N° d'ordre : 184

ÉCOLE CENTRALE DE LILLE

T H È S E

pour l'obtention du

Doctorat de l'École Centrale de Lille

Spécialité : Automatique, Génie informatique, Traitement du Signal et Images

présentée par
Ahmed Mekki

Titre de la thèse :

Contribution à la Spécification et à la Vérification des Exigences Temporelles

Proposition d'une extension des SRS d'ERTMS niveau 2

Soutenue le 18 avril 2012 devant le jury d'examen

Président & Rapporteur	
M. Walter Schön	Professeur des Universités, UTC-Compiègne
Rapporteur	
M. Kamel Barkaoui	Professeur des Universités, CNAM-Paris
Examinateurs	
M. El-Miloudi El-Koursi	Directeur de Recherche, IFSTTAR-Villeneuve d'Ascq
M. David Gouyon	Maître de Conférences, CRAN-Nancy
M. Eric Rutten	Chargé de Recherche (HDR), INRIA-Rhône-Alpes
Encadrant	
M. Mohamed Ghazel	Chargé de Recherche, IFSTTAR-Villeneuve d'Ascq
Directeur	
M. Armand Toguyéni	Professeur des Universités, LAGIS-École Centrale de Lille

Thèse préparée au laboratoire ESTAS de l'Institut Français des Sciences et Technologies des Transports, de l'Aménagement et des Réseaux (IFSTTAR).
Laboratoire LAGIS de l'École Centrale de Lille

École Doctorale SPI 072 (Lille I, Lille III, Artois, ULCO, UVHC, EC Lille)
PRES Université Lille Nord-de-France

À ma grande-mère Bournia

Remerciements

Les travaux de thèse présentés dans ce manuscrit ont été réalisés à l'institut français des sciences et technologies des transports (IFSTTAR) à Villeneuve d'Ascq. Je voudrais saisir cette opportunité pour remercier IFSTTAR de m'avoir offert cette expérience magnifique. La recherche scientifique est un métier magique et c'est un bonheur de l'avoir commencé ici.

C'est avec toute ma gratitude que j'exprime ici mes plus vifs remerciements à Mohamed Ghazel, qui a bien voulu accepter d'encadrer ce travail de thèse. Sans sa disponibilité malgré sa charge de travail, sa patience, les conseils qu'il m'a prodigués tout au long de ces trois ans, ses mots d'encouragement, ce travail n'aurait pas pu, sans doute, être mené à son terme. Merci.

J'exprime ma profonde reconnaissance et remerciement à Armand Toguyéni pour avoir dirigé mes travaux de thèse, pour sa passion, ses conseils et pour m'avoir aidé à améliorer mon travail. Malgré ses responsabilités administratives qui s'ajoutent à sa charge du travail d'enseignement, d'encadrement et de recherche, il a toujours su trouver du temps pour discuter. Merci.

Je remercie chaleureusement Florent Peres, qui a suivi de près une partie de ce travail, sans qui cette partie n'existerait pas sous cette forme. Ses observations judicieuses et ses remarques ont été des plus déterminantes dans la conduite de ces travaux et dans la forme finale qu'ils ont pris.

Remerciements à Monsieur Kamel Barkaoui, professeur des universités au CNAM-Paris, et à Monsieur Walter Schön, professeur des universités à l'université de technologie de Compiègne, pour avoir accepté de rapporter ma thèse. Je remercie également tous les membres du jury pour avoir bien voulu participer à ma soutenance : Monsieur Eric Rutten, directeur de recherche à l'INRIA Rhône-Alpes,, Monsieur David Gouyon, maître de conférences au CRAN-Nancy, et Monsieur El-Miloudi El-Koursi, directeur de recherche à l'IFSTTAR-Villeneuve d'Ascq et directeur de l'unité ESTAS.

Mes remerciements vont également à tous les personnels du centre IFSTTAR-Villeneuve d'Ascq pour la gentillesse et la convivialité dont ils ont fait preuve et qui ont rendu mon séjour très agréable parmi eux.

Remerciements à tous les doctorants, post-doctorants, du IFSTTAR-Villeneuve d'Ascq. Merci donc à Nizar, Sana, Nedim, Sabrine, Jing, Khaled, Stephen et Nesrine, ainsi que à tous mes amis : Khaled M., Issam, Mahmoud D., Mahmoud B., Mamoudou K., Ahmed S. et Abdessalem. J'ai une pensée toute particulière pour mon confrère Youness à qui je souhaite de terminer le plus rapidement possible et dans les meilleures conditions ses travaux de thèse. Je lui souhaite bonne continuation tout au long du chemin qu'il a choisi d'emprunter après sa thèse.

Dans ces moments importants, je pense très fort à ma famille. Tout ça n'aurait jamais été possible sans le soutien inconditionnel de mes parents Saïda et Tijani, qui ont toujours cru en moi. Merci pour avoir fait de moi ce que je suis à présent. Merci à tous mes sœurs et mes frères. Remerciements pour m'avoir toujours soutenu dans mes entreprises et dans les moments difficiles. Une pensée toute particulière pour mon frère Habib pour son soutien inconditionnel durant mes études universitaires à Monastir, à Tunis et à Lille.

