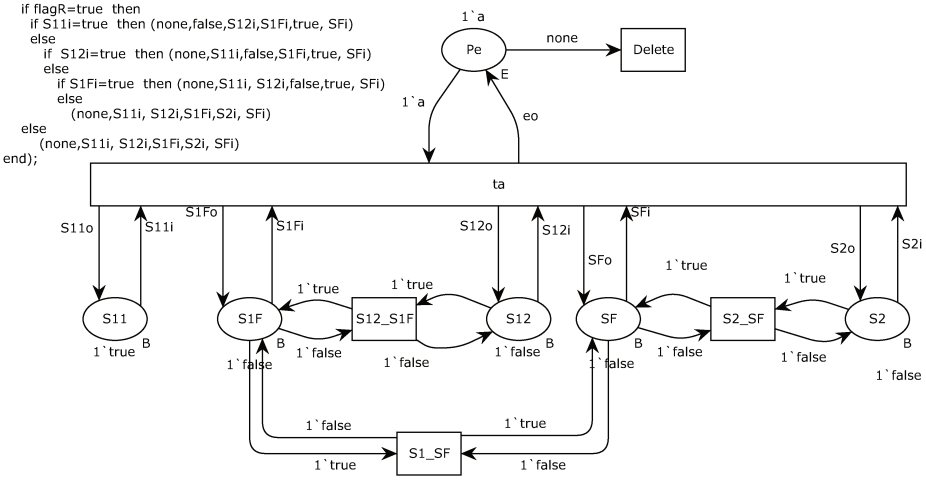


FIGURE 5.18 – Exemple avec les priorités : traduction complète avec code

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

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

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 CPN ML 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 décris 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 CPN ML décrit au-dessus).

Ensuite, il est nécessaire de déterminer quelle transition va être franchie. Nous effectuons cela en utilisant du code CPN ML 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 remit 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 contenants 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.

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

```
if completionEventEnabled() then
  (* No firing of "a" for now: return all tokens as they are, including the event
    itself *)
  S11i, S12i, S1Fi, S2i, SFi, "a"

else
  (* Prepare the resulting values, initially equal to the input values *)
  let S11o, S12o, S1Fo, S2o, SFo =
    S11i, S12i, S1Fi, S2i, SFi
  in

  (* Define flags for all regions *)
  let flagS1, flagR = true, true in

  (* Copy variables *)
  (* no variable to copy in this SMD *)

  if flagS1 then
    if S11i (* and no guard here *) then
      S11o := false (* disable input places *)
      S12o := true (* enable output places *)
      flagR := false (* block parents regions *)
      (* no behaviour to execute here *)

  if flagR then
    if S11i (* and no guard here *) then
      S11o := false (* disable input places *)
      S2o := true (* enable output places *)
      (* no parents regions to block *)
      (* no behaviour to execute here *)
    else if S12i (* and no guard here *) then
      S12o := false (* disable input places *)
      S2o := true (* enable output places *)
      (* no parents regions to block *)
      (* no behaviour to execute here *)
    else if S1Fi (* and no guard here *) then
      S1Fo := false (* disable input places *)
      S2o := true (* enable output places *)
      (* no parents regions to block *)
      (* no behaviour to execute here *)

  (* Return all tokens values *)
  S11o, S12o, S1Fo, S2o, SFo, "none"
```

La première partie du code CPN ML permet de vérifier s'il y a des transitions de terminaison active dans le diagramme états-transitions. Dans ce cas de figure tous les jetons sont remis à leurs places sans les échanger, y compris le jeton de l'événement. Sinon, nous définissons des valeurs de sortie (qui étiquettent les arcs allant de la transition fusionnée), et qui sont initialement égales aux valeurs d'entrée[2]*. Ensuite, les drapeaux sont définis pour chaque région, par exemple le drapeau de la région contenant* S1 *est* flagR*. Nous vérifions pour chaque région si une transition étiquetée avec l'événement* a *peut être franchie, et cela en vérifions successivement que la région contenant la transition est toujours active ("*if flagS1 then*"), que le jeton dans la place d'entrée est présent ("*if S11i*"), et que la garde de la transition est vérifiée (dans cet exemple, il n'y a pas de garde). Dans le cas où ces conditions sont vérifiées nous changeons la valeur des jetons des places source (la valeur devient faux) ainsi que ceux des places destination (la valeur devient true). Nous désactivons également les autres régions contenant cette transition ("*flagR := false*") et nous exécutons les comportements en utilisant la fonction AddBehaviours(). Notons que les combinaisons engendrées par la sortie des états composites (trois combinaisons en sortant de 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 CPN ML pour définir les variables.

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

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

Example 2. *Considérons dans ce qui suit un exemple (Figure 5.14) comportant des comportements, des gardes et de la concurrence. Nous donnons le code CPN ML correspondant.*

```
if completionEventEnabled() then
  (* No firing of "a" for now: return all tokens as they are, including the event
    itself *)
  S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi, "a"

else
  (* Prepare the resulting values, initially equal to the input values *)
  let S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo =
    S1A1i, S1A2i, S1AFi, S1B1i, S1B2i, S1B3i, S1BFi, S2i, SFi
  in

  (* Define flags for all regions *)
  let flagS1A, flagS1B, flagR = true, true, true in

  (* Copy variables *)
  let i' = i in

  if flagS1A then
    if S1A1i and i'>=1 then
      S1A1o := false (* input places *)
      S1A2o := true (* output places *)
      flagR := false (* block parents regions *)
      i:=5 ; i:=i*2 (* transition behaviours *)

  if flagS1B then
    if S1B2i and i'<=4 then
      S1B2o := false (* input places *)
      S1B2o := true (* output places *)
      flagR := false (* block parents regions *)
      i:=i+3 ; i:=i+1 (* transition behaviours *)
    else if S1B2i and i'>=1 then
      S1B2o := false (* input places *)
      S1B3o := true (* output places *)
      flagR := false (* block parents regions *)
      i:=i+3 ; i:=i-2 (* transition behaviours *)

  if flagR then
    if S1A1i and S1B1i (* and no guard here *) then
      S1A1o := false; S1B1o := false (* input places *)
      S2o := true (* output places *)
      (* no parents regions to block *)
      i:=5 ; i:=i-1; i:=i+2 ; i:=0 (* transition behaviours *)
    else if S1A1i and S1B2i (* and no guard here *) then
      S1A1o := false; S1B2o := false (* input places *)
      S2o := true (* output places *)
      (* no parents regions to block *)
      interleave(<i:=5>, <i:=i+3>) ; i:=i-1; i:=i+2 ; i:=0 (* transition behaviours *)
    else if S1A1i and S1B3i (* and no guard here *) then
      ...etc...

  (* Return all tokens values *)
  S1A1o, S1A2o, S1AFo, S1B1o, S1B2o, S1B3o, S1BFo, S2o, SFo, "none"
```

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

- *Dans le cas où une transition peut être franchie dans la région d'en haut de* `S1`*, alors la région d'en bas n'est pas bloquée, seulement la région racine est bloquée afin d'empêcher la transition allant de* `S1` *vers* `S2` *d'être franchie. Par conséquent, une seule transition dans la région d'en haut de* `S1` *et/ou une seule transition dans la région d'en bas de* `S1` *peuvent être franchies.*

- *Dans le cas d'une région orthogonale, nous avons besoin d'énumérer toutes les combinaisons des états sources, le code CPN ML correspondant peut être engendré automatiquement en utilisant la fonction combinations de l'Section 5.5.*

- *Dans le cas de la sortie de l'état* `S1` *en étant dans des régions orthogonales, les comportements sont entrelacés (par exemple, en étant dans l'état* `S1A1` *et l'état* `S1B2`*) en utilisant la fonction interleave définie dans la Section 5.5.*

Discussion

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

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

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

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

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

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

Conclusion

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

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

Chapitre 6

Étude de cas

Contents

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 do 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 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*.*

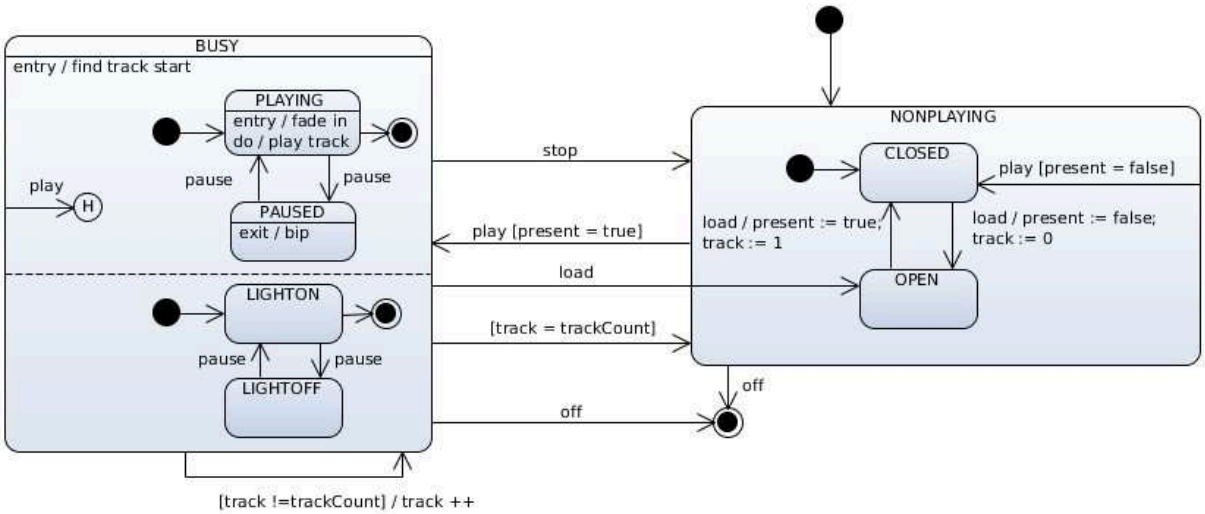


FIGURE 6.1 – Exemple du lecteur de CD

La Figure 6.1 montre le diagramme états-transitions du lecteur de CD [1].

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 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.

- `PAUSED` dénote l'état où le lecteur est en mode pause (la lecture de la piste est suspendue).
- `LIGHTON` dénote l'état où la lumière (la lompe sur le devant du lecteur) est allumée.
- `LIGHTOFF` dénote l'état où la lumière est éteinte.

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

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

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

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

Traduction du diagramme états-transitions du lecteur de CD

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

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

Limites des formalismes discrets pour des comportements continus

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

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

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

Modification du modèle

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

Le modèle modifié est présenté dans la Figure 6.2.

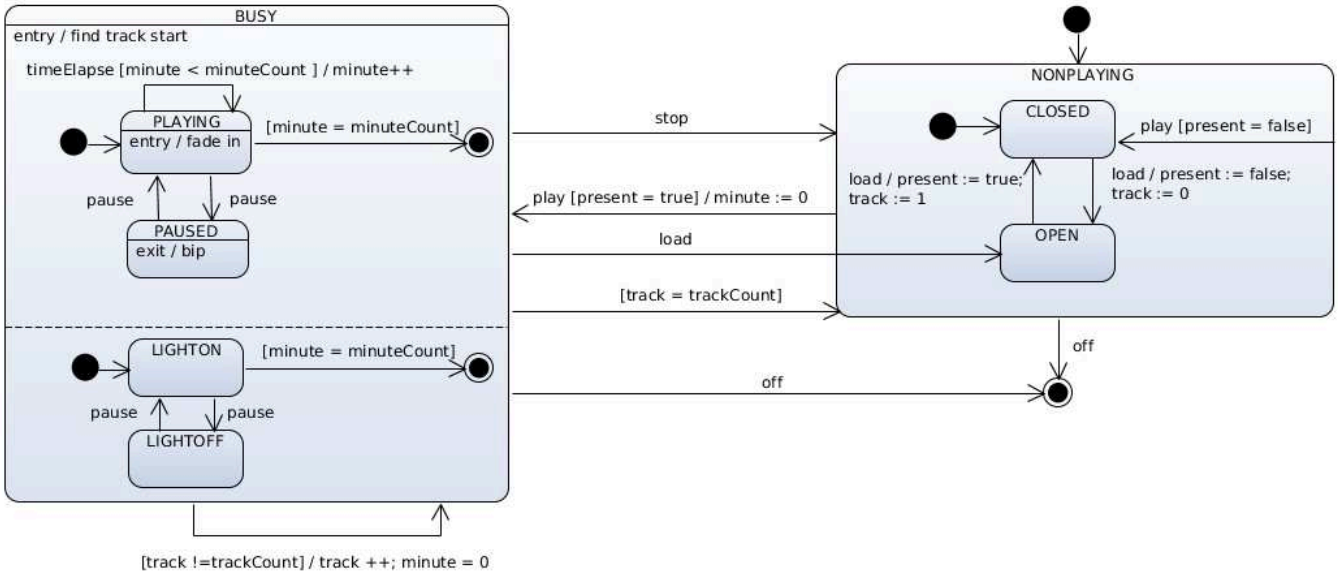


FIGURE 6.2 – Exemple du lecteur de CD (version modifiée)

Vérification

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

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

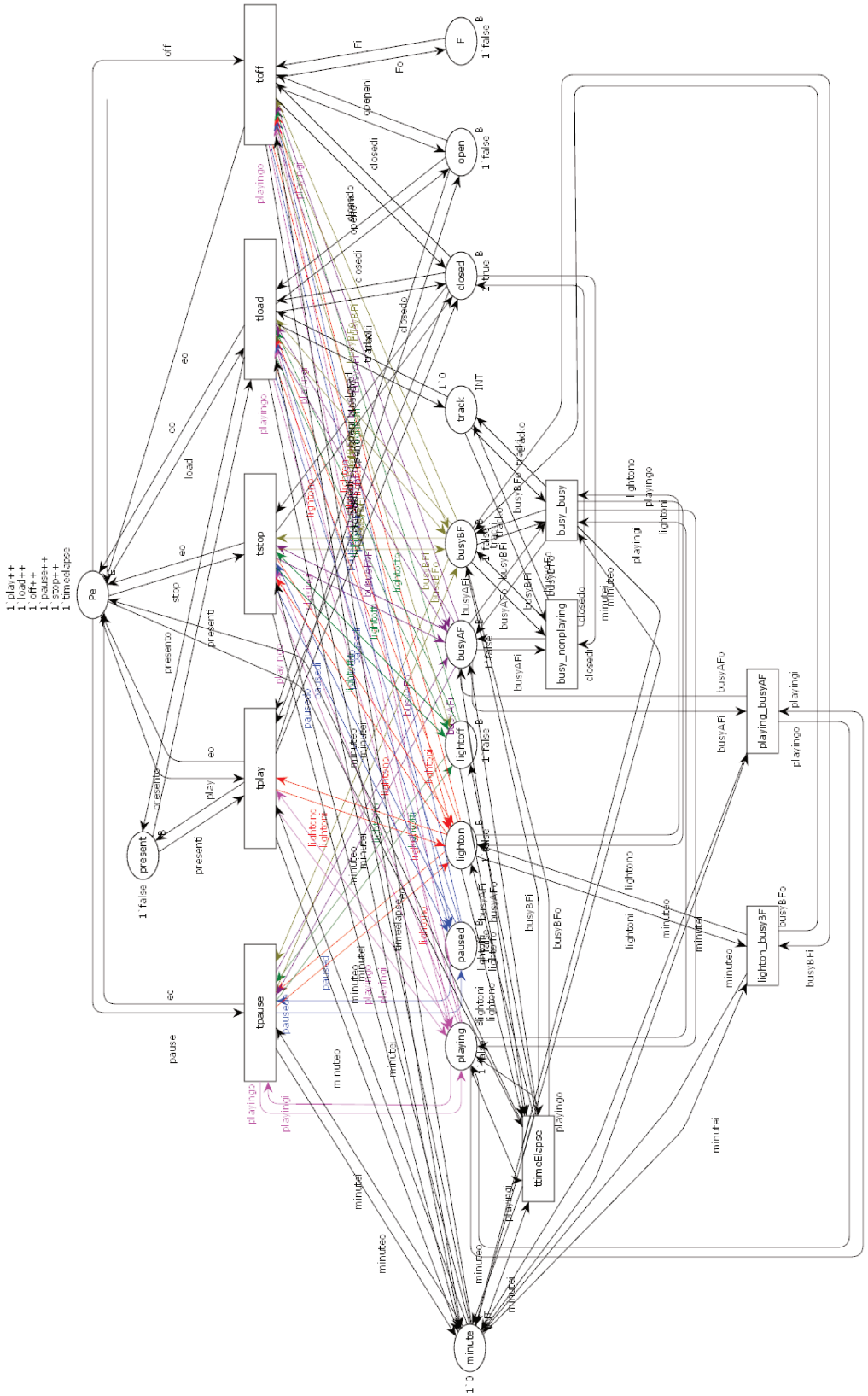


FIGURE 6.3 – Traduction en *CPN* de la version révisée de l'exemple du lecteur de CD : version sans code

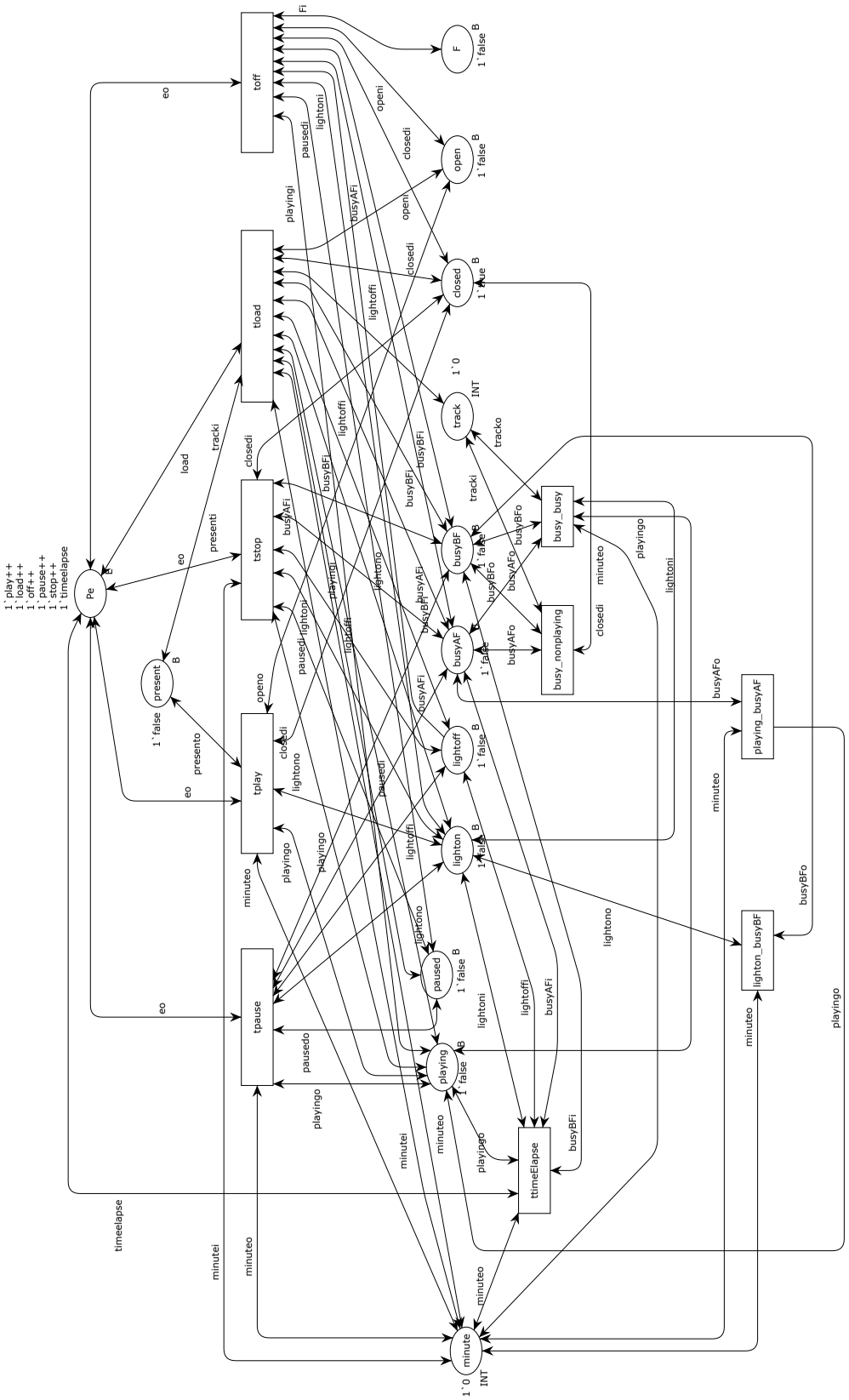


FIGURE 6.4 – Traduction en *CPN* de la version révisée de l'exemple du lecteur de CD : version simplifiée

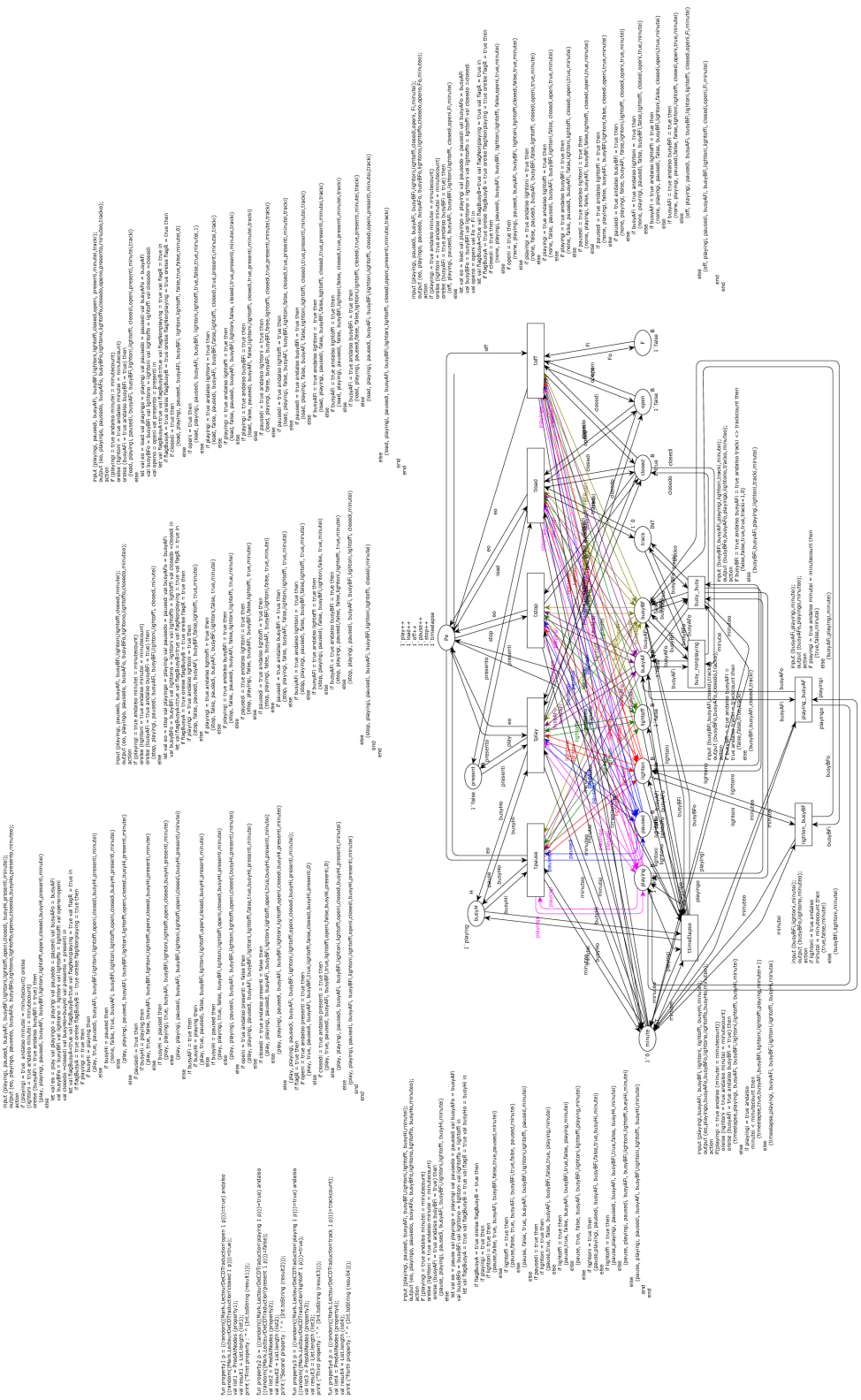


FIGURE 6.5 – Traduction en *CPN* de la version révisée de l'exemple du lecteur de CD : version complète

Nous utilisons CPN Tools 4.0 [**?**] pour vérifier les propriétés sur notre modèle modifié du 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 4.0 s 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.

```
fun property1 p = ((random((Mark.LecteurDeCDTraductionLastVersion'open 1 p)))=true)
    andalso
((random((Mark.LecteurDeCDTraductionLastVersion'closed 1 p)))=true);
val list1 = PredAllNodes (property1);
val result1 = List.length (list1);
print ("First property : " ^ (Int.toString (result1)));
```

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.

```
fun property2 p = ((random((Mark.LecteurDeCDTraductionLastVersion'playing 1 p)))=true)
    andalso
((random((Mark.LecteurDeCDTraductionLastVersion'present 1 p)))=false);
val list2 = PredAllNodes (property2);
val result2 = List.length (list2);
print ("Second property : " ^ (Int.toString (result2)));
```

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 une 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.

```
fun property3 p = ((random((Mark.LecteurDeCDTraductionLastVersion'playing 1 p)))=true)
    andalso
((random((Mark.LecteurDeCDTraductionLastVersion'lightoff 1 p)))=true);
val list3 = PredAllNodes (property3);
val result3 = List.length (list3);
print ("Third property : " ^ (Int.toString (result3)));
```

Propriété 4 (“la valeur de `track` ne dépasse jamais `trackCount`”) Afin de valider cette propriété il suffit de vérifier si dans n’importe quel état de l’espace d’états que `track` ≤ `trackCount`. La propriété est vérifiée avec CPN Tools et son code CPN ML est le suivant.

```
fun property4 p = ((random((Mark.LecteurDeCDTraductionLastVersion'track 1 p)))>trackcount
    );
val list4 = PredAllNodes (property4);
val result4 = List.length (list4);
print ("Forth property : " ^ (Int.toString (result4)));
```

La Figure 6.6 montre l’évaluation des quatre propriétés présentée en haut après génération 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.

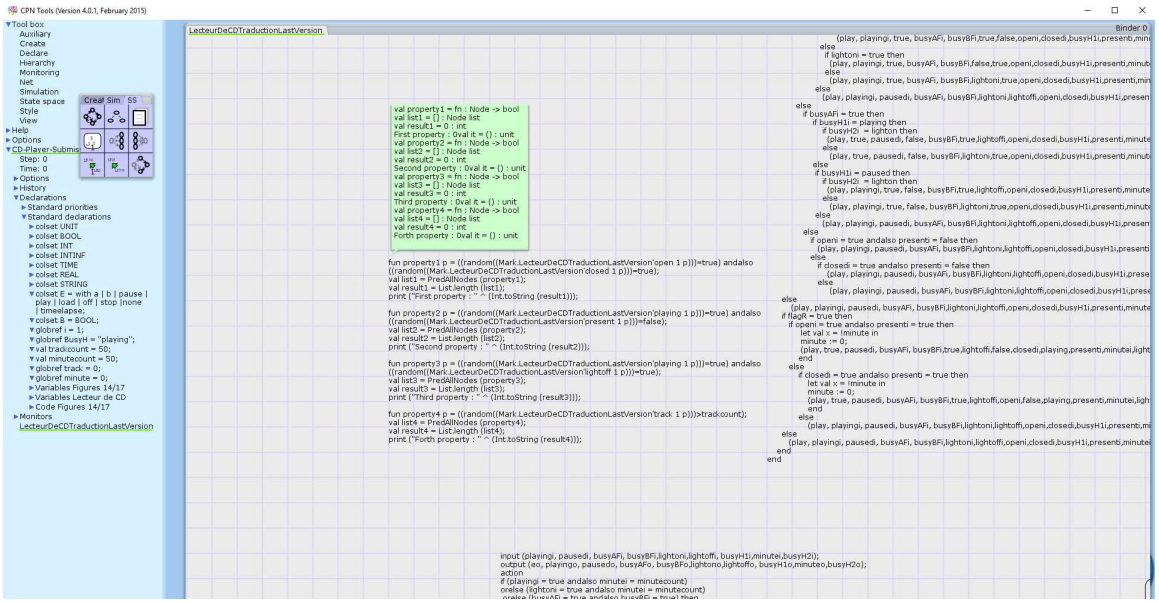


FIGURE 6.6 – Évaluation des propriétés dans CPN Tools

Remarque Notre vérification a été effectuée en utilisant des valeurs fixes pour les variables `trackCount` et `minuteCount`. Effectuer la vérification en utilisant *n’importe* quelle valeur pour `trackCount` et `minuteCount` nécessite des techniques de vérification paramétrées (ce qui sort du sujet de ce travail et de cette thèse).

Étude du passage à l’échelle

Nous présentons dans ce qui suit le tableau récapitulatif des différents résultats de la génération de l’espace d’états en fonction des valeurs des variables trackcount et minutecount. Nous présentons comme éléments le temps de génération de l’espace d’états, la taille de l’espace d’états (nombre des nœuds et des transitions), et le temps de génération du rapport comportant les différentes informations sur l’espace d’états. Nous utilisons l’outil de génération d’espace d’états de CPN Tools pour faire nos calculs. CPN Tools génère l’espace d’état (le temps de génération est indiqué dans le tableau par la colonne *TempsGénération*) et ensuite il vérifie les priorités (où le temps de vérification est instantané).

traccount/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

Conclusion

Afin d'appliquer nos algorithmes et faire de la vérification nous avons présenté dans ce chapitre la traduction de l'exemple du lecteur de CD. En partant du diagrammes états-transitions du lecteur de CD nous avons engendré le réseaux de Petri colorés correspondant. Nous avons montré une version sans la présence des segments de codes dans les transitions et une version complète avec du code. Enfin, nous avons vérifié quelques propriétés en utilisant l'outil CPN Tools.

Chapitre 7

Implémentation de la traduction

Contents

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

nombre d'inconvénients, par exemple l'absence de variables ou de structures de données (deux notions nécessaires pour le fonctionnement de nos algorithmes).

En se basant sur notre expérience de l'utilisation des techniques de l'ingénierie dirigée par les modèles, nous proposons une implémentation (présentée en Section 7.3) en développant notre propre outil. Cet outil (UML2CPN) est développé en utilisant java et des analyseurs syntaxiques.

Implémentation avec Acceleo

Nous présentons dans cette section [?] un travail d'exploration afin d'implémenter une première version de la traduction des diagrammes états-transitions vers les réseaux de Petri colorés [?] en utilisant un outil de transformation de texte vers du modèle (Acceleo).

Acceleo est un outil permettant d'implémenter des générateurs de code. Il est basé sur les techniques modèle vers texte et surtout ne nécessite pas d'avoir le méta-modèle du formalisme de sortie. L'autre avantage d'Acceleo (et un avantage des model2text en général) est que nous pouvons définir le format du code engendré pour qu'il soit conforme à l'outil qui l'utilise. Nous utilisons CPN tools [?] pour manipuler les réseaux de Petri colorés et le fichier que CPN Tools accepte est sous format XMI. Avec Acceleo nous engendrons donc un fichier XMI ayant le même format qu'un fichier de CPN Tools et contenant les informations dont nous avons besoin. Cette implémentation a été présentée dans le travail de [?]. La Figure 7.1 montre un exemple de code Acceleo. La partie gauche de la figure représente un mélange entre du code Acceleo et du texte alors que la partie droite représente l'implémentation d'une fonction avec du code Acceleo : les lignes de code 1, 2 et 3 de la figure à gauche représentent le code Acceleo utilisé dans l'outil (le code sert pour l'implémentation de l'algorithme) et que les lignes 4-8 de la figure à gauche représentent du texte au format des fichiers utilisés dans CPN Tools.

```
1  [template public SupEn1(s : State,
   pere : State, as : StateMachine)]
2    [if (s.Entry().contains('true'))]
3      [if (pere.Entry().contains('true')
   )]
4        <arc id="ArcNRootSENS[s.Name/]
5                [s.Id/]" orientation="
   PtoT" order="1">
6        <posattr x="0.000000"
7                 y="0.000000"/>
8      ...
9    [/template]
10
```

(a) Mélange du code Acceleo avec la syntaxe de CPN Tools

```
1  [template public substates(s : State,
   as : StateMachine)]
2    [if (s.isSimple = true)]
3      [s.Name/]
4    [else]
5      [for (x : State | as.State)]
6        [if (x.Parent = s.Name)]
7          [substates(x, as)/]
8        [/if]
9      [/for]
10   [/if]
11 [/template]
12
```

(b) Mélange du code Acceleo avec la syntaxe de CPN Tools

FIGURE 7.1 – Exemple du code Acceleo

Cependant il y a plusieurs inconvénients à cette approche. D'une part, Acceleo ne permet pas de manipuler les variables, les fonctions et les structures de données. D'autre part, la génération du code dans Acceleo est basée sur la syntaxe de l'outil du formalisme de sortie ce qui implique qu'il n'est pas possible d'adapter l'implémentation dans le cas du changement d'outil du formalisme de sortie (c'est un inconvénients des outils model2text en général).

Au vu des inconvénients de l'utilisation d'Acceleo mais aussi de l'impossibilité d'utiliser un outil de transformation de modèles, nous travaillons sur un outil dont nous implémentons les différentes parties (telles que l'analyseur syntaxique, la fonction de transformation, etc). Cela

nous permettra non seulement de l'adapter en cas de changement d'outils (pour le formalisme de sortie ou d'entrée) mais aussi en cas de modification des algorithmes.

Implémentation d'un nouvel outil : UML2CPN

En nous basant sur notre expérience de l'utilisation des techniques modèle vers le texte, nous avons décidé d'implémenter notre traduction ([**?**]) en définissant notre propre outil UML2CPN. Nous présentons dans cette section les différentes parties de cet outil.

Notons qu'UML2CPN est en cours de développement et qu'uniquement une partie de l'Algorithme 1 (présenté en Section 5.5) est considérée. La partie implémentée est l'étape 1 (en ligne 1) et une partie de l'étape 2 (en ligne 5) de l'Algorithme 1. L'étape 1 permet la génération des places de tous les états ainsi que les transitions de leurs comportements Do. L'étape 2 permet la génération de toutes les transitions dans le diagramme états-transitions ceux avec ou sans événements. La partie que nous implémentons de cette étape (c'est-à-dire l'étape 2) est celle des transitions sans événements. Le développement final de notre outil implémentera l'intégralité des Algorithmes 1 et 2.

Nous utilisons pour le développement d'UML2CPN le langage de programmation *Java*, l'environnement de développement *Netbeans* ainsi que l'analyseur syntaxique *JDOM 2.0.5*[2]. Il est possible d'utiliser d'autres analyseurs syntaxiques pour compléter le développement de ce dernier. UML2CPN prend en entrée un fichier *XMI* qui représente le diagramme états-transitions. Ce fichier est édité en utilisant le plugin *Papyrus*[3] dans *Eclipse*[4]. Une version de notre outil est en cours de développement afin de prendre en compte les fichiers engendrés par *Visual paradigm*[5].

Afin d'expliquer au mieux UML2CPN, nous présentons dans ce qui suit les différents éléments qui le composent :

- Bibliothèques
- Packages et classes
- Interfaces

Bibliothèques

Comme indiqué ci-dessus, nous utilisons *JDOM* comme analyseur syntaxique et cela pour parcourir, manipuler et gérer le fichier *XMI* que l'outil UML2CPN prend en entrée. Il est possible d'utiliser *SAX*[6] comme analyseur syntaxique, cependant *JDOM* est plus simple et plus facile à utiliser. Par contre, il est nécessaire de noter que lors de la manipulation du fichier *XMI* nous utilisons *SAXBuilder* qui est inclus dans les fichiers importés de *JDOM*. Afin de pouvoir accéder à toutes les fonctionnalités de ce plugin il est nécessaire d'une part d'importer les fichiers dans le code Java (les lignes de code 24-25 encadrées avec du rouge dans la partie droite de la Figure 7.2) mais également d'ajouter à la bibliothèque du plugin *JDOM* dans le projet (les lignes encadrées avec du rouge dans la partie gauche de la Figure 7.2).

Packages et classes

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

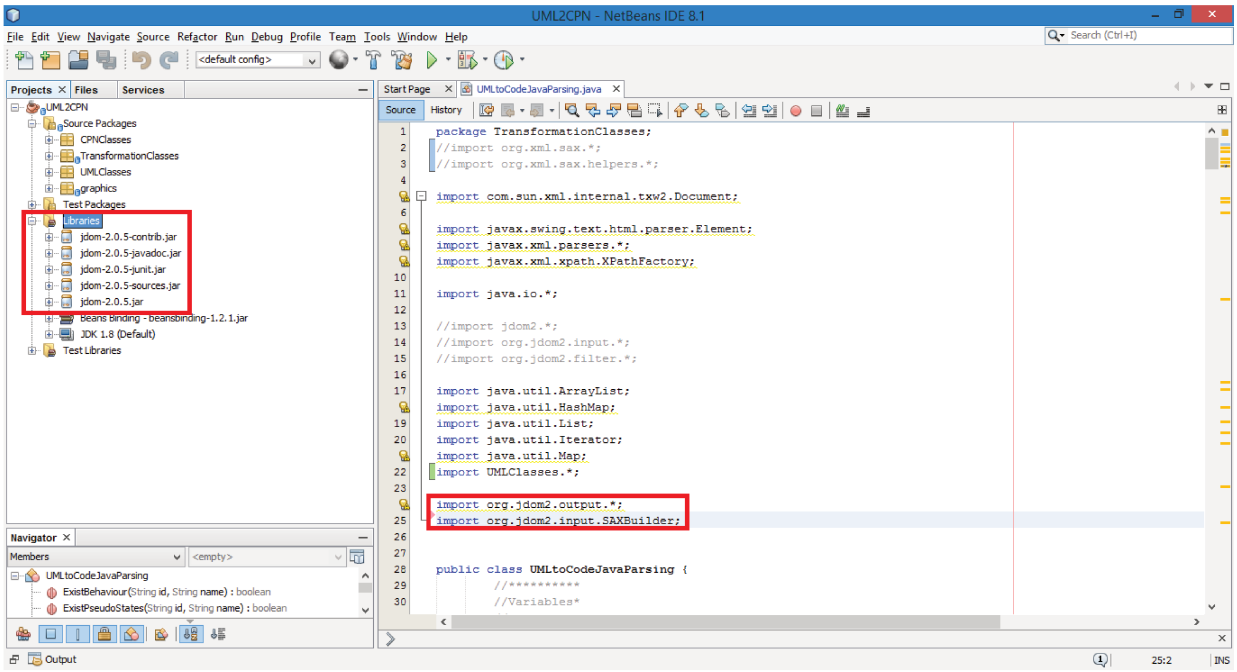


FIGURE 7.2 – Fichiers *JDOM*

- Le package *CPNCLasses* qui comporte les classes *cpnArc*, *cpnPlace* et *cpnTransition*. Ces classes permettent de stocker les données des places, transitions et arcs résultats de l'application des algorithmes de traductions.

- Le package *TransformationClasses* qui comporte les classes *Function*, *UMLToCPNTranslation* et *UMLtoCodeJavaParsing*. La première classe comporte l'ensemble des fonctions utilisées dans l'outil. La seconde classe comporte le code qui implémente nos algorithmes. Enfin, la troisième classe comporte le code qui permet le passage d'un fichier *XMI* vers du code Java.

- Le package *UMLClasses* qui comporte les classes *Behaviour*, *ForkJoin*, *History*, *PseudoState*, *Region*, *SpecialTransition*, *State* et *Transition*. Ces classes contiennent les informations sur tous les éléments syntaxiques du modèle récupéré en entrée dans notre outil. Par exemple, la classe *Behaviour* contient toutes les informations concernant les comportements d'entrée, de sortie et do pour chaque état dans le diagramme états-transitions.

- Le package *graphics* qui comporte les classes *ChooserFile*, *Interface* et *UML2CPNInterface*. Les deux premières classes sont utilisées uniquement pour des tests, par contre la troisième classe permet la définition de l'interface de l'outil.

Interface et utilisation de l'outil UML2CPN

Nous présentons dans cette section l'interface de notre outil ainsi que comment l'utiliser.

La Figure 7.4 montre l'interface de notre outil UML2CPN. Cette interface comporte des affichages et un ensemble de boutons :

- Le bouton *Choose the model* qui permet de choisir le fichier *XMI* à utiliser (ce fichier contient le diagramme états-transitions à traduire).