J'exprime ma profonde gratitude envers ma deuxième famille en France : ma tante Aïcha,

mon oncle Hassen, Atef, Hanan et Fayçal pour leur accueil, leurs encouragements et tous les bons moments que nous avons partagés ensemble. Merci infiniment à Sophie et Zobeir Lafhaj pour m'avoir accueilli à Lille.

Merci à toutes les personnes, qui m'ont toujours soutenu et encouragé au cours de la thèse, et que je n'ai pas citées ici et qui se reconnaîtront à travers ces quelques lignes.

Mention spéciale au printemps Arabe. Je ne peux pas passer sans remercier les jeunes arabes qui acceptent de mourir afin d'être les maîtres de leurs destins. De Sidi Bouzid en Tunisie à Benghazi en Libye, de la place de la Libération ("Tahrir") en Égypte à Deraa en Syrie, les peuples arabes, trop longtemps opprimés, se sont soulevés pour briser les chaines imposées par des dictatures néo-féodales et pour que triomphe la liberté, l'égalité et la justice.

Une pensée spéciale à celle qui n'a eu le temps de dire un dernier au revoir, ma grande-mère Bournia. Ce n'est qu'une fois qui ne sont plus là que l'on se rend compte à quel point certaines personnes qu'on aime tiennent une place si importante dans nos vies. C'est à son âme que je dédie ce travail, elle qui n'aura pu voir son achèvement.
Tu me manques.

Table des matières

Table des figures

Liste des tableaux

Glossaire

ADM Architecture Dirigée par les Modèles. 63–67, 70

AEF Automate à États Fini. 43

AFIS Association Française d'Ingénierie Système. 23

AT Automate Temporisé. 28, 29, 43, 92, 95, 96, 98–101, 105–107, 109, 110, 112, 114, 128, 131, 160

BNF Backus-Naur Form. 48

BTM Balise Transmission Module. 140, 142

CIM Computational Independent Models. 68

CLKS Clocked Labelled Kripke Structure. 88

CTL Computation Tree Logic. 39, 42

DFL Durée de Fermeture Longue. 146–148, 151, 153, 154

DMI Driver Machine Interface. 141, 142

DOC Durée d'Ouverture Courte. 146–148, 151, 153, 154

DoD Department of Defense. 21

ERTMS European Rail Traffic Management System. 30, 31, 134, 137, 139–145, 147, 152, 155

ETCS European Train Control System. 139, 141

ETML European Traffic Management Layer. 140

EVC European Vital Computer. 141, 142, 149, 150

GIL Graphical Interval Logic. 39

GSM-R GSM for Railways. 140–144, 152

IE Ingénierie des Exigences. 23

IFSTTAR Institut Français des Sciences et Technologies des Transports. 5

INESS Interlocking Safety System. 140

IS Ingénierie Système. 19, 21, 23

LTL Linear Temporal Logic. 39, 42

LTM Loop Transmission Module. 142

LTS Labelled Transition System. 71

MDA Model driven Architecture. 63, 64

MMI Man Machine Interface. 141, 142

MOF Meta-Object Facility. 67

MTL Metric Temporal Logic. 40

NASA National Aeronautics and Space Administration. 21

PIM Platform Specific Models. 68

PLTL Propositional Linear Temporal Logic. 42, 126

PN Passage à Niveau. 30, 139, 142, 144–155, 161

PSM Platform Independent Models. 68

QRE Quantified Regular Expressions. 39

RBC Radio Block Center. 141–144, 148, 150

RdP Réseaux de Petri. 42

RTC Run To Completion. 88

RTGIL Real-Time Graphical Interval Logic. 40

RTM Radio Transmission Module. 143

SADT Structured Analysis and Design Technique. 41

SdF Sûreté de Fonctionnement. 19

SM State Machine. 28, 29, 31, 41, 77, 79–83, 85–90, 95–101, 104–112, 114, 133, 150, 151, 160

SRS System Requirement Specifications. 30, 134, 139, 142, 144, 152, 155

STE Système de Transitions Étiqueté. 71–73

STM Specific Transmission Modules. 143

STT Système de Transitions Temporisé. 72–74

TCTL Timed Computation Tree Logic. 40, 42

TIU Train Interface Unit. 141, 143

TTS Timed Transition Systems. 72, 108

UML Unified Modeling Language. 28, 29, 31, 41, 77, 79, 81, 82, 88, 89, 95, 96, 98–100, 105, 132, 133

V&V La Vérification et la Validation. 22, 25, 31, 117, 119, 120, 122, 134

Chapitre 1

Introduction : problématique et contributions

Résumé : *Dans ce premier chapitre, nous introduisons la problématique générale de la thèse et nous donnons un aperçu global des contributions proposées. Tout d'abord, nous commençons par détailler les mécanismes ainsi que les enjeux de l'ingénierie système (IS) pour les systèmes complexes à contraintes de temps. Nous nous basons sur le constat que la maîtrise de la sûreté de fonctionnement (SdF) est un concept clé et prépondérant dans le cycle de vie de ces systèmes. Afin de garantir la SdF, les mécanismes utilisés doivent faire recours à des notations rigoureuses ainsi qu'à des techniques dont l'efficacité est incontestable.*

Néanmoins, la phase de mise en œuvre de ces mécanismes ainsi que la phase de leur intégration présentent des difficultés considérables notamment pour les non-experts. En effet, la complexité de ces mécanismes souvent formels empêche leur déploiement direct vu que la manipulation de tels mécanismes nécessite une expertise bien spécifique. Cette problématique de mise en œuvre est au cœur de nos travaux de recherche qui visent à proposer et implémenter des techniques d'assistance au profit de l'utilisateur, et en particulier pendant la phase de spécification et la phase de vérification et validation.