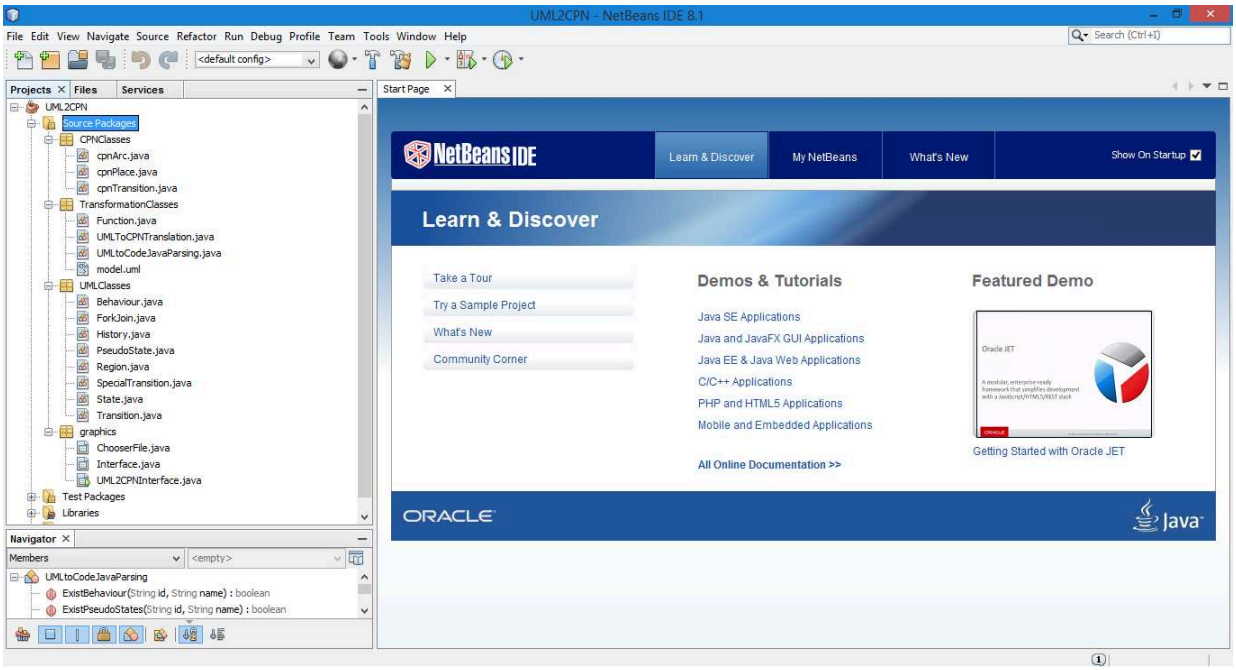


FIGURE 7.3 – Packages et classes utilisés dans UML2CPN

- Le bouton *Parsing the SMD* qui permet de faire le passage de la représentation graphique (en utilisant le fichier *XMI*) des diagrammes états-transitions vers la représentation correspondante en Java (c'est-à-dire les données que le diagramme représente seront stockées dans des structures de données).
- Le bouton *Translation to CPN* qui permet de traduire le diagramme états-transitions vers le réseau de Petri coloré correspondant en utilisant nos algorithmes.
- Le bouton *Generation of CPNTools file* qui permet d'engendrer le fichier contenant le résultat de la traduction et qui est adapté à ce que l'outil CPN Tools prend en compte comme fichier.
- L'affichage *SMD states* contient tous les états du diagramme états-transitions récupérés à partir du fichier *XMI* en entrée.
- L'affichage *SMD transitions* contient toutes les transitions du diagramme états-transitions source.
- L'affichage *Fork and Join Transition* contient toutes les transitions Fork et Join du diagramme états-transitions source.
- L'affichage *CPN places* contient toutes les places du réseau de Petri coloré après la traduction.
- L'affichage *CPN transitions* contient toutes les transitions du réseau de Petri coloré résultat.
- L'affichage *CPN arcs* contient tous les arcs du réseau de Petri coloré résultat.

Nous avons défini une interface simple pour notre outil afin de faciliter l'utilisation de ce dernier par les utilisateurs. L'utilisateur commence par récupérer son fichier *XMI* en cliquant

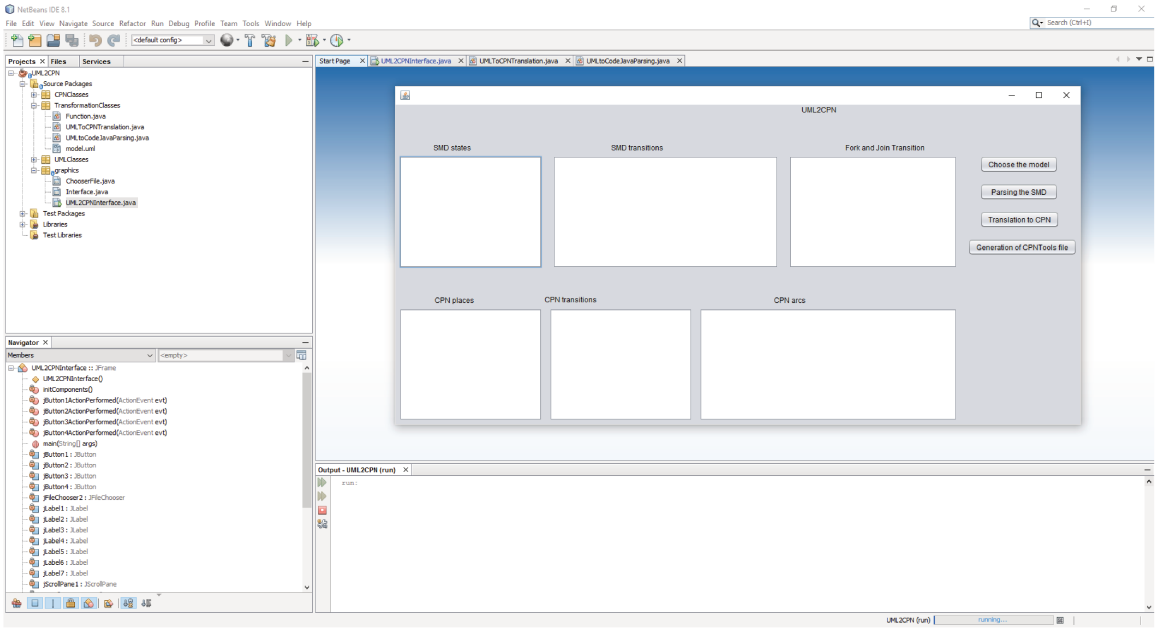


FIGURE 7.4 – L'interface de l'outil UML2CPN

sur le bouton *Choose the model*. Dans les fichiers engendrés par *Papyrus* il y a le fichier *.di* qu'il faut charger comme montré dans la Figure 7.5 (le fichier est model.di dans la figure).

Une fois que l'utilisateur choisit le fichier *XMI*, il peut récupérer les différentes données de son diagramme (tels que les noms des états, les comportements, les transitions etc) afin des les stocker et les utiliser après en cliquant sur le bouton *Parsing of the SMD*. Une fois le code engendré, l'outil affichera tous les états, les transitions et les transitions Fork et Join que le diagramme états-transitions (récupéré en entrée) comporte. La Figure 7.6 montre l'état de l'outil après l'utilisation du bouton *Parsing of the SMD*.

Dans l'étape suivante, l'utilisateur peut appliquer nos algorithmes afin d'engendrer le réseau de Petri coloré (correspondant au diagramme états-transitions source). Afin de faire la traduction, l'utilisateur doit cliquer sur le bouton *Translation to CPN* qui permet d'afficher les places, transitions et arcs du réseau de Petri coloré résultat (voir les zones d'affichages *CPN places*, *CPN transitions* et *CPN arcs* dans l'outil). Rappelons que notre outil est en cours de développement et actuellement nous n'avons implémenté qu'une partie de nos algorithmes. La Figure 7.7 montre l'application de la traduction sur le diagramme états-transitions source et le résultat (les réseaux de Petri colorés ne sont pas visibles graphiquement car c'est un module que nous n'avons pas encore implémenté) .

Enfin, l'utilisateur peut engendrer (en cliquant sur le bouton *Generation of CPNTools file* voir la Figure 7.8 pour le bouton) le fichier compatible avec CPN Tools et qui contiendra le réseau de Petri coloré (résultat de la traduction). Ce fichier sera directement utilisable avec l'outil CPN Tools. Rappelons que cette fonctionnalité n'est pas encore implémentée dans notre outil et par conséquent le réseau de Petri coloré ne sera pas affiché.

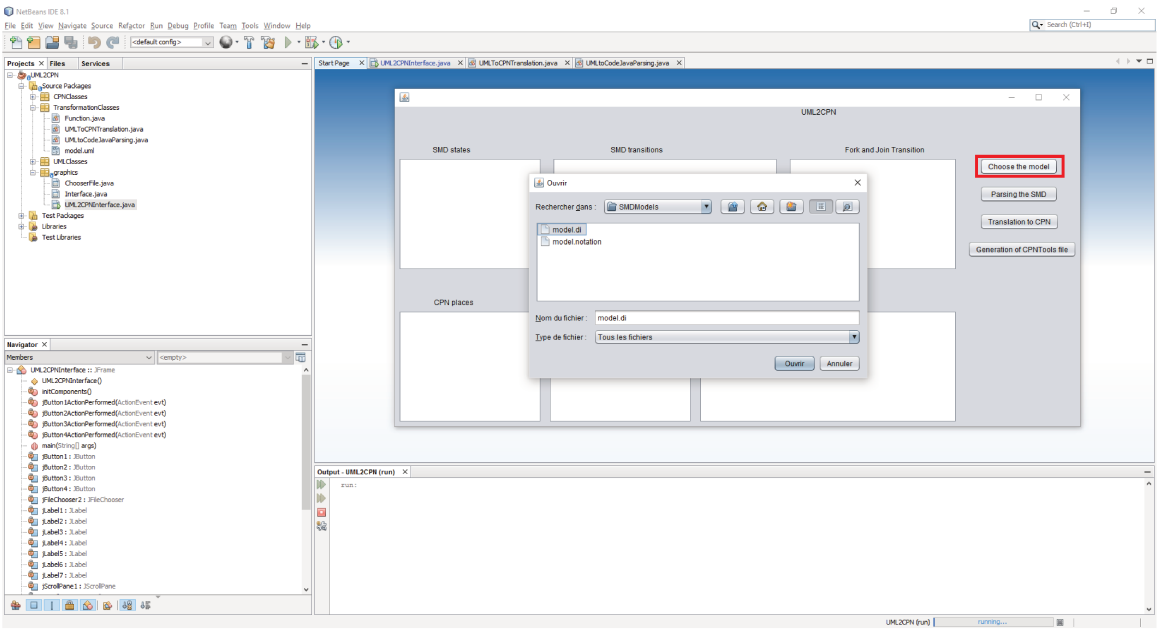


FIGURE 7.5 – La récupération du fichier *XMI* dans UML2CPN

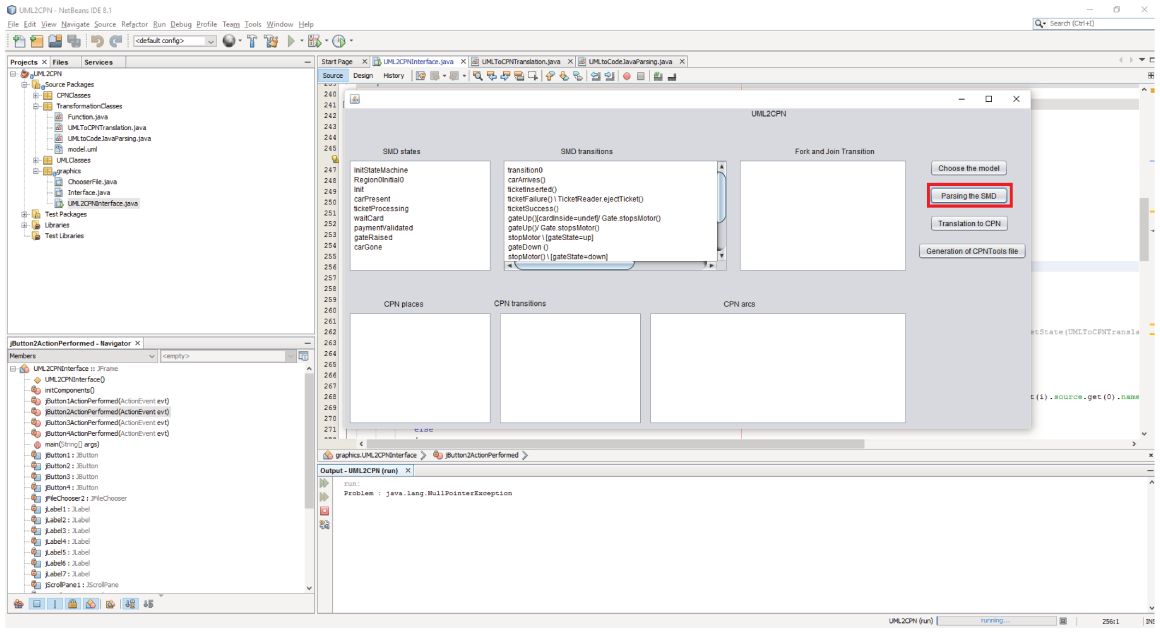


FIGURE 7.6 – La récupération des données du SMD à partir du fichier *XMI* dans UML2CPN

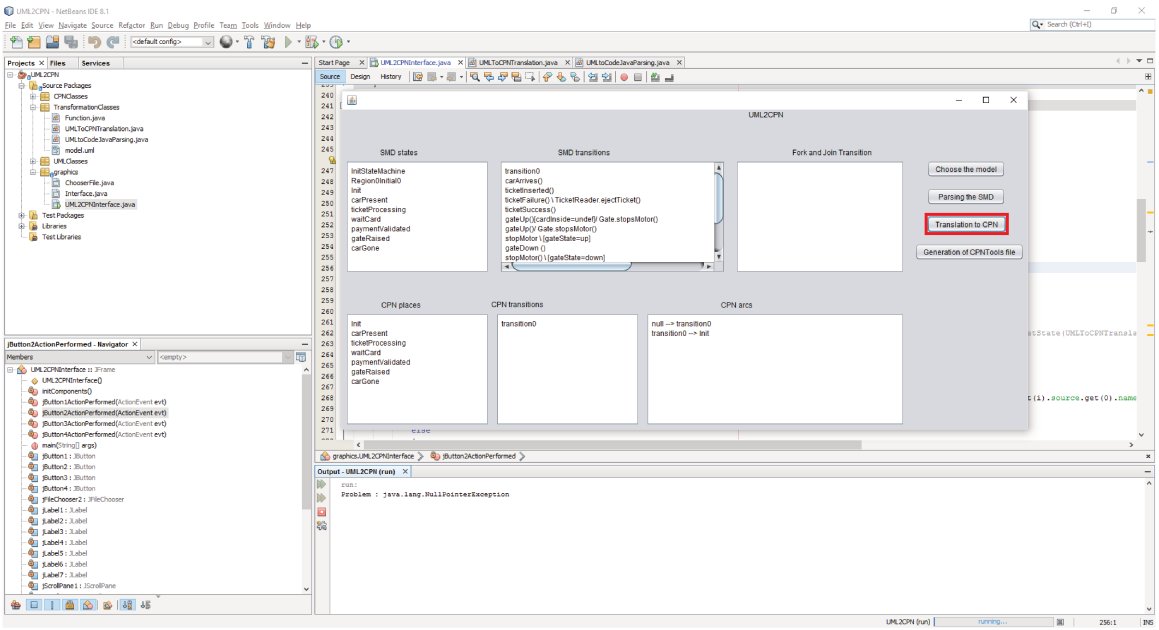


FIGURE 7.7 – La traduction du diagramme états-transitions vers le réseaux de Petri coloré correspondant dans UML2CPN (version non graphiques)

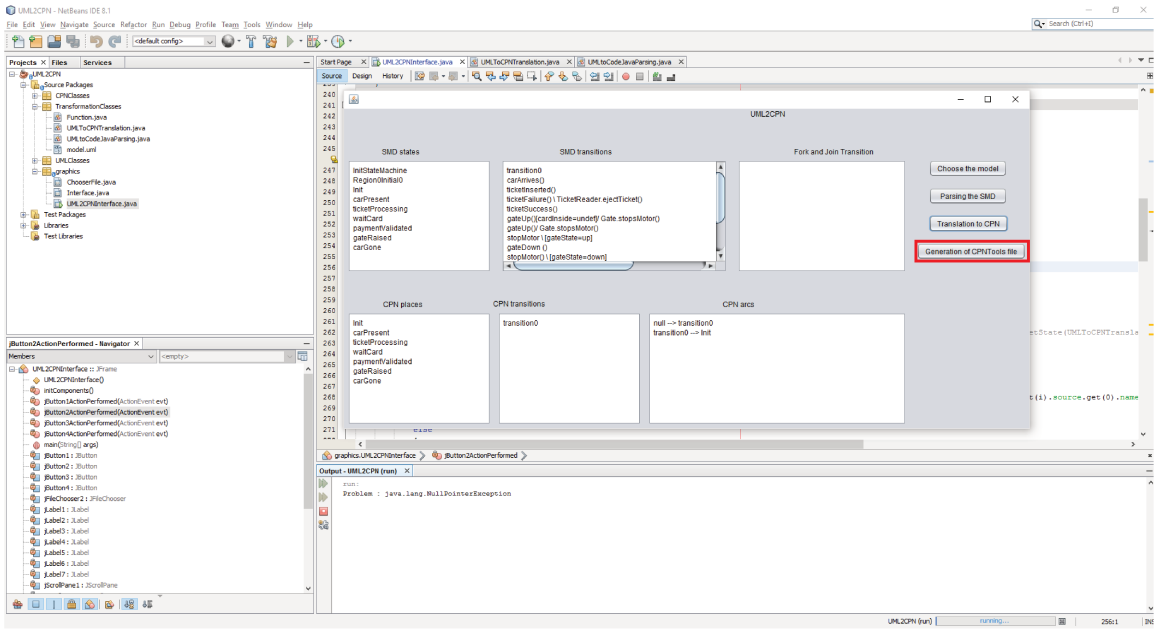


FIGURE 7.8 – La génération du fichier CPN Tools dans UML2CPN (module en cours de développement)

Conclusion

Notre outil UML2CPN est en cours de développement et nous souhaitons le finaliser afin d'avoir une traduction automatique des diagrammes états-transitions vers les réseaux de Petri colorés. La version actuelle d'UML2CPN implémente une partie de l'Algorithme 1 de notre traduction. Afin de compléter l'outil et donc implémenter entièrement notre traduction, nous souhaitons terminer l'implémentation du premier algorithme et écrire l'implémentation du second algorithme. Enfin, nous avons l'intention de créer un outil qui combine à la fois l'outil *EasySM* (outil d'aide à la modélisation avec les diagrammes états-transitions présenté en Section 4.9) ainsi que l'outil UML2CPN (l'outil de traduction des diagrammes états-transitions vers les réseaux de Petri colorés). Cela nous permettra d'avoir un outil complet que l'utilisateur peut utiliser pour faire de la modélisation et de la vérification.
Notons que notre outil UML2CPN sera également mis à jour une fois que nous aurons intégré les expressions temporelles dans notre traduction.

Chapitre 8

Expression temporelle

Contents

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 nous sommes inspiré des travaux présentés dans la Section 3.3.4.

La Figure 8.1 montre le diagramme de classes de la classification des contraintes temporelles avec les différents besoins temporels qui sont considérés dans cette thèse.

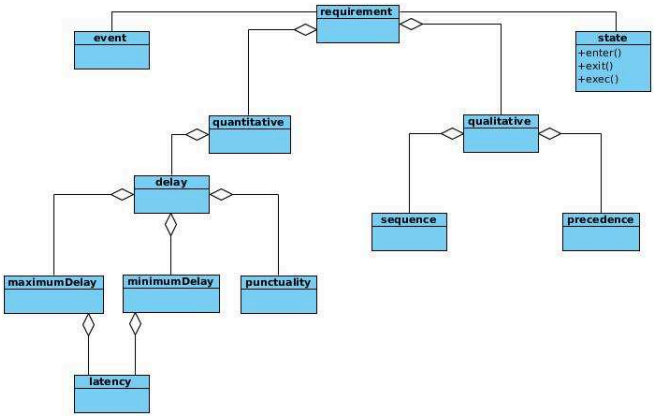


FIGURE 8.1 – Nouvelle classification des besoins temporels

Comme la Figure 8.1 le montre nous considérons les contraintes temporelles vis-à-vis des événements mais également dans les états. La classification (présente dans la Figure 8.1) comporte deux types de contraintes : contraintes quantitatives (contraintes temporelles) et contraintes qualitatives (contraintes logiques). La première partie permet de représenter le délai (minimum et maximum), la ponctualité ainsi que la latence. La seconde partie permet de représenter la séquence et la précédence entre événements. La représentation des contraintes temporelles vis-à-vis les événements est faite en utilisant les deux parties. En revanche, la représentation des contraintes temporelles dans les états est faite en utilisant uniquement la première partie (c'est-à-dire les contraintes quantitatives).

Nous présentons dans ce qui suit la syntaxe et la sémantique basée sur cette classification. Nous présentons également la combinaison des contraintes qualitatives et des contraintes quantitatives.

Syntaxe et sémantique des annotations temporelles

Nous présentons dans cette sous-section la syntaxe/sémantique correspondantes à la classification que nous proposons dans le cadre de cette thèse. Nous illustrons cette syntaxe par une grammaire (voir la Figure 8.4) avec sa description. Nous accompagnons cette grammaire par le Tableau 8.1 qui comporte les différents terminaux utilisés.