Sommaire

1.1 Contexte et problématique

1.1.1 Ingénierie système

De nos jours plusieurs systèmes sont des implémentations à grande échelle avec une complexité considérable. Face à cette complexité croissante, la disposition de formalismes adéquats pour la description de différent points de vue sur le système, ainsi que de méthodes outillées, est devenue une nécessité afin de garantir la maîtrise de ces implémentations tout au long de leurs cycles de vie. Cette nécessité s'est traduite en un domaine de recherche intitulé l'Ingénierie Système (IS) dont les premiers travaux datent des années 60. L'IS a été explorée depuis par des organismes majeurs tels que la NASA (National Aeronautics and Space Administration) ou le DoD (Department of Defense) américain. Leur objectif était de définir des méthodologies/approches globales de conception, de production et d'exploitation adaptées à la complexité et à la criticité toujours croissante de leurs programmes spatiaux et militaires. Cette initiative a été ensuite appliquée dans des secteurs industriels très variés : énergie nucléaire, transport (ferroviaire, automobile, aéronautique), protocoles de télécommunications, etc [Seidner 2009, Dobre 2010].

Définition 1.1 *[Seidner 2009] "L'Ingénierie Système est un processus :*

- *collaboratif et interdisciplinaire de résolution de problèmes ;*
- *s'appuyant sur les connaissances, méthodes et techniques issues des sciences et de l'expérience ;*
- *mis en œuvre pour définir un système qui satisfasse un besoin identifié et qui soit acceptable par l'environnement ;*
- *défini sur tous les aspects du problème, dans toutes les phases de développement et de la vie du système."*

Ainsi, l'IS peut être également définie comme étant un processus dont l'objectif est de cadrer la conception et la mise en œuvre des systèmes. Nombreuses sont les propositions qui visent à caractériser des approches d'IS (la norme ANSI/EIA-632 [1] [EIA 1998], la norme IEEE 1220 (figure 1.1) [2] [IEEE 2005], la norme ISO 15288 [3] [ISO 2008], le standard MIL-STD-499B [4] [Force 1994]) et qui ont été par la suite normalisées. L'IS prend d'autant plus d'intérêt quand le système en question présente un niveau de criticité. Or, le développement de tels systèmes repose principalement sur deux étapes, à savoir l'étape de spécification et l'étape de vérification/validation.

Définition 1.2 *[Sommerville 1997] La spécification est généralement une activité de description/caractérisation du produit (système et/ou logiciel) ainsi qu'une extraction et expression (formalisation) des exigences (ou propriétés) qu'on lui a fixées. Cette étape prend en entrée le cahier des charges pour générer un ensemble de documents et/ou modèles qui expriment et représentent ces propriétés. Ces modèles doivent décrire d'une façon claire, précise et non ambiguë les exigences/propriétés qu'on cherche à vérifier sur le produit à développer. Les objectifs de l'étape de spécification peuvent être résumés comme suit :*

- *fournir, détailler et clarifier certains aspects (aspects temporels par exemple) du produit à concevoir,*

1. Processes for Engineering a System, 1999
2. Standard for application and Management of the Systems Engineering Process, 1995 et révisé en 2005
3. Systems Engineering - System Life-Cycle Processes, 2003
4. Systems Engineering, 1969 mis à jour en 1994

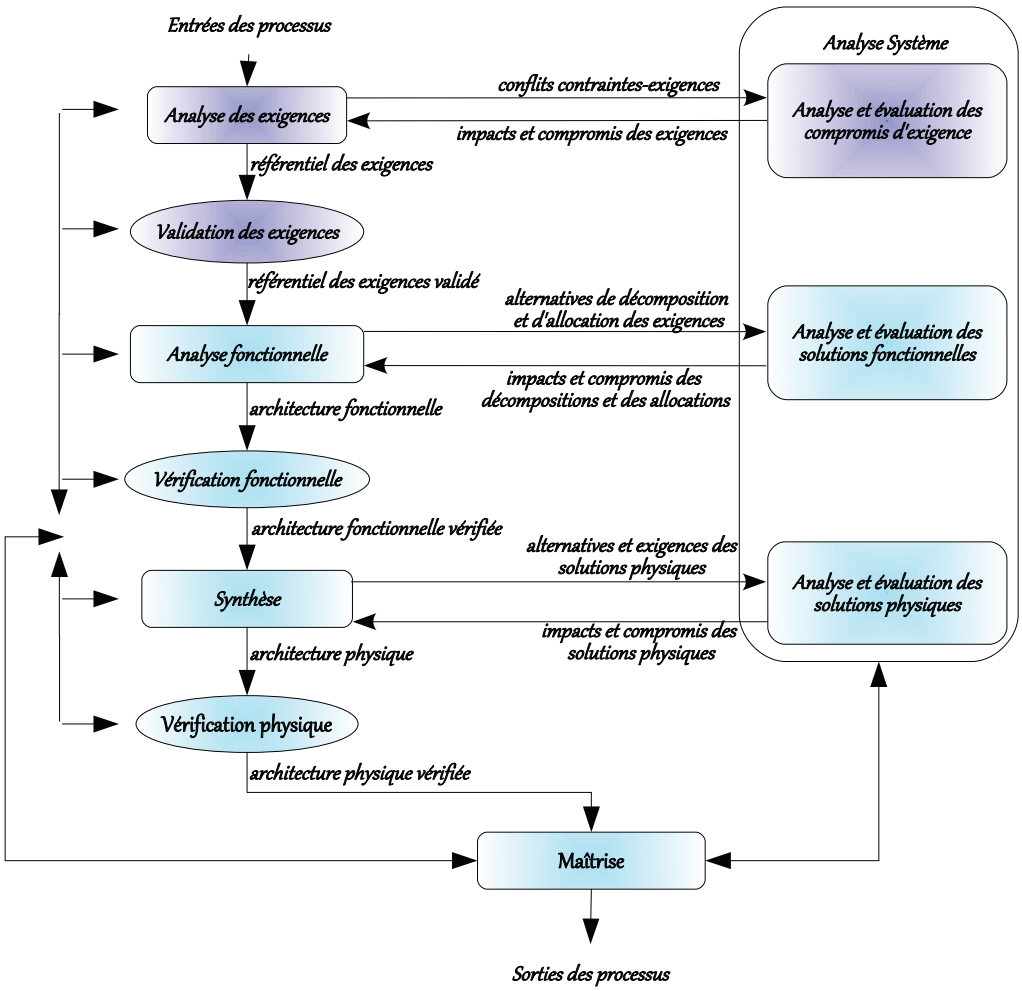

FIGURE 1.1: Processus d'Ingénierie Système [Seidner 2009, IEEE 2005]

- *identifier les sources de défaillances du produit,*
- *exprimer les exigences, les propriétés et les contraintes que le système doit satisfaire.*

Définition 1.3 *La vérification et la validation (V&V) visent à garantir que le produit à concevoir répond aux besoins des utilisateurs exprimés sous forme d'exigences [Boehm 1988].*

Étant donné que les conséquences de certaines défaillances dans les systèmes critiques [Schön 2008] peuvent être importantes à la fois sur les plans humain et matériel, tout doit être mis en œuvre pour garantir un haut niveau de sécurité. Cela implique la nécessité de disposer de méthodes et outils de spécification et de vérification de tels systèmes. Ces différents moyens servent de support pour valider les différentes contraintes fonctionnelles et dysfonctionnelles de ces systèmes à la fois *critiques* et *complexes*.

Définition 1.4 *La complexité, selon Morin [Morin 1994], "a toujours affaire avec le hasard". Elle est fortement liée à l'imprévisibilité et l'imprédictibilité. En d'autres termes, "on ne peut pas comprendre l'organisation d'un système complexe dans tous ses détails, ni prévoir ses comportements" [Ott 2009]. "Pour un observateur, un système est complexe parce qu'il tient pour certain l'imprévisibilité potentielle des comportements" [Moigne 1999]. En plus, selon Ott [Ott 2009], "la complexité contient toujours une part d'incertitude, que celle-ci soit liée aux limites de notre*

entendement ou à son inscription dans le phénomène. Cela implique les notions d'aléa et de hasard. En effet, lorsque l'on est face à la complexité, on est également face à l'incertitude, à l'aléatoire, au désordre, au doute. Certes, la complexité comprend des quantités d'unités et d'interactions qui renforcent l'imprévisibilité. Mais elle comprend également des incertitudes, des indéterminations et des phénomènes aléatoires".

Une classe de ces systèmes est dite "critique" quand les défaillances qui peuvent s'y produire peuvent avoir des dégâts matériels et/ou humains très importants [Schön 2008, Belmonte 2010]. Dans cette classe on trouve des systèmes issus de différents domaines comme : le domaine du transport (aérien, ferroviaire, routier), le domaine financier (banques, bourses), le domaine médical ou celui de la production énergétique. Or, certains délais d'interaction ainsi que le temps de réponse de certaines entités composant ces systèmes peuvent être critiques. Par conséquent, des dépassements de délai, par exemple, peuvent entraîner des pannes significatives sur le fonctionnement du système. De manière générale, les contraintes (exigences) temporelles qui reflètent des relations temporelles qui doivent caractériser le comportement du système, peuvent exprimer des propriétés de sûreté de fonctionnement comme, par exemple, la sécurité et la disponibilité.

Dans le cadre de nos travaux nous nous intéressons aux exigences temporelles. En particulier, la spécification et la vérification de ce type d'exigences est au cœur des travaux réalisés dans le cadre de cette thèse. Il est indispensable de rappeler que ces deux étapes sont indispensable durant un processus d'IS. Néanmoins, la réussite d'un processus d'ingénierie système repose principalement sur la réussite du processus d'ingénierie des exigences [Konaté 2009]. Celui-ci est abordée dans la section suivante.