Nous avons besoins de définir certains concepts que nous utiliserons pour la description de notre syntaxe comme suit. Soit `S1` et `S2` deux états et soit `Transit` une transition entre `S1` et `S2`. L'état `S1` comporte un comportement do noté `doexp`. E est l'événement qui étiquette la transition `Transit`. Ces éléments seront utilisés pour l'explication des différentes annotations du Tableau 8.1. Deux points à noter :

- Les temps utilisés dans notre syntaxe (par exemple, E `at` T) sont des temps relatifs et les intervalles pour l'évaluation des contraintes sont fermés.
- L'événement dans une transition est enrichi par une contrainte temporelle, donc c'est toujours l'événement tout à gauche de la contrainte qui est celui de la transition.

Notre choix de la classification proposée dans la Figure 8.1 est basé sur la gestion des contraintes temporelles portant sur les événements (mais également sur les états voir Section 8.3.3).

Chaque événement peut apparaître au plus T unités de temps (`at_most`), au minimum T d'unités de temps (`at_least`), exactement T unités de temps (`at`), ou bien en séquence/précédence avec d'autres événements (`after`, `before`). Il faut noter que la contrainte temporelle est vérifiée pour un événement donné dans le cas où l'état source de la transition étiquetée par l'événement (sous cette contrainte) est actif (c'est-à-dire l'état dans lequel se trouve le système). Donc le temps est mesuré à partir du moment où l'état source de la transition est actif. Dans le cas où la transition a comme source un pseudo-état join alors il faut que tous les états sources des transitions allant vers le pseudo-état join soient actifs afin que la contrainte de la transition puisse être évaluée. 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

Afin d'exprimer la sémantique informelle de chaque annotation syntaxique présentée dans le Tableau 8.1, nous présentons dans ce qui suit la description annotation :

- Dans le cas de `at` : E `at` T, c'est-à-dire l'événement E apparaît exactement T unités de temps après l'activation de l'état source de cette transition étiquetée par l'événement E. Dans le cas où l'événement E n'apparaît pas exactement T unités de temps après l'activation de l'état `S1`, la contrainte ne sera pas satisfaite. Nous avons également *doexp* `at` T, c'est-à-dire l'exécution du comportement do doit se terminer exactement T unité de temps après l'activation de l'état concerné par le comportement do.

- Dans le cas de `at_most` : E `at_most` T_{max}, c'est-à-dire l'événement E apparaît au plus T_{max} unités de temps après l'activation de l'état source de cette transition. Dans le cas du comportement *do*, nous avons *doexp* `at_most` T_{max}, c'est-à-dire l'exécution du comportement *do* doit se terminée au plus T_{max} unités de temps après l'activation de l'état concerné par le comportement do.

- Dans le cas de `at_least` : E `at_least` T_{min}, c'est-à-dire l'événement E apparaît au minimum T_{min} unités de temps après l'activation de l'état source de cette transition. Nous avons également *doexp* `at_most` T_{min}, c'est-à-dire l'exécution du comportement do doit se terminer au minimum T_{min} unités de temps après l'activation de l'état concerné par le comportement do.

- Dans le cas de `between` : E `between` $[T_{min}, T_{max}]$, c'est-à-dire l'événement E apparaît au minimum T_{min} unités de temps et au plus T_{max} unités de temps après l'activation de l'état source de cette transition. Dans le cas du comportement do, nous avons *doexp* `between` $[T_{min}, T_{max}]$, c'est-à-dire l'exécution du comportement do doit se terminer au minimum T_{min} unités de temps et au plus T_{max} unités de temps après l'activation de l'état concerné par le comportement do.

- Dans le cas de `after` : E_2 `after` E_1 (où E_2 est l'événement de la transition enrichi par la contrainte), c'est-à-dire l'événement E_2 apparaît après l'événement E_1. La sémantique que

nous considérons par rapport au `after` est que pour chaque occurrence de l'événement E_1 on peut avoir au plus une occurrence de l'événement E_2.

Afin d'expliquer la sémantique que nous considérons pour l'élément syntaxique `after` (le même principe sera pour l'élément syntaxique `before`), nous présentons l'exemple de la Figure 8.2. Dans cet exemple, l'occurrence de l'événement e_1 est conditionnée par l'occurrence de l'événement e_2. Nous présentons les séquences de franchissement de transitions possibles et impossibles que nous autorisons ou interdisons.

- La séquence d'occurrence d'événements e_1 e_2 e_1 e_2 b est autorisée car pour chaque occurrence de l'événement e_1 il est possible de franchir une transition ayant la contrainte e_2 $after$ e_1 (autrement dit avoir une occurrence de l'événement e_2). La séquence d'occurrence d'événements suivante e_1 e_2 e_1 e_1 e_1 e_2 b est également autorisée car on considère la dernière occurrence de l'événement e_1.
- La séquence d'occurrence d'événements e_1 e_2 e_2 b n'est pas autorisée car la première occurrence e_1 permet l'occurrence e_2, par contre la seconde occurrence e_2 n'est pas autorisée car il n'y a eu aucune occurrence de l'événement e_1.

Un cas particulier que nous avons remarqué dans cet exemple est la présence d'une situation d'inter-blocage dans le cas où l'événement b n'apparaît pas, que l'événement $e1$ n'apparaît qu'une fois. Cette situation implique que la transition entre l'état `S12` et l'état `S13` peut être franchie par contre le reste des transitions non.

- Dans le cas de `before` : E_1 `before` E_2 (où E_1 est l'événement de la transition enrichi par la contrainte), c'est-à-dire si l'événement E_2 apparaît, alors il y a l'occurrence de l'événement E_1 avant au moins une fois. Nous illustrons la sémantique du `before` avec un exemple et nous présentons les séquences de transitions autorisées. Dans la Figure 8.3 un exemple diagramme états-transitions utilisant l'élément syntaxique `before`. Dans ce qui suit les séquences de transitions autorisées et non autorisées avec un cas d'inter-blocage.
  - La séquence d'occurrence d'événements e_2, e_1, e_2, e_1, b est autorisée car pour chaque occurrence de l'événement e_1 il y a eu une occurrence de l'événement e_2.
  - La séquence d'occurrence d'événements e_2, e_1, e_1, e_1, b n'est pas autorisée car afin que la deuxième occurrence de e_1 soit possible il est nécessaire que e_2 soit apparu avant au moins une fois.
  - Dans le cas où l'événement b n'apparaît pas et dans le cas où on a la séquence de transitions e_2, e_2, e_1, alors on ne peut plus franchir de transitions car l'événement e_1 ne peut apparaître que si l'événement e_2 apparaît juste avant.

  Il est nécessaire que cette contrainte soit évaluée au moment où l'état `S1` est actif.

Remarques Une remarque à prendre en compte est la non présence des contraintes temporelles exprimées avec les éléments syntaxiques `after` et `before` dans les états. Nous considérons que l'exécution du comportement do ne dépend pas de l'occurrence des événements mais dépend de son temps d'exécution.

Nous présentons dans ce qui suit la grammaire représentant l'expressivité de la syntaxe que nous proposons au-dessus (voir Figure 8.4).

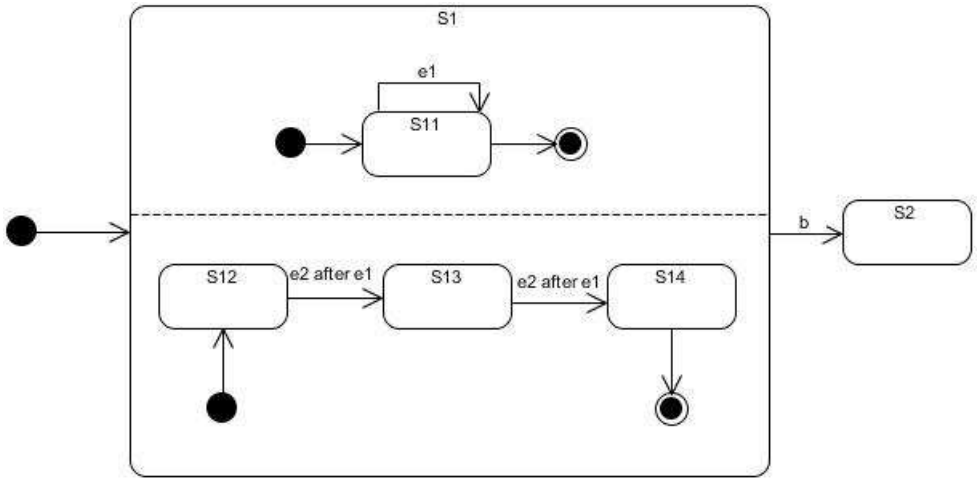


FIGURE 8.2 – Exemple d'explication pour l'élément syntaxique `after`

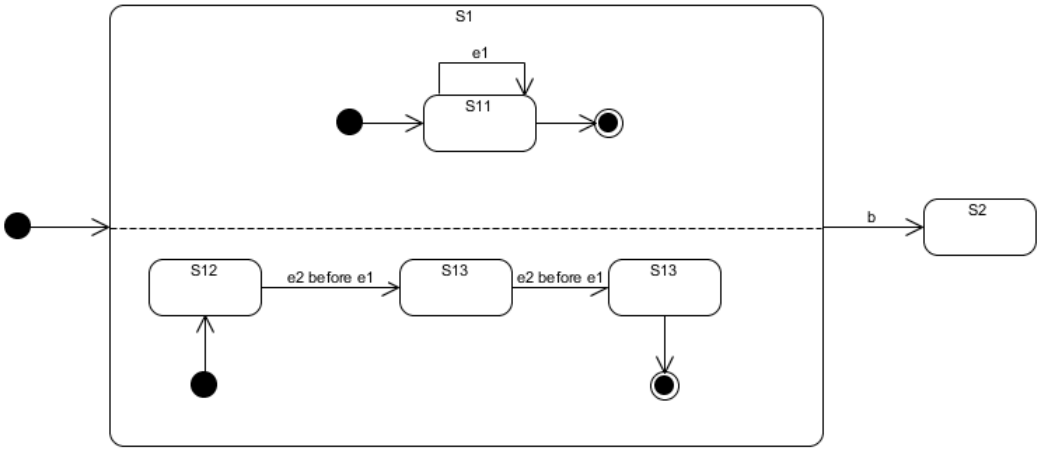


FIGURE 8.3 – Exemple d'explication pour l'élément syntaxique `before`

Description de la grammaire

Nous décrivons dans ce qui suit les différentes clauses de la grammaire proposée dans Figure 8.4. Nous supposons qu'il y a contrainte pour préciser que la clause *EventExpo* est destinée pour les expressions temporelles concernant les événements et que la clause *StateExp* est destinée pour les expressions temporelles concernant les états.