1.1.2 Ingénierie des exigences

Étant donnée l'absence d'une définition standard pour l'exigence, plusieurs définitions ont été proposées. Ici, nous donnons deux définitions qu'on trouve dans la littérature. Une première est donnée par l'association française d'ingénierie système (AFIS) [5] pour laquelle une exigence est définie comme suit : *une exigence prescrit une propriété dont l'obtention est jugée nécessaire. Son énoncé peut être une fonction, une aptitude, une caractéristique ou une limitation à laquelle doit satisfaire un système, un produit ou un processus.* La deuxième définition est donnée par IEEE : *une exigence est une condition ou une capacité qui doit être satisfaite par un système ou un composant d'un système pour satisfaire un contrat, une norme, une spécification, ou autres documents formellement imposés* [IEEE 1990].

L'ingénierie système vise à satisfaire aux attentes et aux besoins des clients (maîtres d'ouvrage). Ces attentes et besoins nécessitent d'être formalisés pour pouvoir être exploités par le processus d'IS. L'ingénierie des exigences vise à proposer des moyens pour améliorer la rédaction et l'expression de contenu de ces attentes et de ces contraintes, et pour vérifier leur conformité [Lepors 2010].

Selon AFIS, *l'Ingénierie des Exigences (IE) consiste, au travers de méthodes, règles et processus, à établir et maintenir un référentiel unique qui s'affine et se complète au cours du développement et qui est maintenu tout au long de la vie du système.* Les éléments de base de cette discipline sont les besoins et les exigences. Contrairement aux besoins qui représentent une vision des perceptions et des attentes de l'utilisateur final, les exigences représentent la vision du concepteur des ces besoins (figure 1.2).

5. http://www.afis.fr

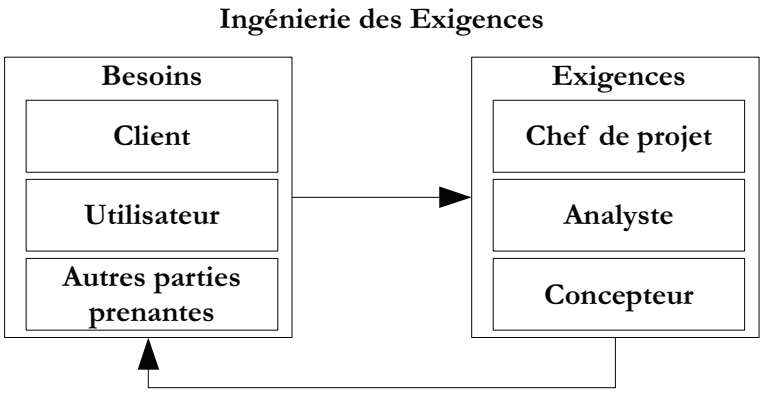

FIGURE 1.2: Besoins et Exigences [Konaté 2009]

Concrètement, un processus d'ingénierie des exigences est décomposé en quatre étapes [Kotonya 1998] : élicitation, analyse, spécification et vérification. Dans le cadre de nos travaux nous nous focalisons sur les deux dernières étapes à savoir l'étape de spécification et l'étape de vérification.

1.1.3 Problématique

Nos travaux visent comme application les systèmes complexes à contraintes du temps. Un système complexe à contrainte de temps est un système complexe pour lequel le facteur temps est un paramètre essentiel du fonctionnement. Il en résulte qu'il est indispensable pour l'expression de la grande partie des ses exigences de sécurité. Le cycle de développement de tels systèmes repose principalement sur deux éléments clés : le premier est l'expression d'une manière claire et précise de leurs exigences alors que le deuxième est la conformité et la validité de leurs exigences. Plusieurs méthodes ont été proposées afin de mener ces deux étapes. Elles sont abordées ci-après.

1.1.3.1 Méthodes de spécification

Plusieurs méthodes et techniques ont été proposées comme support pour mener à bien l'étape de spécification des exigences ainsi que l'étape de vérification et validation. Dans la littérature, on trouve principalement deux classes de méthodes de spécification : les méthodes qui permettent de générer des spécifications semi-formelles et celles qui produisent des spécifications formelles. Ces deux classes de méthodes de spécification reposent sur des langages et notations de spécification.

Une méthode de spécification, qui génère des spécifications semi-formelles, repose généralement sur un langage semi-formel graphique et/ou textuel. Les langages utilisés possèdent généralement une syntaxe bien définie. Cependant, la sémantique de ces langages souffre souvent de certaines ambiguïtés d'interprétation qui rendent leur exploitation directe impossible. En effet, la syntaxe et la notation des langages semi-formels se situent souvent à mi-chemin entre les langages naturels et les langages formels. Cependant, ce type de modélisation présente une limitation majeure qui est l'absence d'une sémantique claire et précise. Par conséquent, la vérification des modèles exprimés sur la base de ces langages n'est pas garantie.