- *val* est une variable avec une valeur rationnelle représentant une valeur temporelle.
- *e*, *e1* et *e2* sont des événements.
- *TimeExp : := EventExp | StateExp*, une expression temporelle (*TimeExp*) qui peut être : (i) *EventExp* une expression temporelle par rapport aux événements (ii) *StateExp* une expression temporelle par rapport aux comportements do des états.
- *EventExp : := e TimedAnnotation | e LogicalAnnotation | e CombinationAnnotation*, une expression temporelle par rapport aux événements qui peut être : (i) *TimedAnnotation* une annotation temporelle qui utilise des contraintes quantitatives (c'est-à-dire `at`, `at_least`, etc) (ii) *LogicalAnnotation* une annotation logique qui utilise des contraintes qualitatives

```
Grammaire

TimeExp ::= EventExp | StateExp

EventExp ::= e TimedAnnotation | e LogicalAnnotation | e CombinationAnnotation

StateExp ::= do doExpression TimedAnnotation

TimedAnnotation ::= at val | at_least val | at_most val | between [val1,val2]

LogicalAnnotation ::= after e1 | before e1

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).

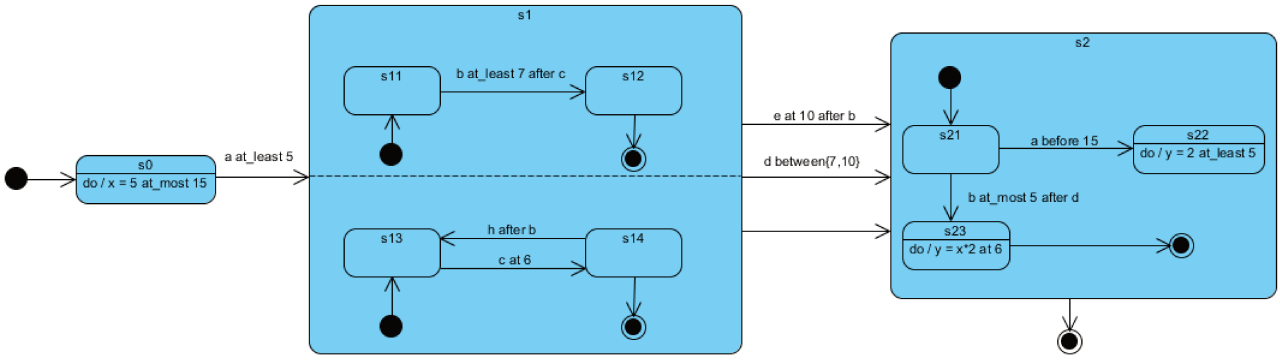


FIGURE 8.5 – Représentation des contraintes temporelles dans les diagrammes états-transitions

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) :

- `a at_least 5` représente une expression temporelle pour les événements. Cette expression est incluse dans la clause *EventExp : := e TimedAnnotation*. Le diagramme états-transitions de la Figure 8.5 comporte d'autres expressions incluses dans la même clause, telles que : `c at 6`, `d between`[7, 10].

- `b at_least 7 after c` représente une combinaison d'expression temporelle et logique. Cette expression est incluse dans la clause *EventExp : := e CombinationAnnotation*. Le diagramme états-transitions de la Figure 8.5 comporte également d'autres expressions incluses dans la même clause, telles que : `e at 10 after b`, `b at_most 5 after d`.

- `h after b` représente une expression logique. Cette expression est incluse dans la clause *EventExp : := e LogicalAnnotation*. Le diagramme états-transitions de la Figure 8.5 comporte une autre expression incluse dans la même clause : `a before 15`.

- `x = 5 at_most 15` représente une expression temporelle pour le comportement do de l'état `s0`. Cette expression est incluse dans la clause *StateExp : := do doExpression TimedAnnotation*. Le diagramme état-transitions de la Figure 8.5 comporte d'autres expressions incluses dans la même clause : `y = x * 2 at 6`, `y = 2 at_least 5`.

Gestion de l'échec de la garde temporelle

Concernant la prise en compte de l'échec d'une condition temporelle sur l'exécution du comportement do d'un état, c'est-à-dire comment le système va réagir dans le cas où la condition temporelle n'est pas vérifiée.

Une première solution est d'ajouter un état erreur qui modélise l'état du système s'il y a un échec d'une condition temporelle. Dans la phase de vérification du réseau de Petri coloré résultat de la traduction du modèle UML, il suffit de vérifier la présence de jetons dans la place correspondante à cet état erreur. Par exemple, dans la Figure 8.6 si la contrainte temporelle de la transition avec l'événement `ticketInserted` n'est pas respectée alors la transition de l'état `carPresent` vers `Failure` sera franchie. Quand l'échec concerne la condition temporelle d'une transition alors c'est l'état précédant la transition qui aura un lien avec l'état `Failure` (par exemple l'état `carPresent`).

Une seconde solution est de ne pas ajouter d'états d'échecs et de ne pas gérer l'échec de la condition temporelle. La détection de l'échec de la condition temporelle sera géré au moment de la vérification (après transformation vers les réseaux de Petri), car ça sera le model checker qui le fera.

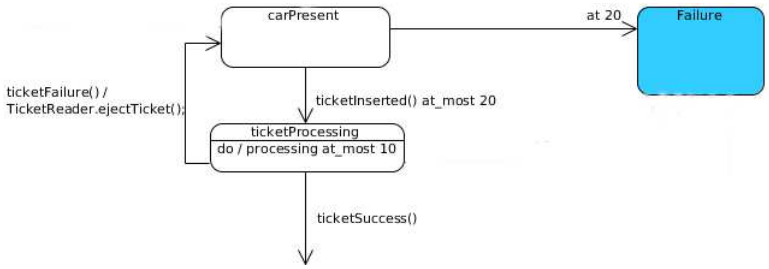


FIGURE 8.6 – Gestion de l'échec des contraintes temporelles

Exprimer le temps dans les états ou dans les transitions ?

Une question qu'on peut se poser sur l'expression d'une contrainte temporelle, est-ce qu'on doit l'exprimer dans les états ou dans les transitions ?

Dans le cas où le temps concerne le passage du système d'un état à un autre et donc par conséquent l'occurrence de l'événement, alors nous exprimons le temps dans les transitions. Par exemple, dans la Figure 8.7, la transition étiquetée avec l'événement `carLeaves` comporte une contrainte temporelle. La présence de cette contrainte est parce quelle concerne l'occurrence de l'événement `carLeaves` et non pas le comportement do de l'état `gateRaised` (qui dans l'exemple ne possède pas un comportement do).

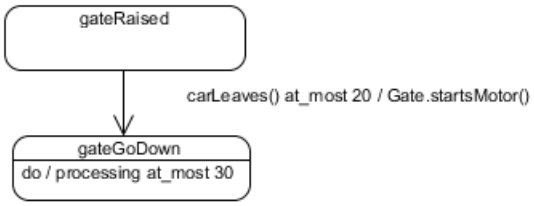


FIGURE 8.7 – Le temps dans les transitions

Dans le cas où le temps concerne l'exécution d'un comportement dans un état donné et non pas l'occurrence d'un événement, alors nous exprimons le temps dans les états en définissant la contrainte par rapport au comportement do. Par exemple dans la Figure 8.8, l'état `ticketProcessing` comporte une contrainte temporelle concernant son comportement do (c'est-à-dire le comportement do `processing` qui a pour contrainte 30 secondes). La contrainte temporelle est exprimée dans l'état et non pas dans la transition (étiquetée avec l'événement `ticketSuccess`) car cela concerne l'exécution du comportement do et non pas l'occurrence de l'événement. La présence de la contrainte dans l'état permet également d'éviter de gérer la contrainte dans la transition étiquetée avec l'événement `ticketSucess` et dans la transition étiquetée avec l'événement `ticketFailure`.

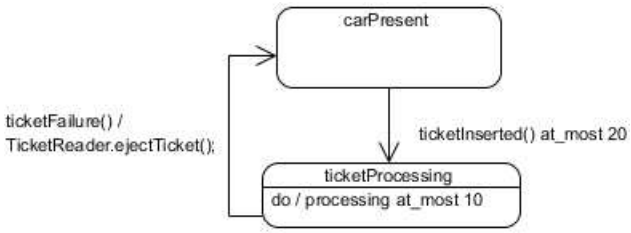


FIGURE 8.8 – Le temps dans les états

Mesurer le temps Les horloge ne sont pas utilisées dans notre annotation des diagrammes états-transitions. Cependant, il est possible d'utiliser une horloge globale pour chronométrer le temps globale du modèle mais également définir des horloges locales pour chaque état. L'absence d'horloge globale dans nos annotations temporelles est due à la phase de vérification de notre modèle qui se fait dans les réseaux de Petri colorés temporisés (après la traduction) et non pas dans les diagrammes états-transitions.

Unités de temps

La diversité des contraintes temporelles dans les systèmes temporisés implique l'utilisation de différentes unités de temps. Nous considérons un ensemble d'unités qui accompagnent les expressions temporelles dans les diagrammes états-transitions. L'unité de temps par défaut est

la seconde et dans le cas où il n'y a pas d'unité dans l'expression temporelle cela implique que l'unité est la seconde.

Les unités prises en compte dans cette définition sont les suivantes :

- *s* secondes qui est l'une des mesures de base
- *ms* milliseconde définie par un facteur de 0,001 par rapport aux secondes
- *ns* nanoseconde définie par un facteur de 0,001 par rapport aux millisecondes
- *min* minutes définie par un facteur de 60 par rapport aux secondes
- *h* heure définie par un facteur de 60 par rapport aux minutes
- *day* jour défini par un facteur de 24 par rapport aux heures

D'autres travaux considèrent les unités de temps dans leurs annotations tels que MARTE. Ce dernier comprend plus d'unités telles que les tic. Nous ne considérons pas les tic dans les unités de temps que nous prenons en compte car c'est dédié à la représentation des unités dans le cas de processeurs. Étant donné que nos études de cas ne considèrent pas les tic, nous les avons pas considéré dans nos unités de temps.

Les unités de temps dans le model checking

Dans la phase de vérification les unités de temps ne sont pas considérées et par conséquent dans le cas de la présence de ces unités dans les diagrammes états-transitons cela peut créer des problèmes. C'est pour cette raison qu'il faut faire une conversion des unités vers l'unité la plus petite du modèle pour que le modèle soit homogène.

Comparaisons

Afin de renforcer la raison de notre choix nous proposons une comparaison des différents éléments pris en considération selon notre classification avec d'autres travaux (voir le Tableau 8.2).

$\surd$: la contrainte est considérée

$\times$: la contrainte peut être modélisée par l'approche

Le temps dans UML Nous avons présenté dans la Section 3.3 les contraintes temporelles qui sont considérés dans la spécification d'UML. Les deux éléments syntaxiques utilisés pour la représentation des contraintes temporelles dans UML sont `at` et `after`. L'élément syntaxique `after` correspond à l'élément `at` que nous utilisons pour la représentation de la latence. L'élément syntaxique `at` correspond à la représentation du temps absolu vis à vis l'occurrence des événements. Nous avons la possibilité de représenter cette contrainte (représentée par l'élément syntaxique `at` dans UML) en supposant qu'il y a un événement de début qui apparaît et à partir du quel nous pouvons dire un événement e apparaît un temps T après cet événement. Par exemple, si l'événement de début est t_0 qui intervient au temps 0, alors la contrainte e `at` n (en utilisant le `at` d'UML) sera traduite par e `at` n `after` t_0 dans notre traduction. Cette sémantique 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

MARTE [?]

La gestion du temps présentée dans MARTE est différente de ce que nous considérons dans cette thèse. MARTE base sa prise en compte du temps par la gestion des horloges physiques et logiques et par la prise en compte de trois contraintes principales.

- Contraintes basées sur la coïncidence (*Coincidence-based constraints*)
- Contraintes basées sur la précédence (*Precedence-based constraints*)
- Combinaison des relations de coïncidence et de précédence (*Mixed constraints*)

Contraintes

Le Tableau 8.2 montre que nous considérons dans notre travail les contraintes temporelles les plus utilisés. Par exemple, notre représentation de la séquence en utilisant l'élément syntaxique `after` est équivalente à la clause 12 (*CyclicTimedResponses*) représentée dans [**?**] (le travail dans [**?**] reprend le même esprit de la représentation des contraintes temporelles que celui dans [**?**]).

Nous sommes convaincu qu'il existe d'autres contraintes que nous ne considérons pas dans nos annotations (comme par exemple la gestion des horloges et toutes les contraintes autour, etc.) car nous souhaitons prendre en compte les contraintes les plus utilisés et garder notre syntaxe à la fois simple et expressive.

Exemples : prise en compte des contraintes temporelles

Barrière de parking La Figure 8.9 montre l'introduction des contraintes temporelles sur un exemple de barrière de parking. Comme montré dans l'exemple, une seule voiture est traitée à la fois, et le temps entre l'arrivée d'une voiture et son départ ne doit pas dépasser 150 secondes. Étant donné que le passage de la voiture par la barrière est composée de plusieurs étapes, le temps que la voiture ne dois pas dépasser dans son passage sera réparti sur ces différentes étapes. En effet, la valeur 150 secondes a été obtenue en additionnant les différentes valeurs des contraintes dans le diagramme. Cette addition est possible car il n'y a que des contraintes `at_most` où toute transition avec cette contrainte doit être franchie au plus une durée de temps. Dans le cas où il y a d'autres contraintes (telles que `at` ou `at_least`) l'addition sera impossible. Par exemple, l'insertion du ticket doit se faire au plus 20 secondes. La détection de l'échec de la contrainte temporelle se fera dans la phase de vérification. La prise en compte des contraintes temporelles dans l'exemple de la Figure 8.9 se manifeste de deux manières. Dans la première nous gérons le temps d'apparition des événements. Par exemple, afin de définir une condition temporelle sur l'insertion d'un ticket nous utilisons la contrainte suivante `ticketInserted`() `at_most` 20, qui veut dire "l'insertion du ticket doit se faire au plus 20 secondes après l'arrivée de la voiture". Cette contrainte est définie pour l'occurrence de l'événement car le temps ici concerne le moment où le système change d'état.

Dans le seconde nous gérons le temps d'exécution du comportement `do` de chaque état. Par exemple, afin de définir une condition temporelle concernant le traitement d'un ticket nous utilisons dans l'état `ticketProcessing` la contrainte suivante `do`/`processing` `at_most` 10, qui veut dire "le traitement du ticket (représenté ici par le comportement do `processing`) doit se faire au plus 10 secondes après l'activation de l'état `ticketProcessing`". Autrement dit, le système peut rester dans l'état `ticketProcessing` au plus 10 secondes.

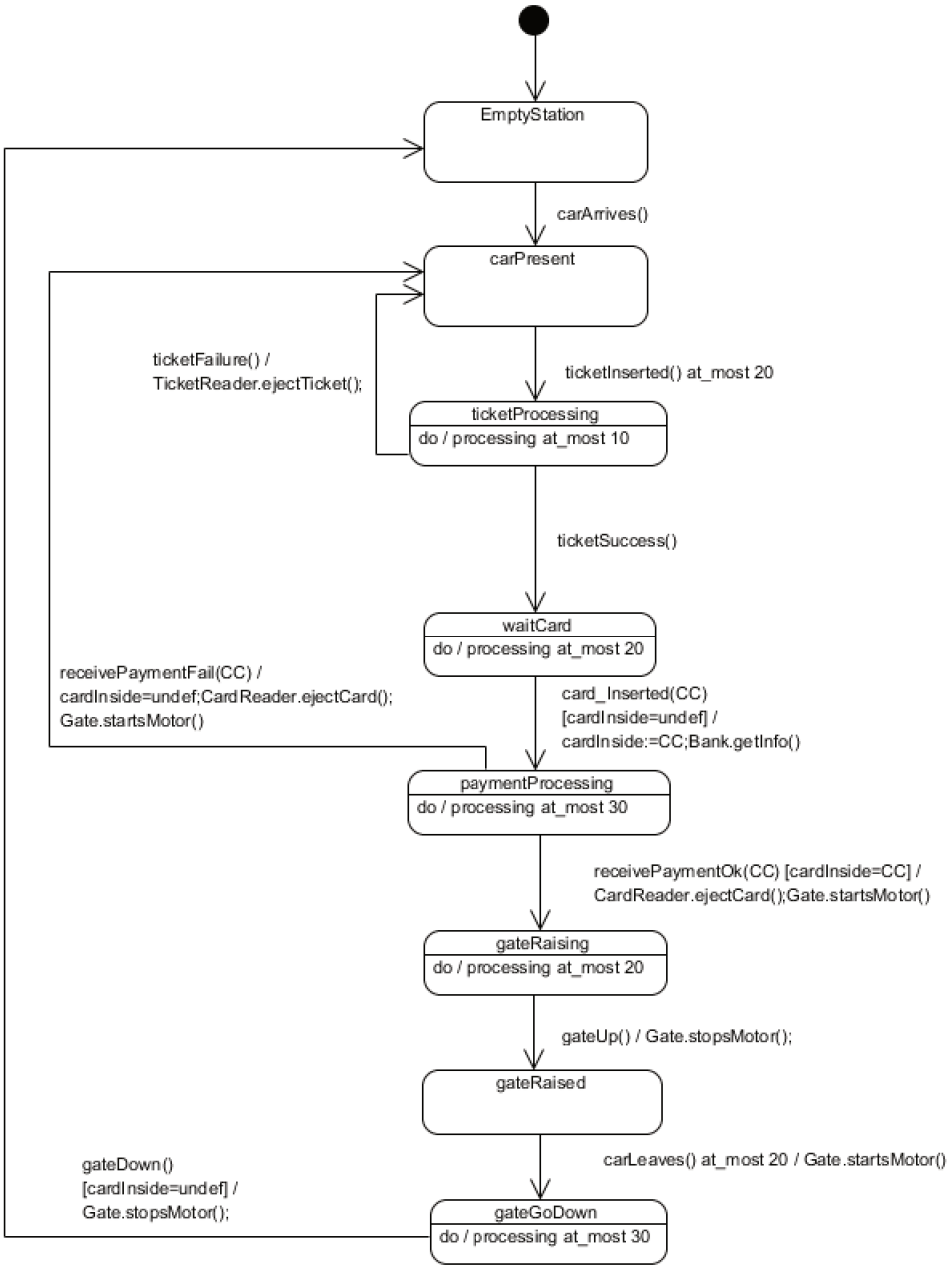


FIGURE 8.9 – Exemple de la barrière de parking temporisé

Bibliothèque La Figure 8.10 montre l'introduction des contraintes temporelles sur l'exemple de la bibliothèque. Dans la modélisation de l'exemple de la bibliothèque (voir Chapitre 4) nous avons remarqué que certaines parties nécessitaient des contraintes temporelles. De ce fait nous avons intégré le temps dans l'exemple et cela dans deux parties principales. La première dans l'état `tagged`, le bibliothécaire ne peut pas retirer un livre plus 168 heures (7 jours). La seconde vis-à-vis de l'événement `bringBack`, l'utilisateur ne peut pas garder un livre plus de 168 heures. Cela est modélisé dans la transition étiquetée avec l'événement `bringBack` avec la contrainte `bringBack at_most 168h after lend()`. Autrement dit, l'événement `bringBack` doit apparaître au plus 168 heures après l'occurrence de l'événement `lend`. L'erreur engendrée en cas de dépassement dans la contrainte de temps est prise en compte lors de la phase de vérification et non pas dans le diagramme états-transition.

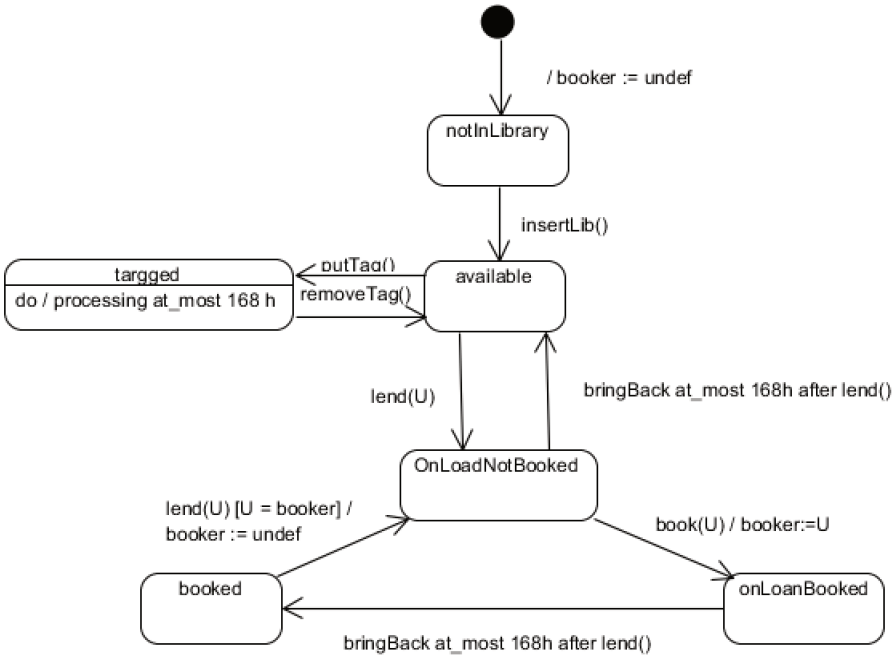


FIGURE 8.10 – L'exemple de la bibliothèque temporisé

Location de scooter

Nous présentons dans ce qui suit la description textuelle de l'exemple de *location de scooter* qui sera utilisée pour la conception du diagramme états-transitions correspondant.

> *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.*

Afin de modéliser cet exemple, nous avons constaté qu'il est possible de considérer différents points de vue :

- Point de vue la station : nous pouvons modéliser cet exemple en considérant le fonctionnement de la station comme acteur principal.
- Point de vue scooter : nous pouvons également considérer le scooter et les différentes étapes de son allocation comme acteur principal.
- Point de vue client : enfin, nous pouvons considérer le client et ce qu'il peut faire dans une station comme acteur principal.

Ce qui nous intéresse dans cet exemple sont les différentes étapes de la location du scooter et donc par conséquent modéliser cet exemple du point de vue scooter. La Figure 8.11 montre le diagramme états-transitions correspondant à l'exemple du scooter en prenant en compte la gestion des contraintes temporelles.

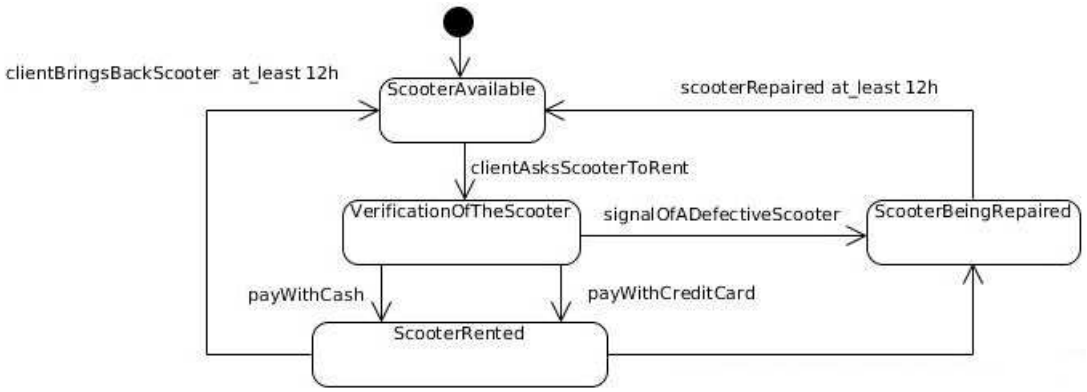


FIGURE 8.11 – L'exemple de location de scooter temporisé (version 1)

Nous avons constaté lors de la modélisation des contraintes temporelles que les deux événements venant de l'état `ScooterRented` ont une contrainte temporelle qui est la même. Afin d'éviter de gérer cette contrainte séparément pour chaque événement et par conséquent augmenter la complexité, nous avons exprimé la contrainte temporelle au sein de l'état `ScooterRented`. Cette contrainte concerne le fait que le scooter soit loué pour une durée de 12h. Dans la Figure 8.12 la contrainte temporelle est supprimée des deux transitions venant de l'état `ScooterRented` et

ajoutée dans l'état `ScooterRented` en considérant que le comportement do de cet état (qui est l'utilisation du scooter) ne dépasse pas 12h.

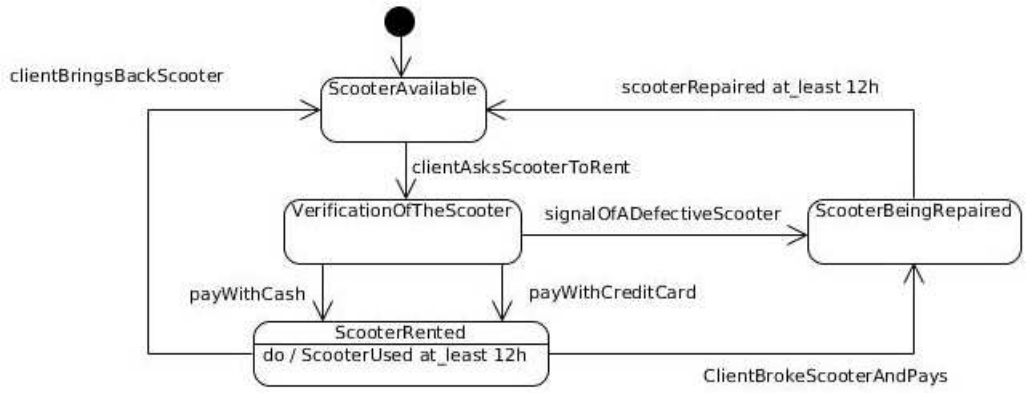


FIGURE 8.12 – L'exemple de location de scooter temporisé (version 2)

Le temps dans les réseaux de Petri colorés

Il est possible dans les réseaux de Petri colorés d'ajouter des contraintes temporelles et cela afin de modéliser et vérifier les systèmes temps-réel. En effet, on s'intéresse ici à l'ajout du temps sur les places. La différence majeure entre un réseau de Petri coloré et un réseau de Petri coloré temporisé est la présence d'une valeur appelée estampillage de date (*time stamp*). Cela signifie que le marquage d'une place qui a des jetons avec estampillage de date est maintenant un multi-ensemble temporisé spécifiant les éléments du multi-ensemble avec leur nombre d'occurrence, leur valeur et leur estampillage de date. Par exemple dans la Figure 8.13, le nombre d'occurrence du jeton 1‘(2, “g and An″)@0 est 1, la valeur du jeton est (2, “g and An″) et l'estampillage de date est 0. La valeur de l'estampillage de date peut être un entier ou un réel positif. L'estampillage de date permet de savoir à quel moment le jeton peut être consommé de la place avec une transition franchissable (ayant comme source cette place). La valeur des jetons peut avoir un estampillage de date dans le cas où la place contenant ces jetons a un type temporisé.

Notons que l'estampillage de date par défaut est *@0* et par conséquent les jetons avec cet estampillage peuvent être consommés à n'importe quel moment.

Le modèle *CPN* possède une horloge globale qui représente le temps du modèle. L'exécution d'un modèle *CPN* avec du temps est contrôlée avec cette horloge globale et cela fonctionne de la même manière que la gestion de la file des événements utilisée dans la plupart des simulations des événements discrets. La valeur de cette horloge ne change pas tant qu'il y a des transitions qui sont franchissables, c'est-à-dire des transitions ayant des places sources avec un type temporisé ou non. La valeur de l'horloge globale change s'il n'y a pas de transition franchissable et sa valeur prend alors celle du jeton ayant l'estampillage de date le plus proche. Chaque marquage se trouve dans un intervalle fermé dans le modèle *CPN* temporisé.

Notons qu'il est possible d'avoir un conflit dans le cas où il y a des transitions franchissables en même temps. Notons également que la distribution des jetons avec leur estampillage de date dans les places ainsi que la valeur de l'horloge globale est appelée *marquage temporisé*.

La Figure 8.13 modélise l'exemple d'envoi de paquets avec accusé de réception (l'exemple est présenté dans [**?**]). Nous remarquons dans cet exemple qu'il y a un ensemble de jetons sans présence de temps, c'est-à-dire que tous les estampillages de date sont à 0 par défaut. Le changement de valeur des estampillages de date se fait grâce aux expressions dans le compartiment temporel des transitions ou aux expressions dans les arcs. Par exemple, dans le compartiment temporel de la transition SendPacket il y a l'expression $@ + 9$. Cela veut dire qu'au moment de franchir la transition l'estampillage de date du jeton consommé sera augmenté de 9. Notons que la valeur de l'horloge globale de ce modèle est initialement égale à 0 (au début de la vérification).

La Figure 8.14 montre le réseau de Petri coloré de la Figure 8.13 avec le marquage initial. Notons que les transitions franchissables sont celles entourées d'un halo vert. Dans le marquage initial de cet exemple, seule la transition SendPacket est franchissable car :

- Il y a un jeton dans la place NextSend ($n = 1$), il y a un jeton dans la place Send ($n = 1$) et il y a un jeton dans la place Wait (ces places sont les places source de la transition SendPacket).
- La valeur de chaque jeton de ces trois places comporte un estampillage de date égal à 0 et qui est égal à la valeur de l'horloge globale (c'est-à-dire 0).

La Figure 8.15 montre l'étape suivante après le franchissement de la transition SendPacket.

Après le franchissement de la transition SendPacket nous remarquons plusieurs changements comme suit :

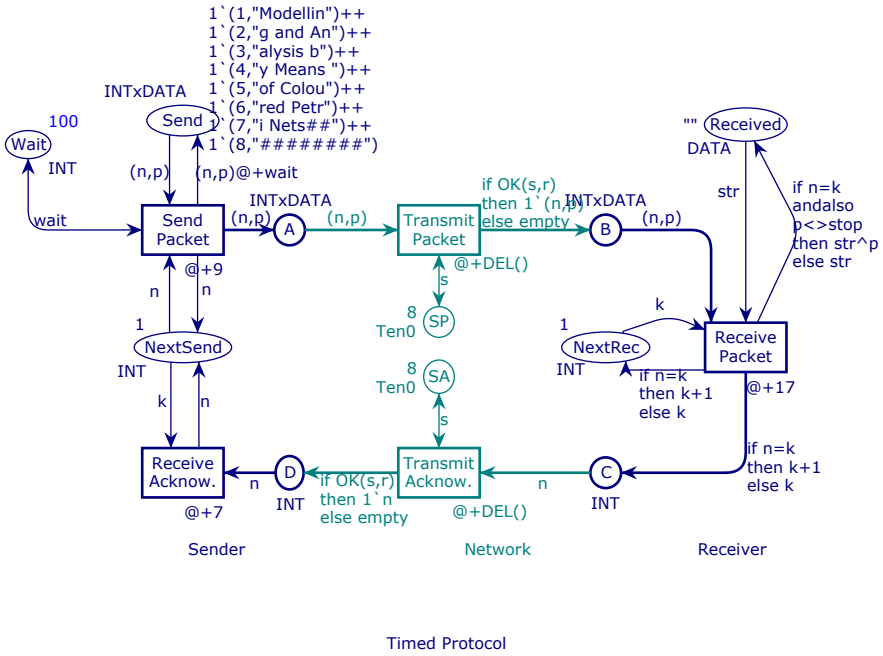


FIGURE 8.13 – Exemple d'un réseau de Petri coloré avec du temps ([?])

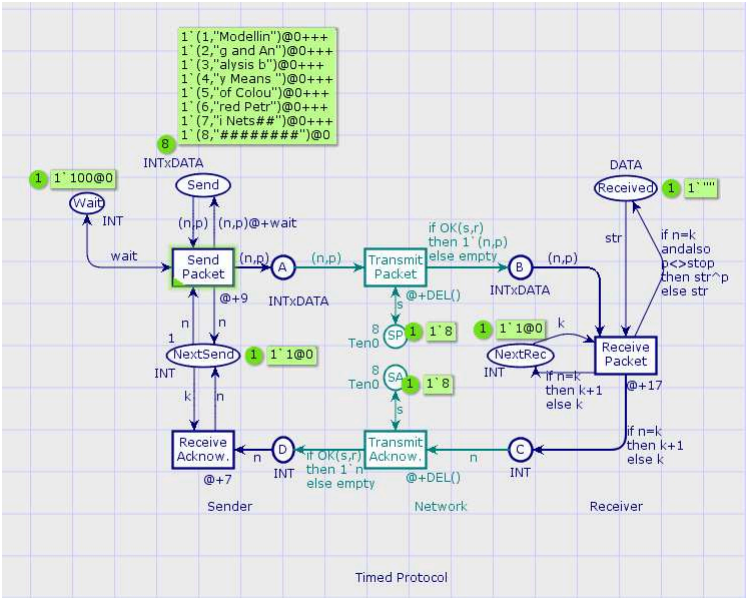


FIGURE 8.14 – Exemple d'un réseau de Petri coloré avec du temps avec marquage ([?])

- La consommation du jeton $(1, \text{"Modellin"})@0$ dans la place Send et sa remise dans la même place mais avec un estampillage différent $(1, \text{"Modellin"})@109$. Ce changement d'estampillage de date est dû à l'expression $@ + 9$ dans le compartiment temporel de la transition SendPacket, mais aussi à l'expression dans l'arc allant de la transition SendPacket à la place Send $((n, p)@ + \text{wait})$ avec $\text{wait} = 100$).

- La consommation du jeton $100@0$ dans la place Wait et sa remise dans la même place mais avec un estampillage différent $100@9$. Ce changement d'estampillage de date est dû à l'expression $@ + 9$ dans le compartiment temporel de la transition SendPacket.

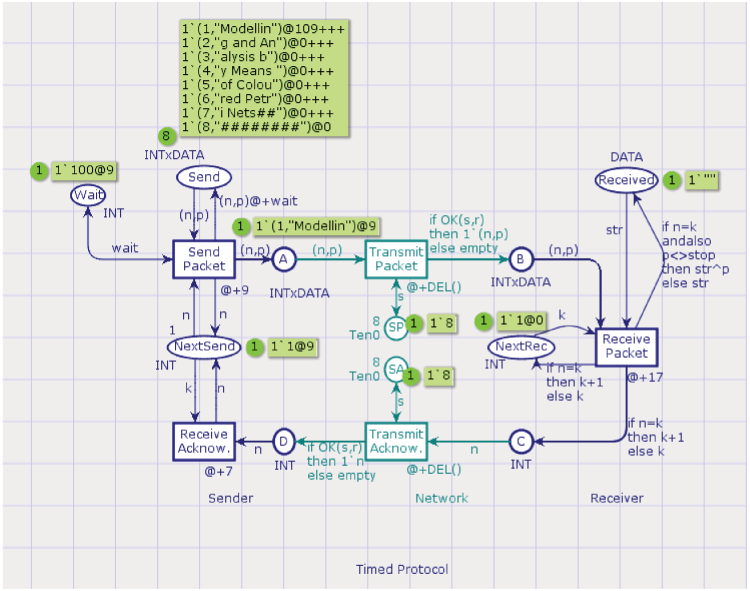


FIGURE 8.15 – Franchissement de la transition SendPacket

- La production du jeton (1,"Modellin")@9 dans la place A.

Après le franchissement de la transition SendPacket, on ne peut plus franchir de transition. En effet, la transition SendPacket est infranchissable car l'estampillage de date du jeton 100 dans la place Wait est égal à 9. La transition TransmitPacket est infranchissable car l'estampillage de date du jeton (1,"Modellin") dans la place A est égal à 9. Le reste des transitions ne sont pas franchissables car il n'y a pas de jetons dans toutes leurs places source. Par conséquent, il est nécessaire d'incrémenter la valeur de l'horloge à 9, c'est-à-dire l'estampillage de date le plus proche afin que la transition TransmitPacket soit franchissable. La Figure 8.16 montre la valeur de l'horloge globale avant et après l'incrémentation de l'horloge globale.

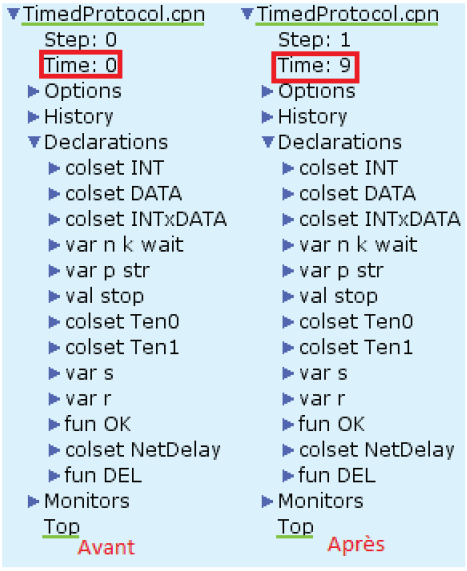


FIGURE 8.16 – Valeur de l'horloge globale avant et après le franchissement de la transition SendPacket

Une fois que la valeur de l'horloge globale est incrémentée, la transition TransmitPacket devient franchissable car :

- Il y a un jeton dans la place A.
- L'estampillage de date du jeton dans cette place est égal à la valeur de l'horloge globale, c'est-à-dire 9.

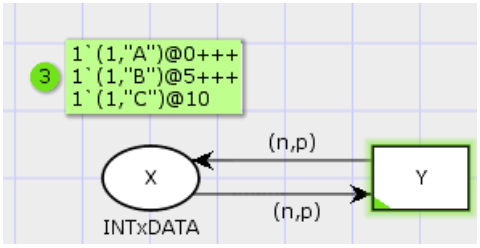


FIGURE 8.17 – Gestion des estampillages de date dans les transitions

La Figure 8.17 montre un simple exemple avec une place (ayant plusieurs jetons) et une transition qui consomme et produit dans cette place. Afin que la transition soit franchissable il suffit qu'un seul jeton possède un estampillage de date inférieur ou égal à la valeur de l'horloge globale. Par exemple, dans la place X il y a trois jetons, le premier avec un estampillage de date de 0, le second avec un estampillage de date de 5 et le dernier avec un estampillage de date de 10. La valeur de l'horloge globale est initialement égale à 0 et par conséquent la transition Y est franchissable car il y a au moins un jeton (celui avec l'estampillage de date égal à 0) qui a un estampillage de date inférieur ou égal à 0.

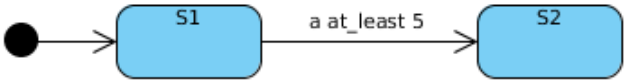
Introduction des contraintes temporelles dans la traduction

Afin de prendre compte les contraintes temporelles et de vérifier les systèmes temporisés, nous présentons dans cette section l'intégration des contraintes temporelles dans notre traduction. Nous utilisons les diagrammes états-transitions annotés avec du temps (présentés dans le Chapitre 8), ainsi que les réseaux de Petri colorés temporisés (présentés dans la Section 2.3).

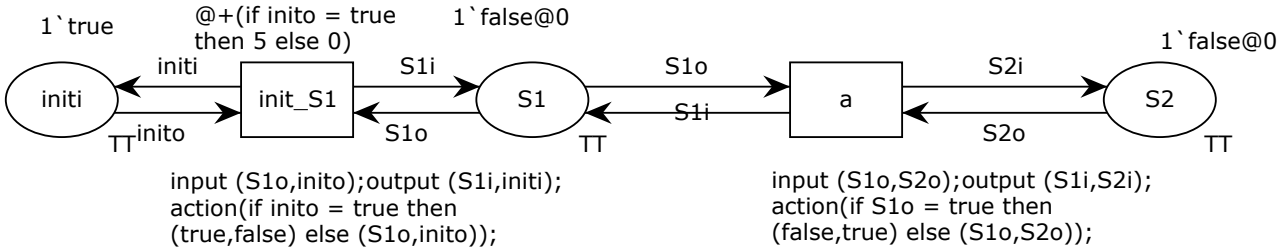
L'intégration du temps dans notre traduction est basée sur un ensemble de règle permettant de représenter chaque contrainte (sa syntaxe et sa sémantique) dans les réseaux de Petri colorés. Nous utilisons un ensemble de diagrammes états-transitions pour représenter les contraintes et nous donnons leurs correspondant dans les réseaux de Petri colorés temporisés.

Traduction du délai minimum

Le délai minimum est modélisé en utilisant l'élément syntaxique `at_least` et qui veut dire qu'un événement doit apparaître au minimum un délai donné après l'activation de son état source. L'exemple de la Figure 8.18 montre le diagramme états-transition (représentant le délai minimum) et sa traduction correspondante.



(a) SMD



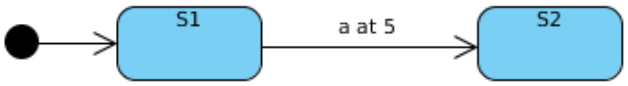
(b) CPN

FIGURE 8.18 – Traduction d'`at_least`

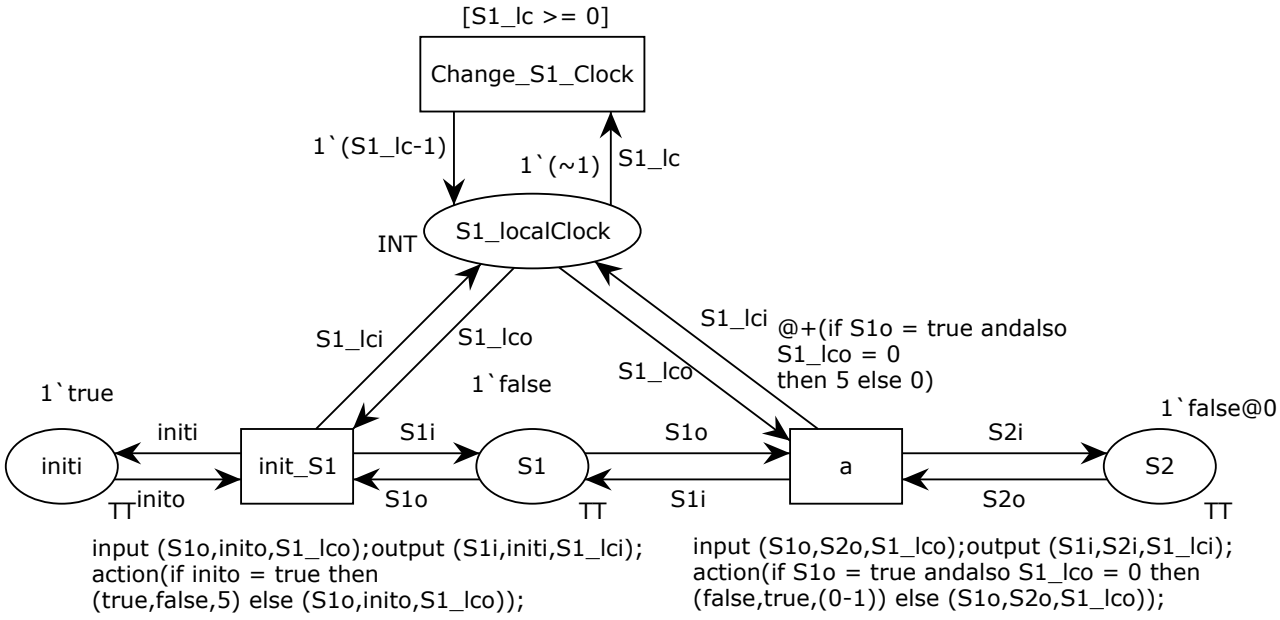
Dans l'exemple de la Figure 8.18a nous avons une transition étiquetée avec l'événement `a` (allant de l'état `S1` vers l'état `S2`). Cette transition ne peut pas être franchie que si l'événement `a` apparaît au minimum 5 secondes (ou plus) après l'activation de l'état `S1`. Rappelons qu'un jeton avec un estampillage de date dans les réseaux de Petri colorés temporisés ne peut être consommé que si son estampillage est inférieur ou égal à celui de l'horloge globale ; cette sémantique correspond à la sémantique du délai minimum. Par conséquent, nous traduisons cet exemple en utilisant les Algorithmes 1 et 2 et en ajoutant un jeton avec 5 unités de temps(l'estampillage de ce jeton est engendré grâce à la partie temporelle de la transition init_S1). Une fois que la place S1 sera active le jeton aura la valeur 5 et par conséquent la transition ne sera franchie que si l'horloge globale soit à la valeur 5 ou plus.

Traduction de la ponctualité

La ponctualité est modélisée en utilisant l'élément syntaxique `at` et qui veut dire qu'un événement doit apparaître exactement à un délai donné après l'activation de son état source. L'exemple de la Figure 8.19 montre le diagramme états-transition (représentant la ponctualité) et sa traduction vers les réseaux de Petri colorés temporisés correspondante.



(a) SMD



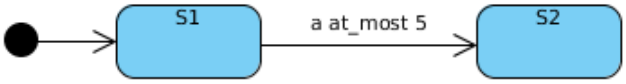
(b) CPN

FIGURE 8.19 – Traduction d'`at`

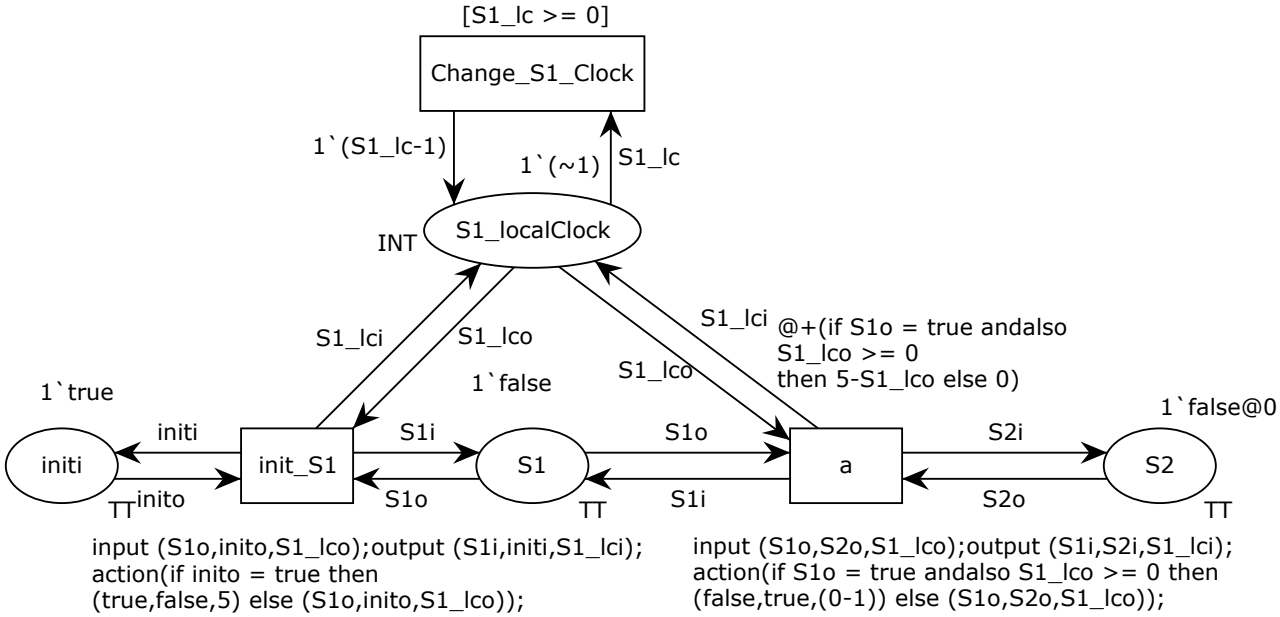
Dans l'exemple de la Figure 8.19a nous avons une transition étiquetée avec l'événement `a` (allant de l'état `S1` vers l'état `S2`). Cette transition ne peut pas être franchie que si l'événement `a` apparaît exactement 5 secondes après l'activation de l'état `S1`. Étant donné que la sémantique de la ponctualité est différente de la sémantique donnée pour la consommation des jetons avec un estampillage de date, nous ajoutons un mécanisme afin que la sémantique de cette contrainte (la ponctualité) soit respectée. Nous utiliserons un compteur (modélisé par la place S1_localClock) et qui contiendra la valeur 5 et une transition (modélisée avec Change_S1_Clock) qui permet de décrémenter la valeur stockée dans la place S1_localClock. La place S1_localClock est initialement à la valeur *-1* car la place S1 (concernée par le compteur) n'est pas encore active. Une fois que cette place est active la valeur du jeton dans la place S1_localClock sera à 5. La transition Change_S1_Clock pourra être franchie tant que la valeur du jeton est inférieure ou égale à 5. La transition a sera franchissable dans le cas où la place S1 comporte un jeton true (c'est-à-dire la place est active) et dans le cas où la valeur du jeton dans la place S1_localClock est à 0. Dans le cas où la valeur du jeton dans la place S1_localClock devient négative il ne sera plus possible de franchir la transition a.

Traduction du délai maximum

Le délai maximum est modélisé en utilisant l'élément syntaxique `at_most` et qui veut dire qu'un événement doit être apparaître au maximum à un délai donné après l'activation de son état source. L'exemple de la Figure 8.20 montre le diagramme états-transitions (représentant le délai maximum) et sa traduction vers les réseaux de Petri colorés temporisés correspondante.



(a) SMD



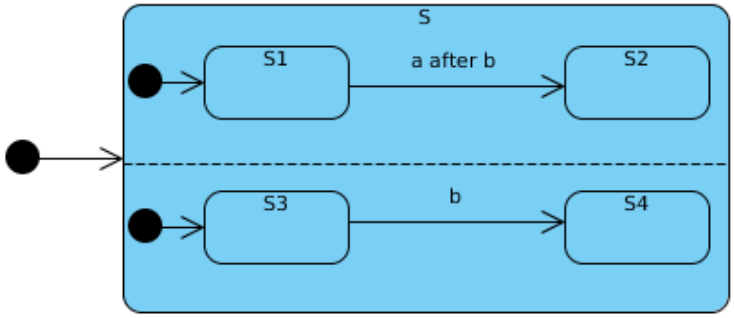
(b) CPN

FIGURE 8.20 – Traduction d'`at_most`

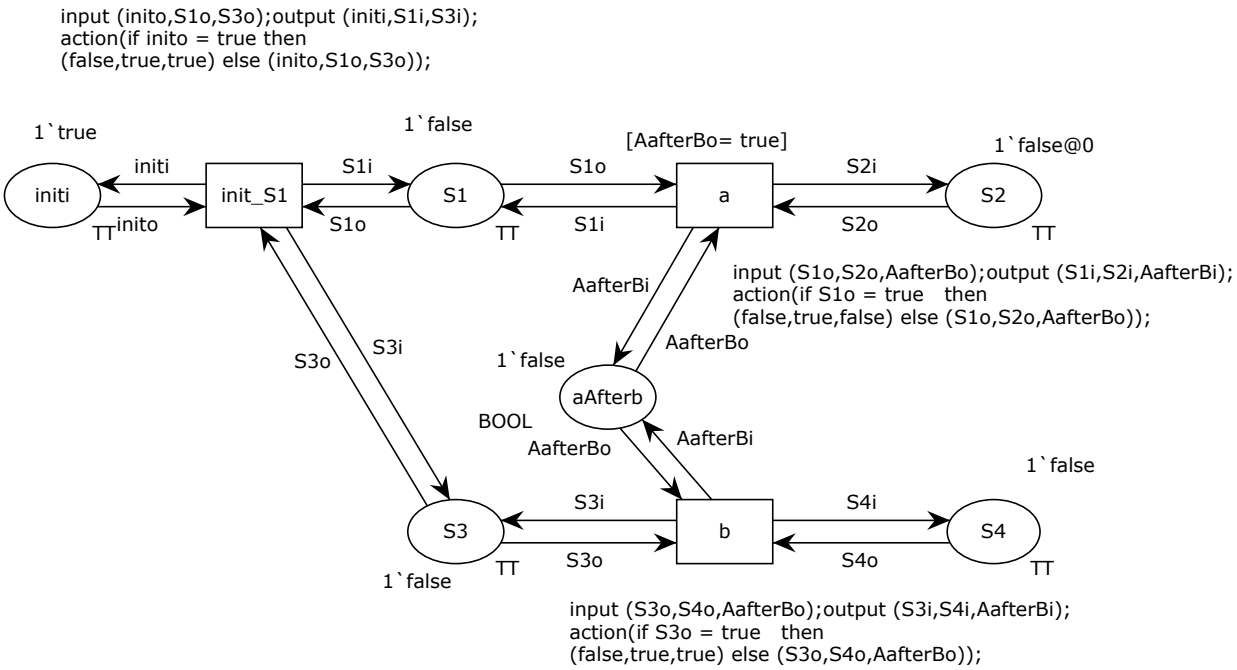
Dans l'exemple de la Figure 8.19a nous avons une transition étiquetée avec l'événement `a` (allant de l'état `S1` vers l'état `S2`). Cette transition ne peut pas être franchie que si l'événement `a` apparaît au plus 5 secondes après l'activation de l'état `S1`. Étant donné que la sémantique du délai maximum est différente de la sémantique donnée pour la consommation des jetons avec un estampillage de date, nous reprenons le mécanisme utilisé pour la ponctualité afin que la sémantique de cette contrainte (le délai maximum) soit respectée. Nous utiliserons la place S1_localClock (contenant initialement un jeton avec la valeur *-1*) et la transition Change_S1_Clock pour représenter le mécanisme (celui du compteur). La différence avec le mécanisme utilisé dans la ponctualité est la garde dans la transition `a`, c'est-à-dire au lieu d'avoir comme garde S1_lco = 0 nous avons la garde S1_lco <= 5 (S1_lco est la variable transportant le jeton de la place S1_localClock). La transition `a` sera franchissable dans le cas où la place S1 comporte un jeton true (c'est-à-dire la place est active) et dans le cas où la valeur du jeton dans la place S1_localClock est supérieure ou égale 0 (cela est modélisé par la garde S1_localClock >= 0 dans le code segment de la transition `a`). Dans le cas où la valeur du jeton dans la place S1_localClock devient négative il ne sera plus possible de franchir la transition `a`.

Traduction de la séquence d'événements

La séquence d'événements est modélisée en utilisant l'élément syntaxique `after` et qui veut dire qu'un événement doit être apparaître après l'occurrence d'un autre. L'exemple de la Figure 8.21 montre le diagramme états-transitions (représentant la séquence d'événement) et sa traduction vers les réseaux de Petri colorés temporisés correspondante.



(a) SMD



(b) CPN

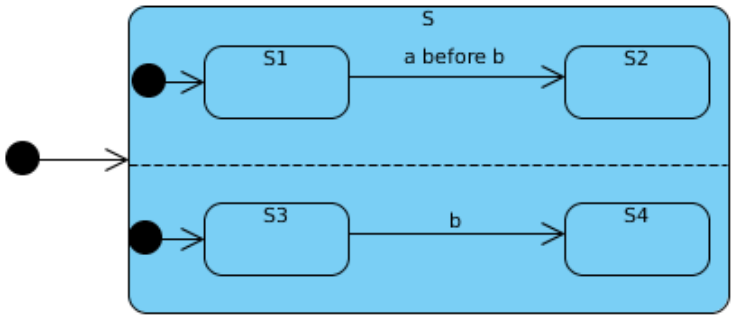
FIGURE 8.21 – Traduction d'`after`

Dans l'exemple de la Figure 8.21a nous avons une transition étiquetée avec l'événement `a` (allant de l'état `S1` vers l'état `S2`) et une transition étiquetée avec l'événement `b` (allant de l'état `S3` vers l'état `S4`). La transition avec l'événement `a` ne peut être franchie que si la transition avec l'événement `b` soit déjà franchie. Dans la Figure 8.21 nous présentons le réseaux de Petri coloré résultat de la traduction de l'exemple dans la Figure 8.21a. Afin de prendre en compte le séquencement des deux événements `a` et `b` nous utilisons une place supplémentaire aAfterb qui comporte initialement un jeton de valeur *false* (1‘false). Tant que la valeur du jeton dans cette place est à *false* la transition `a` est infranchissable (cela est modélisé par la présence de la garde [AafterBo = true]). Une fois que la transition `b` est franchie, la valeur du jeton change à *true* et la transition `a` devient franchissable. La valeur du jeton est remise à *false* une fois que la transition `a` est franchie. Notons que dans notre traduction pour la prise en compte des priorités entre événements (voir la Section 5.9), nous définissons une transition pour chaque ensemble de transitions étiquetées avec le même événement. Par conséquent, la définition que nous présentons concernant le séquencement de transitions est prise en compte. En effet, la transition `a` doit être franchie à la dernière occurrence de l'événement `b` ; et étant donné qu'il y a une seule transition

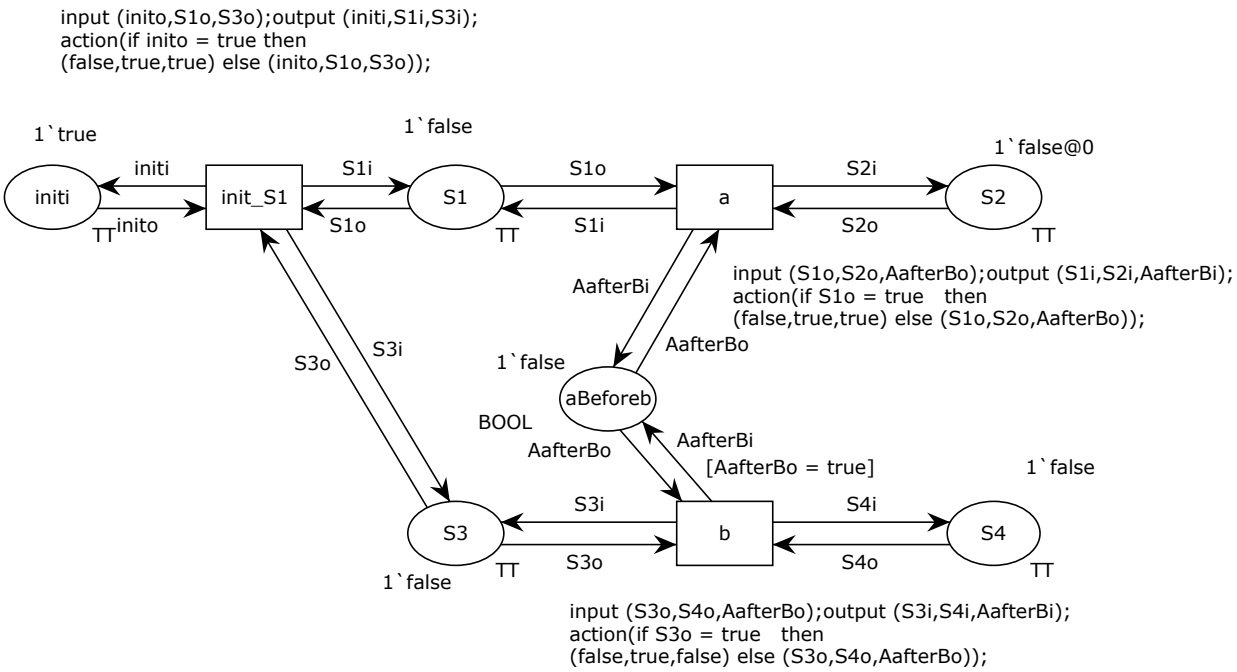
pour l'événement `b` alors la définition est respectée.

Traduction de la précédence

La précédence d'événements est modélisée en utilisant l'élément syntaxique `before` et qui veut dire qu'un événement doit être apparaître avant un autre. L'exemple de la Figure 8.22 montre le diagramme états-transitions (représentant la précédence d'événement) et sa traduction vers les réseaux de Petri colorés temporisés correspondante.



(a) SMD



(b) CPN

FIGURE 8.22 – Traduction de `before`

Dans l'exemple de la Figure 8.22a nous avons une transition étiquetée avec l'événement `a` (allant de l'état `S1` vers l'état `S2`) et une transition étiquetée avec l'événement `b` (allant de l'état `S3` vers l'état `S4`). La transition avec l'événement `b` ne peut être franchie que si la transition avec l'événement `a` soit déjà franchie. Dans la Figure 8.21 nous présentons le réseaux de Petri coloré résultat de la traduction de l'exemple dans la Figure 8.21a. Le mécanisme pour gérer la précédence des événements est le même que celui pour le séquencement (présence de la place supplémentaire et utilisation de gardes).

Conclusion

Nous avons présenté dans cette section un travail préliminaire concernant l'intégration des contraintes temporelles dans ce travail de thèse. Nous proposons une syntaxe et une sémantique pour annoter les diagrammes états-transitions avec du temps. Nous présentons ensuite un début d'intégration des contraintes temporelles dans notre traduction des diagrammes états-transitions 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.

Perspectives à court terme

Implémentation Nous souhaitons améliorer le prototype UML2CPN que nous avons présenté dans ce document pour automatiser notre traduction ; cela afin de vérifier formellement des diagrammes états-transitions plus complexes mais aussi pour étudier la complexité de notre traduction pour l'optimiser.

Intégration avec d'autres diagrammes Une autre perspective importante est l'intégration des diagrammes d'activités dans la traduction des diagrammes états-transitions UML. La combinaison des deux diagrammes permettra la modélisation d'autres aspects des systèmes, tels que la représentation des données. Il y a par exemple des travaux précédents que nous pouvons utiliser pour la formalisation des diagrammes d'activités, par exemple, en utilisant les réseaux de Petri colorés [**?**]. Nous pouvons également nous inspirer du travail qui traduit un ensemble de la syntaxe des machines à états SysML et les diagrammes d'activités vers CSP [**?**].

Extension de la méthode de spécification Le travail que nous présentons dans cette thèse à propos de la méthode de spécification considère un ensemble d'éléments syntaxiques des diagrammes états-transitions. Par exemple, les états simples, les états composites, les transitions (avec événements, gardes et comportements). Cependant, une perspective importante serait de prendre en compte l'ensemble des éléments syntaxiques considérés dans la traduction. Cela permettrait, une fois que la méthode de spécification a été utilisée, de traduire le diagrammes états-transitions (résultat de la méthode) vers les réseaux de Petri colorés (en utilisant notre traduction).

Contraintes temporelles Un autre point important à prendre en compte dans nos perspectives est l'intégration complète du temps dans nos approches. En effet, nous avons présenté dans cette thèse un travail préliminaire afin de considérer les contraintes temporelles et par conséquent vérifier les systèmes temporisés. Nous souhaitons compléter l'intégration du temps dans nos approches.

Études de cas Une autre perspective que nous souhaitons effectuer est d'élargir l'ensemble des études de cas auxquelles nous appliquons nous approches. En effet, dans cette thèse nous détaillons que deux études de cas (la barrière de péage et le lecteur de CD). La prise en compte de plusieurs études de cas permettra de démontrer que nous considérons un ensemble large des systèmes dans nos approches.

Extension de la traduction Nous souhaitons étendre notre traduction en prenant en compte l'ensemble des éléments syntaxiques des diagrammes états-transitions. Nous pensons que les éléments considérés dans notre thèse sont importants et suffisent à modéliser la plus part des systèmes. Cependant prendre l'ensemble des éléments syntaxiques est un ajout non négligeable afin de considérer l'intégralité des systèmes.

Perspectives à long terme

Comparaison avec d'autres outils Nous souhaitons une fois le développement de notre outil UML2CPN de le comparer avec d'autres outils. La comparaison consiste à utiliser les mêmes diagrammes états-transitions en entrée et comparer le résultat de la traduction. Faire une telle

comparaison (en termes d'aspects syntaxiques considérés, de sémantique définit pour l'outil, de performances et en termes de vérification pouvant être effectuée) sera une perspective intéressante que nous souhaitons effectuer.

Équivalence de sémantiques Nous souhaitons pour une perspective à long terme prouver formellement l'équivalence entre le modèle SMD d'entrée et le modèle CPN de sortie ; mais la sémantique présentée par l'OMG des diagrammes états-transitions est semi-formelle. Cependant, nous pouvons utiliser la sémantique opérationnelle des diagrammes états-transitions présentée dans [**?**].

Combinaison d'outils Nous avons présenté dans ce travail de thèse deux outils, à savoir *EasySM* qui implémente la méthode de spécification et UML2CPN qui implémente la traduction. Afin d'automatiser le processus de modélisation et de vérification (c'est-à-dire l'application de la méthode de spécification ensuite la traduction vers les réseaux de Petri colorés), nous souhaitons créer un outil qui combine *EasySM* et UML2CPN. Le nouveau outil sera capable de prendre en entrée un diagramme de classes, des observateurs d'états (avec leurs invariants), des événements (avec leurs conditions/réactions) et donner directement (i) Un "Ok" dans le cas où le système est vérifiée correctement (ii) Un contre-exemple dans le cas où le système n'est pas vérifiée avec une possibilité de faire un raffinement automatique (ou semi-automatique) du diagramme états-transitions en utilisant le contre-exemple. Par conséquent, avoir un outil unique au fonctionnement transparent qui permettra à l'utilisateur de faire de la modélisation et de la vérification.

Annexe A

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

[SO] inLib
Type: bool
Multiplity: One
Invariant: not(inLib) => (not(onLoan) and (not(booked) and ((booker = Undef) and not(tagged))))

[SO] onLoan
Type: bool
Multiplity: One
Invariant: onLoan => (not(tagged) and inLib)

[SO] booked
Type: bool
Multiplity: One
Invariant: booked => ((booker <> Undef) and not(tagged))

[SO] booker
Type: User
Multiplity: FromZeroToOne
Invariant: (booker = Undef) <=> not(booked)

[SO] tagged
Type: bool
Multiplity: One
Invariant: tagged => (not(onLoan) and not(booked))

[SO] final
Type: bool
Multiplity: One
Invariant: not(final)

3. Branch: Main, Events