Une notation est dite formelle quand elle est fondée sur des bases mathématiques et par conséquent elle possède une sémantique claire et précise (formelle) [Gervais 2004]. Les spécifications formelles se distinguent des autres spécifications par (i) leur capacité/pouvoir d'ex-

pression, (ii) la précision des concepts utilisés, et par conséquent, (iii) l'exactitude des spécifications obtenues. En effet, les méthodes reposant sur des spécifications formelles permettent d'identifier, de repérer et de réduire les impacts des erreurs à travers une spécification complète [Hall 1990]. Ainsi, l'utilisation et la mise en œuvre de ces méthodes, dans le processus de conception, permet (a) de détecter les erreurs assez tôt durant le processus de développement et (b) de réduire considérablement l'étape de vérification (automatisation à l'aide d'outils). Cela garantit une réduction significative des coûts et des délais [Bowen 1995]. Selon [Bowen 1995], ces méthodes peuvent aider dans la compréhension d'un produit à concevoir et à implémenter (par l'équipe de développement). En effet, à travers leur syntaxe formelle et leur sémantique claire, les spécifications formelles permettent à l'équipe de développement de disposer d'une description précise du système. Cela est très intéressant vu que la disposition de telle description facilite les étapes suivantes, à savoir la conception, l'implémentation, la vérification et la validation. Par conséquent, les méthodes reposant sur des spécifications formelles ont gagné du terrain dans l'industrie, notamment dans le secteur des systèmes critiques. Néanmoins, malgré leurs bases théoriques solides, la manipulation de langages formels reste une tâche difficile pour les non-experts et une source importante d'erreurs, étant donné le caractère abstrait inhérent à ce type de langages. Le premier objectif de notre travail est de proposer des mécanismes pour cadrer l'équipe de développement durant la phase de spécification tout en cachant, autant que possible, à l'équipe de développement certains aspects formels liés à cette étape.

1.1.3.2 Techniques de vérification

Nombreuses sont les techniques de vérification/validation (V&V) qu'on trouve dans la littérature et qui ont prouvé leur efficacité. Cependant, il y a plusieurs limitations sous-jacentes à la mise en œuvre des ces techniques. D'un côté, malgré la simplicité et l'intuitivité de la mise en application de certaines techniques comme le *test* et/ou la *simulation*, le résultat obtenu ne garantit très souvent pas l'absence d'erreurs ou de dysfonctionnements, étant donné que ces techniques ne sont pas exhaustives. D'un autre côté, l'utilisation de techniques exhaustives comme la *preuve* ou le *model-checking* nécessite des compétences en logique ainsi qu'un savoir-faire spécifique pour pouvoir extraire les exigences ainsi que pour modéliser le système à vérifier. Le second objectif de ce travail est de proposer des outils pour assister l'équipe de V&V pendant la phase de vérification, et en particulier en rendant, autant que possible, transparents certaines étapes du processus de vérification.

1.1.3.3 En pratique

Dans la pratique, le client (maître d'ouvrage) fournit une description souvent textuelle du produit demandé. Cette description exprime les fonctionnalités du produit ainsi que les besoins auxquels il doit répondre. Le document fourni est souvent appelé "cahier des charges" et est généralement exprimé en langage naturel. Cependant, étant donnée la nature de ce document, la phase d'extraction et d'identification des besoins et des propriétés/exigences du futur produit reste très souvent ardue. Cette étape d'extraction est une source importante d'erreurs étant donné le manque de précision du langage naturel. Plus tard dans le cycle de développement du produit, une étape de modélisation du comportement des différentes entités est également indispensable. En pratique, avec l'introduction de l'ingénierie dirigée par les modèles, nous faisons recours de plus en plus aux modèles lors de la conception et la modélisation du système. La spécification ainsi que le modèle développé en réponse à la spécification sont souvent utilisés comme entrées à la phase de vérification. Cette dernière étape consiste à vérifier que le modèle développé en entrée vérifie bien les propriétés qu'on lui a fixées.

Remarque 1.1 *En résumé, afin d'être exploitable (automatiquement) par les étapes suivantes (conception, développement, vérification et validation), les exigences doivent être exprimées d'une manière précise et non ambiguë, c'est-à-dire d'une manière formelle. En outre, étant données les difficultés liées à l'utilisation des notations formelles, la mise à disposition de moyens pour assister et guider l'utilisateur lors de la phase de spécification des exigences s'avère très utile.*

1.2 Contributions de la thèse

Dans cette section, un aperçu des principales contributions de la thèse est donné. La figure 1.3 présente de manière sommaire les volets de notre contribution.