```
[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)

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)		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

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

```
1
```

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

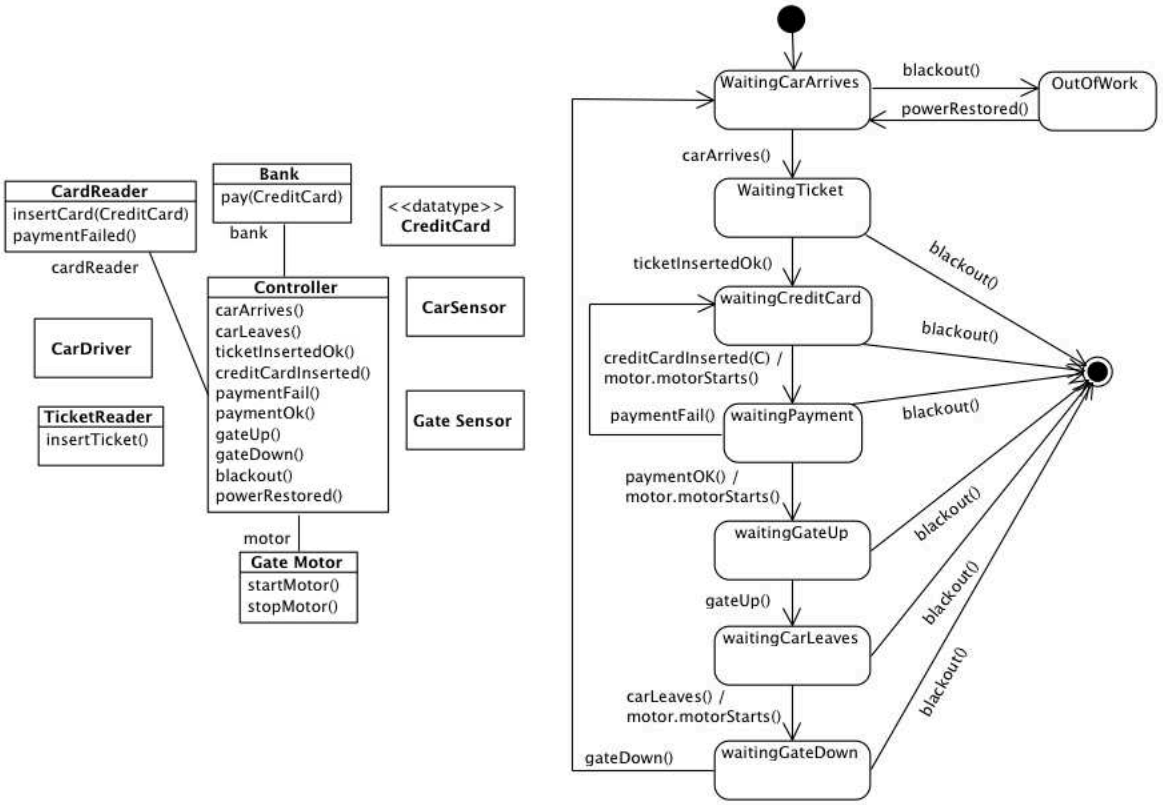
[ASS] booker
  + CL: Copy
  + CL: User

Questionnaires de l'expérience des métriques

Nom :_____ Prénom :_____ Formation :

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 ☐

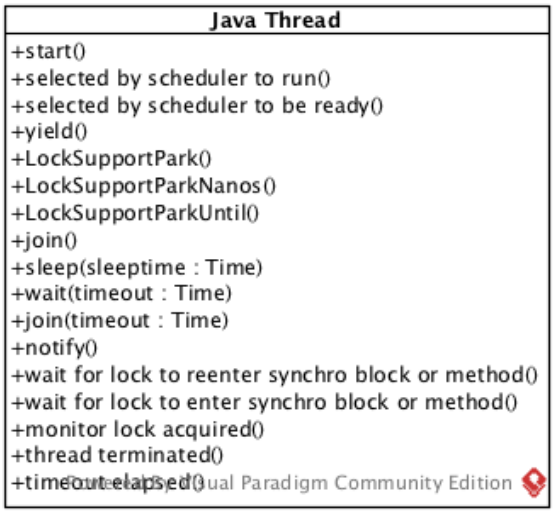
Le contrôleur tombera en panne ☐

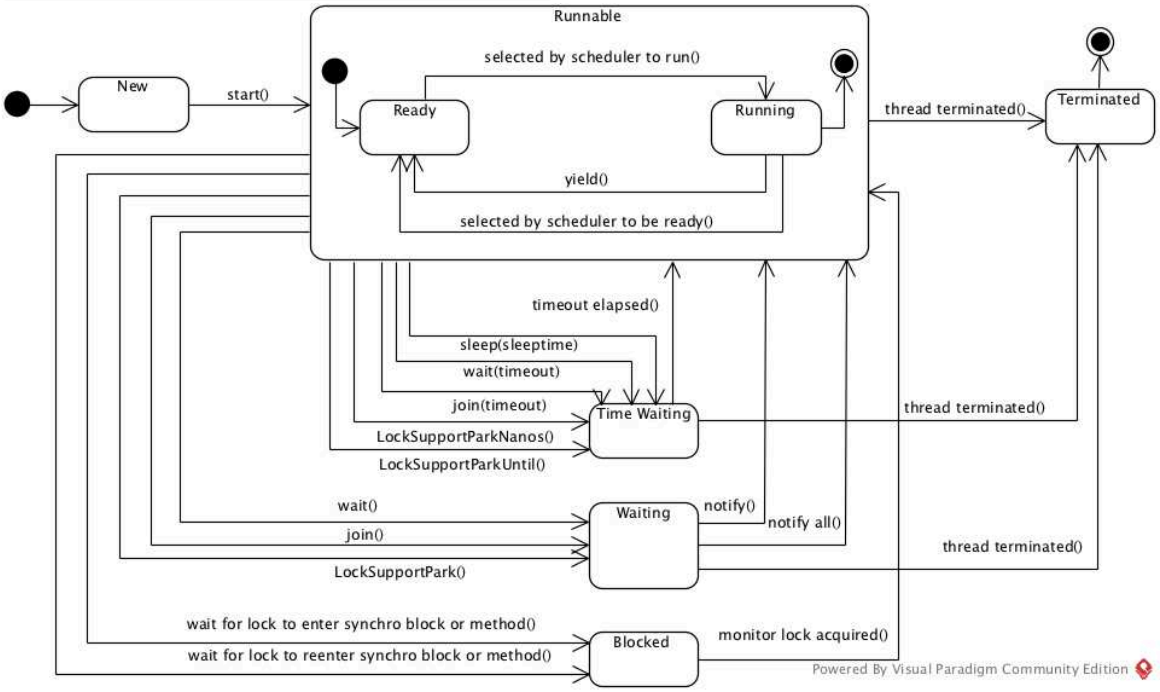
Une coupure de courant finira par se produire ☐

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 ☐

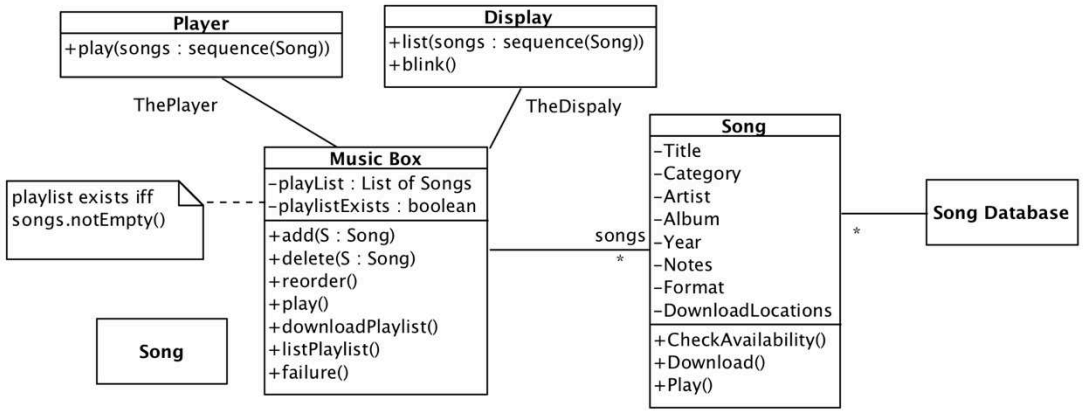
start() ; join(timeout) ; notify all(); selected by scheduler to run() OUI ☐ NON ☐

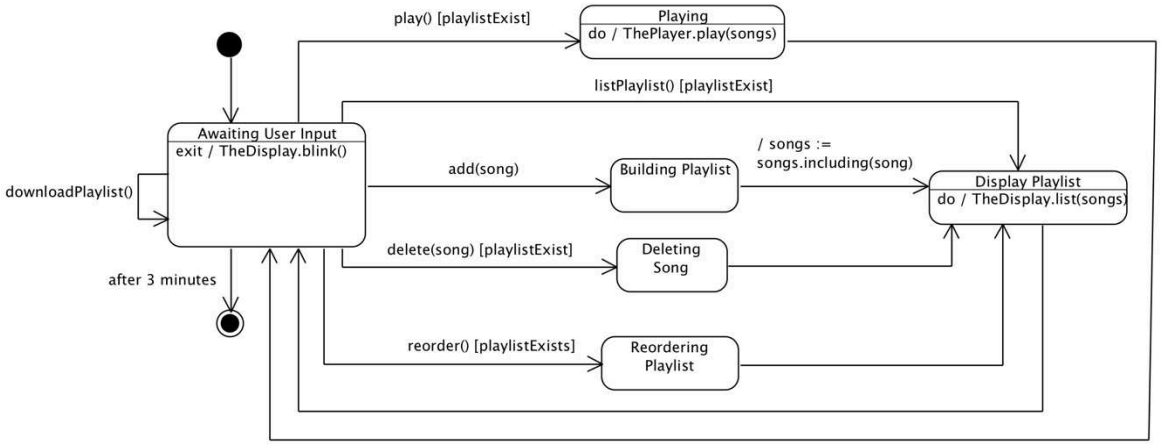
T4) Est-il vrai qu'un thread qui vient d'être créé et reçoit trois fois de suite l'événement start() sera dans l'état Ready? OUI ☐ NON ☐

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 :_____





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 ☐

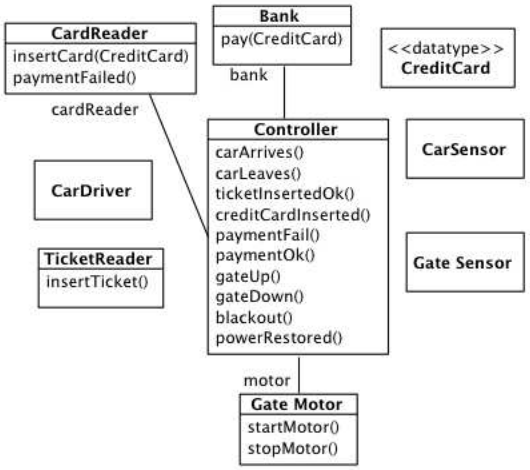
L'instance de Music Box réagit seulement au premier événement, et les 1999 suivants sont ignorés ☐

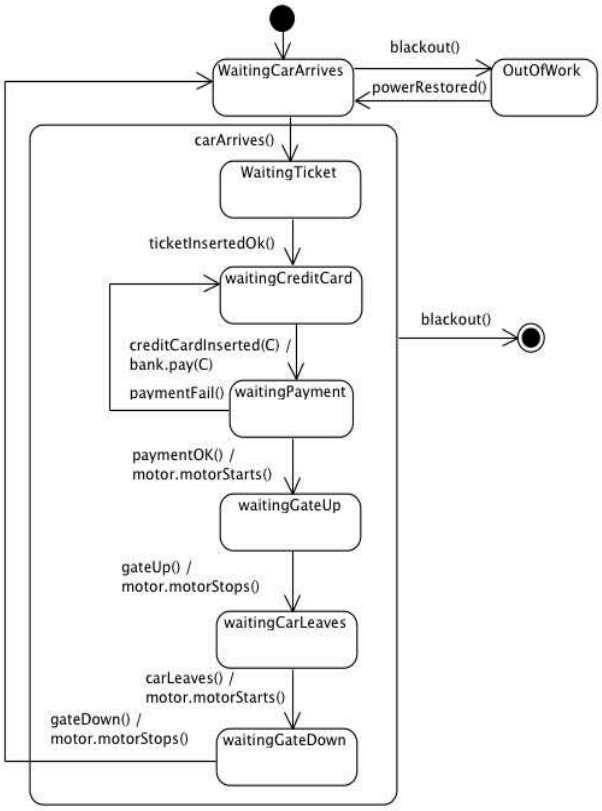
Heure fin :_____

Nom :_____ Prénom :_____ Formation :

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 ☐

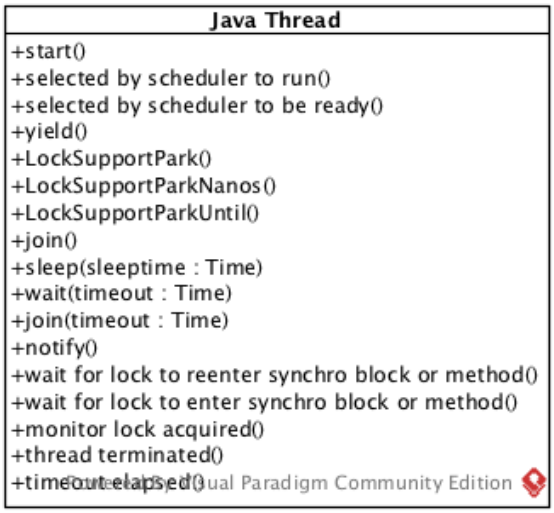
Le contrôleur tombera en panne ☐

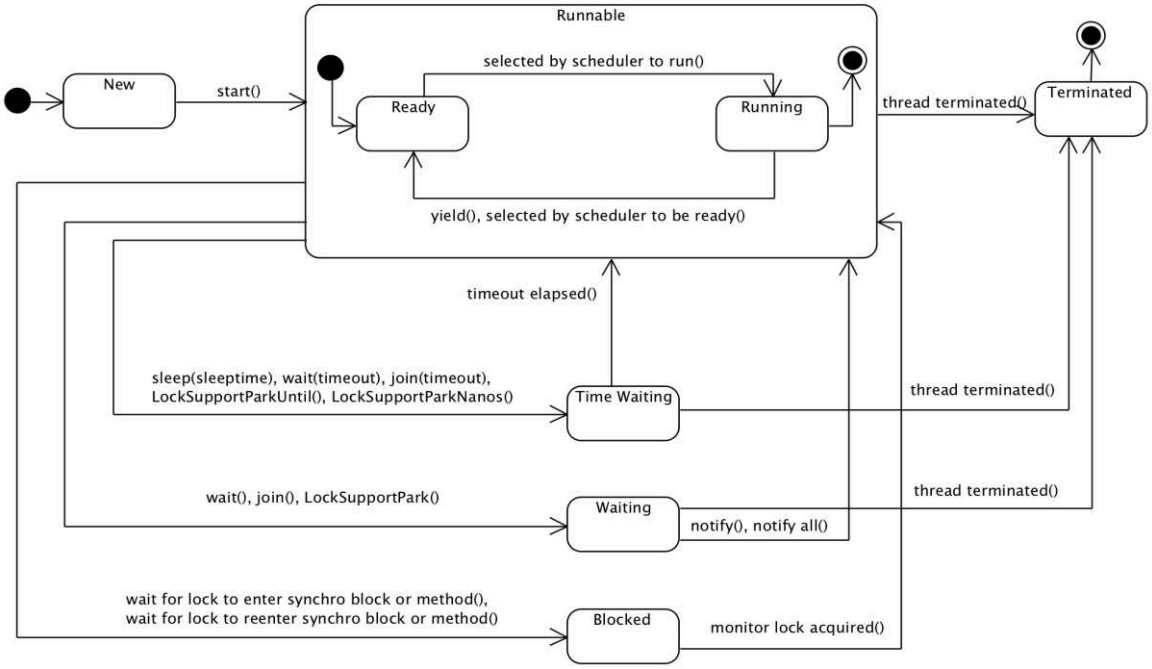
Une coupure de courant finira par se produire ☐

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 ☐

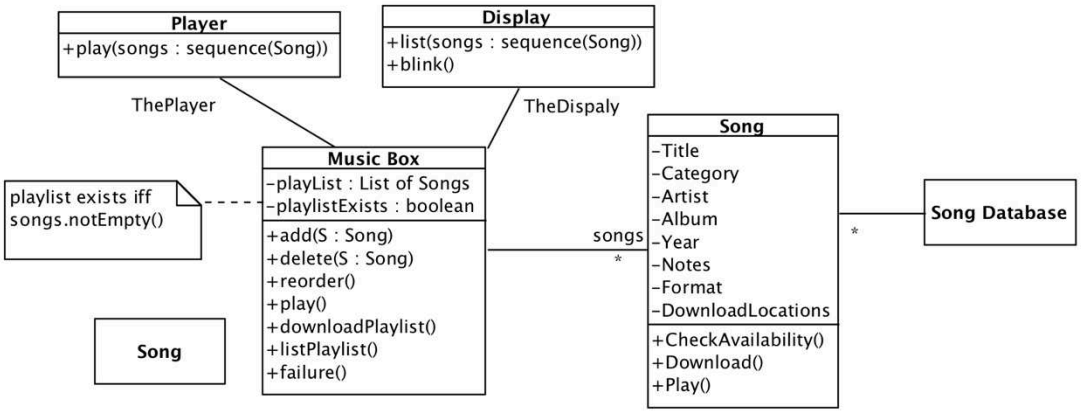
start() ; join(timeout) ; notify all(); selected by scheduler to run() OUI ☐ NON ☐

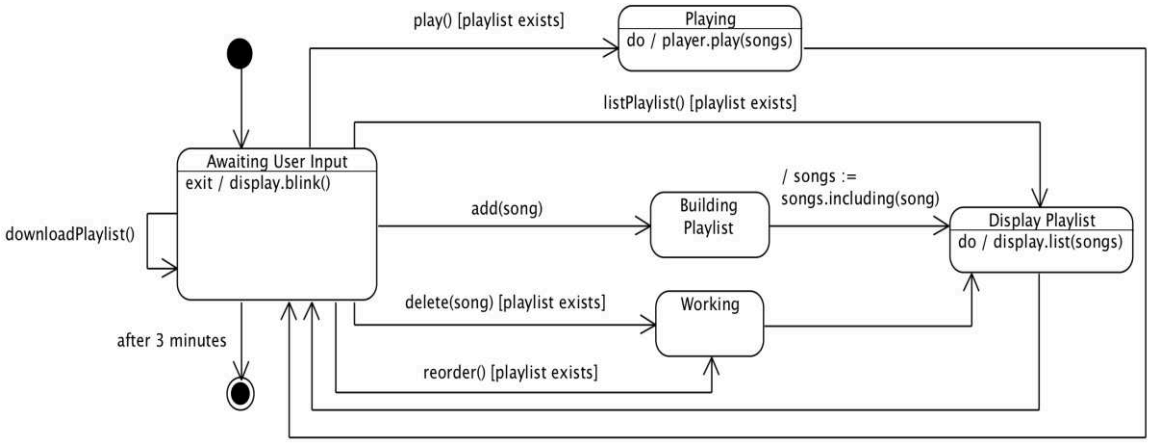
T4) Est-il vrai qu'un thread qui vient d'être créé et reçoit trois fois de suite l'événement start() sera dans l'état Ready? OUI ☐ NON ☐

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 :_____





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`).

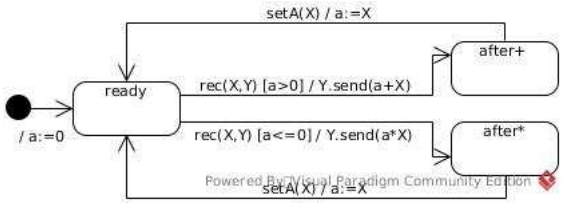


FIGURE A.1 – Exemple de l'article avec des états dupliqués

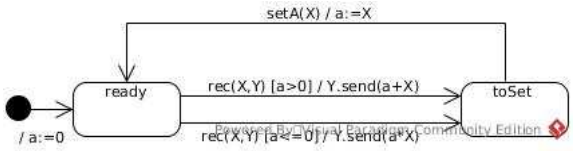


FIGURE A.2 – Exemple de l'article sans états dupliqués

Exemple de l'inscription Exemple utilisé pour l'étude du critère "de l'introduction de la hiérarchie et des états composites" dans un diagramme états-transitions. La Figure A.3 représente le diagramme états-transition de l'inscription à un séminaire avec la présence de plusieurs transitions étiquetées de l'événement `cancelled` sortant de différents états et allant vers le même état (ici l'état final). La Figure A.4 représente le même diagramme mais avec l'introduction d'un état composite (`Ongoingregistration`) et la suppression des transitions étiquetées avec l'événement `cancelled` et en les remplaçant par une transition (avec le même événement) allant de l'état composite `Ongoingregistration` vers l'état final.

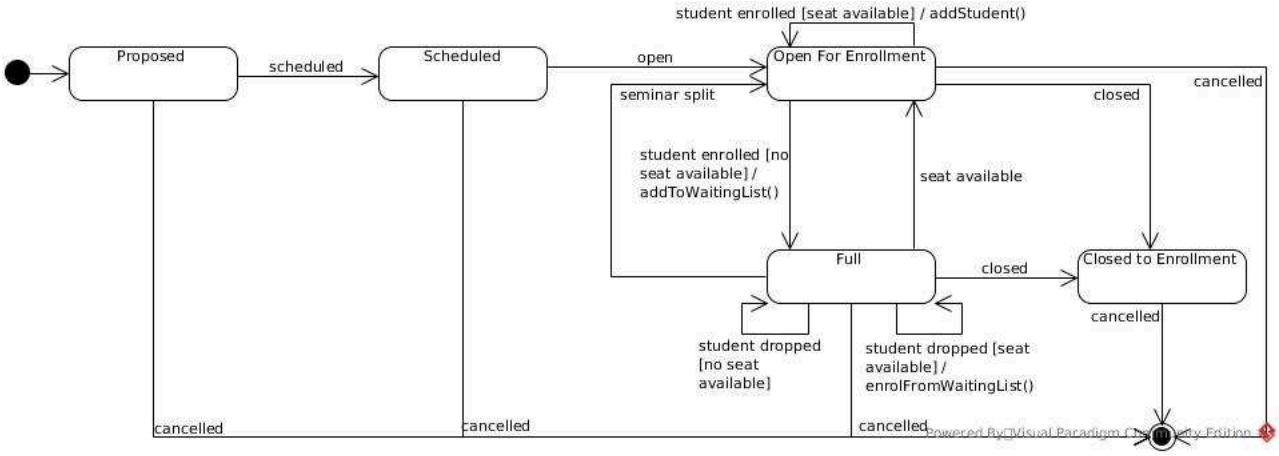


FIGURE A.3 – Exemple de l'inscription dans un séminaire sans hiérarchie

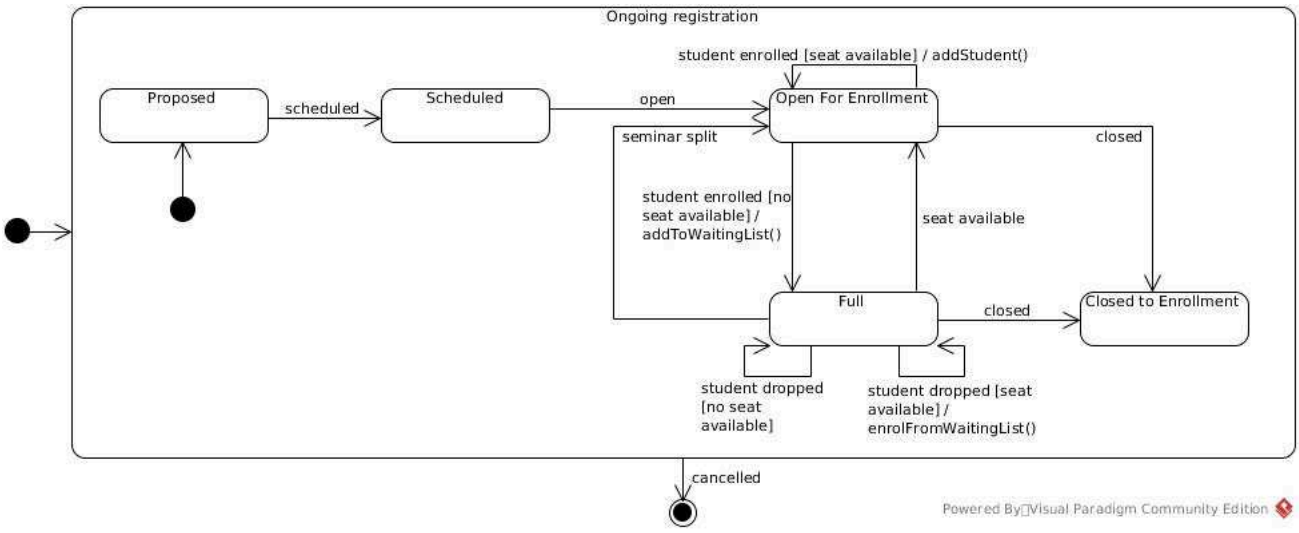


FIGURE A.4 – Exemple de l'inscription dans un séminaire avec hiérarchie

Exemple du cycle de vie d'un objet Exemple utilisé pour l'étude du critère "de l'utilisation événements multiples" dans un diagramme états-transitions. La Figure A.5 représente le diagramme états-transitions de la vie d'un objet avec la présence de plusieurs transitions sortant du même état, allant vers le même état mais avec des événements différents (*e.g.* de l'état `ExistsandReferenced` vers `ExistsandNotReferenced`). La Figure A.6 représente le même diagramme mais avec la factorisation des transitions sortant d'un seul état et allant vers le même état (*e.g.* la transition de l'état `DoesnotExistsandreferenced` vers l'état final).

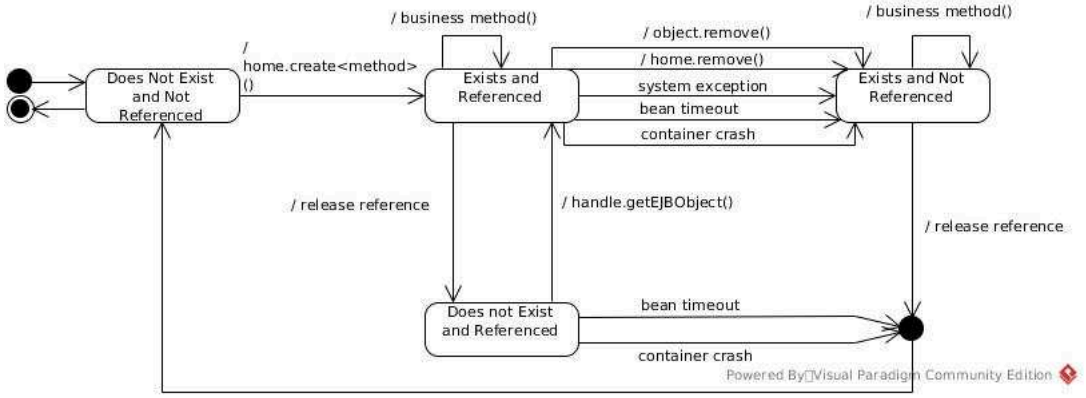


FIGURE A.5 – Exemple de la vie d'un objet avec des événements dans des transitions différentes

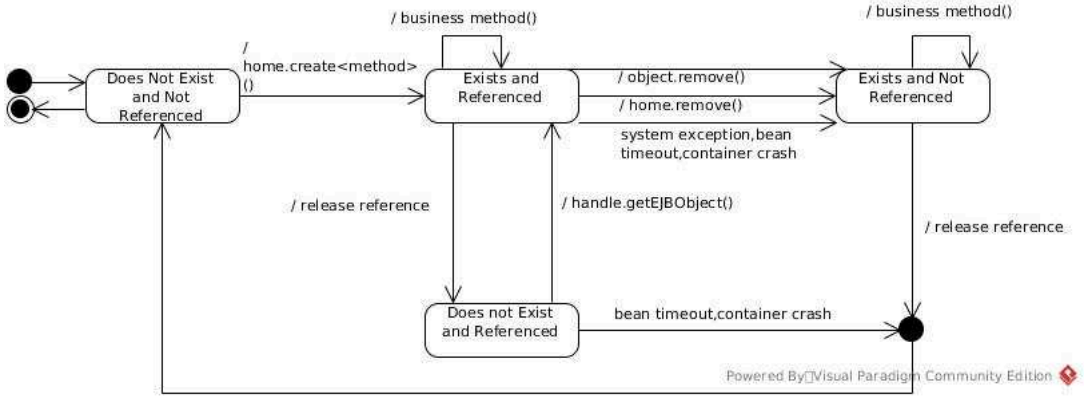


FIGURE A.6 – Exemple de la vie d'un objet avec des événements dans une seul transition