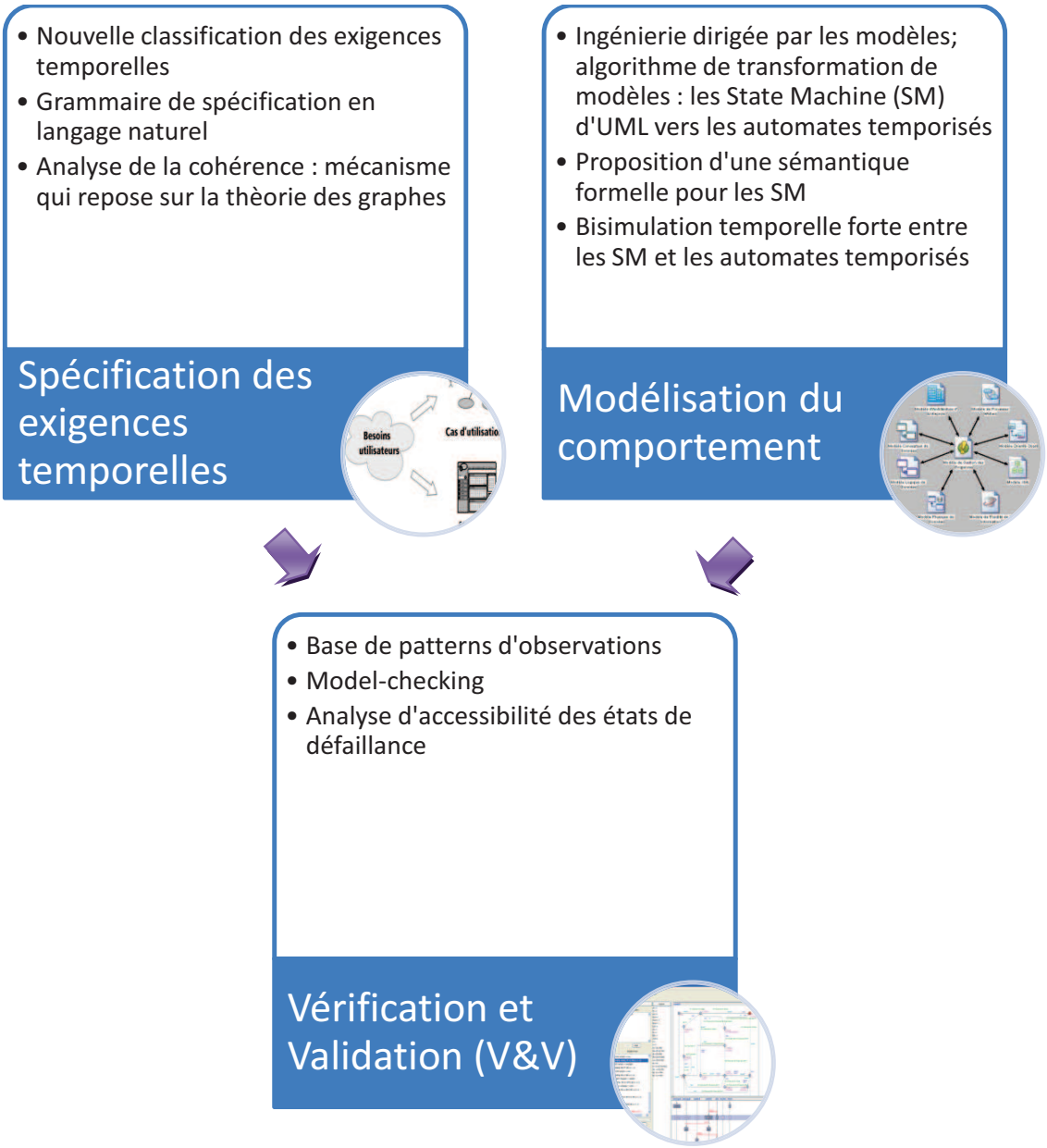

FIGURE 1.3: Les 3 volets de nos contributions

1.2.1 Étape de spécification des exigences temporelles

1.2.1.1 Une nouvelle classification des exigences temporelles

Nous avons commencé par identifier les différents types d'exigences temporelles les plus fréquemment rencontrées dans les spécifications. À la fin de cette étape nous avons défini une nouvelle classification des exigences temporelles qui comprend ces différents types d'exigences [Mekki 2009b, Mekki 2011a]. Le premier avantage de cette nouvelle classification réside dans la grande diversité des exigences qu'elle couvre. En particulier, notre classification comprend à la fois des exigences qualitatives, et des exigences quantitatives. Ensuite, notre classification prend en compte à la fois les exigences portant sur des événements et celles portant sur des états, et ce, de manière homogène.

1.2.1.2 Grammaire de spécification

Afin d'assister et guider l'utilisateur lors de la phase d'expression des exigences, nous avons développé une nouvelle grammaire [Mekki 2010b, Mekki 2011a]. Le langage généré à partir de cette grammaire est fini et chaque phrase générée identifie (et correspond à) une classe unique dans notre classification des exigences (et vice versa). Cette grammaire sert de socle à la phase de spécification des exigences temporelles. Concrètement, les propriétés temporelles que le système doit satisfaire sont exprimées sous forme d'assertions produites sur la base de notre grammaire qui propose des motifs pré-établis écrits en langage naturel. Les assertions ainsi composées sont à la fois simples et précises. Le processus d'identification des exigences temporelles produites est par conséquent rendu automatique à partir des assertions produites. Contrairement aux approches classiques, la spécification que nous proposons ne repose pas sur un cahier des charges existant. Concrètement, nous visons à assister et cadrer le maître d'ouvrage lors de la création de ce dernier.

1.2.1.3 Analyse de la cohérence

Les exigences générées sur la base de notre grammaire sont syntaxiquement précises et correctes quand elles sont prises individuellement. Cependant, cela ne garantit en aucun cas la cohérence de l'ensemble des exigences exprimées. Ainsi, une étape de vérification de la cohérence des exigences demeure nécessaire. À ce sujet, nous avons développé un certain nombre de mécanismes capables de détecter différents types d'incohérences entre les exigences [Ghazel 2010]. Cette étape est importante puisqu'en cas de présence de contradictions entre les exigences d'un système, aucune implémentation ne pourra satisfaire les exigences données. Par conséquent, cette vérification permet un gain important puisqu'elle permet d'identifier (assez tôt) les erreurs de spécification avant de concevoir le système et exécuter le processus de vérification sur le modèle du système.

Pour cela, nous avons développé un algorithme basé sur une représentation des exigences exprimées sous forme d'un graphe orienté. La vérification de la cohérence se fait donc à travers l'exploration de ce graphe. Dans celui-ci, les nœuds représentent les entités (les événements) et les arcs orientés -et éventuellement pondérés- représentent les relations entre les nœuds. Différentes pondérations ont été utilisées sur les arcs pour caractériser les relations qui lient les entités. Une pondération peut correspondre par exemple à une quantité de temps. Nous avons également identifié certains sources d'incohérence qui sont liées à l'incohérence d'ordre, de quantification et d'apparition entre les exigences. L'étape d'expression des exigences ainsi que l'activité de vérification de leur cohérence sont supportées par un outil logiciel dont les

fonctionnalités seront exposées au fur et à mesure du manuscrit. La figure 1.4 présente les éléments de ces contributions.

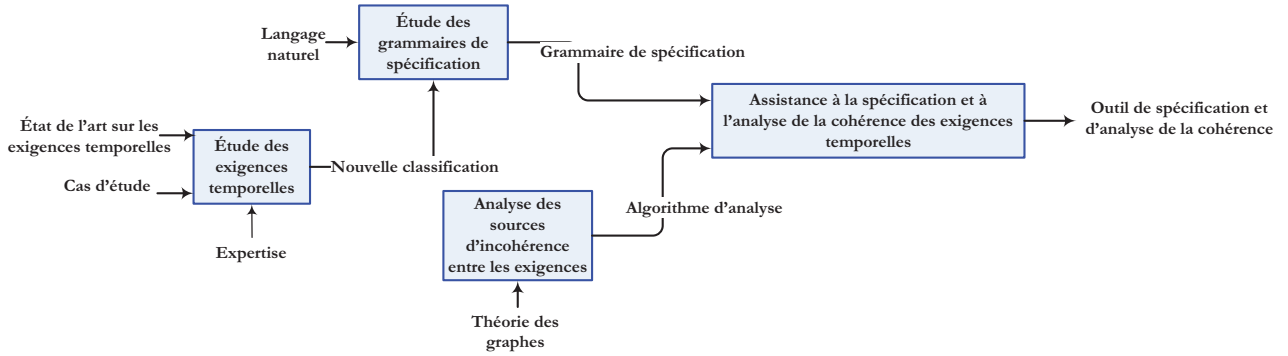

FIGURE 1.4: Contributions pour l'étape de spécification

1.2.2 Modélisation du comportement : transformation de modèles

Avec l'analyse de la cohérence, nous avons fini l'étape de spécification des exigences temporelles. L'étape de modélisation du comportement du système à vérifier est l'une des principales contributions de nos travaux. Elle vise à proposer une méthode pour assister la phase de modélisation du système à vérifier tout en tirant profit des avantages des modèles semi-formels (graphiques, intuitifs, simple à manipuler, flexibles) et des avantages des notations formelles (rigueur, précision, simulables, analysables). Précisons ici que l'étape de spécification des exigences vise à produire des représentations claires et précises des propriétés qu'on cherche à vérifier tandis que que l'étape de modélisation comportementale vise à produire un modèle du système à vérifier. Ce modèle est utilisé par la suite dans l'étape de vérification qui repose sur la technique de *model-checking*.

Dans notre étude, nous avons défini une transformation qui prend en entrée des modèles State Machine (SM) (états-transitions) d'UML avec des annotations temporelles qui génère un ensemble des modèles automate temporisé (AT). Cette transformation [Mekki 2010c] jouera un rôle non négligeable dans l'approche globale que nous proposons pour la vérification des exigences temporelles. En effet, étant données l'intuitivité et la flexibilité des SM d'UML, nous donnons la possibilité au concepteur de partir d'un modèle comportemental en SM avec des annotations temporelles. Nous proposons ici une transformation de ces modèles SM en modèles automates temporisés sur lesquels la vérification des exigences est déroulée. Autrement, l'idée est de faire profiter l'utilisateur de la flexibilité des SMs qu'il peut annoter en utilisant certaines annotations temporelles que nous avons définies. L'algorithme de transformation que nous avons développé se charge de la traduction des modèles SM annotés vers des modèles AT. Cela a nécessité bien évidemment de définir une sémantique précise pour les SMs. L'avantage de cibler un modèle AT est que ce modèle est suffisamment expressif et différents outils de *model-checking* sur AT existent. Il est important de préciser que notre contribution focalise sur la transformation des modèles dont l'entrée est un modèle SM annoté. L'étape de transformation de modèles est assurée par un outil logiciel que nous avons implémenté et dont les fonctionnalités seront détaillées plus loin dans le manuscrit. La figure 1.5 donne un aperçu des étapes de ces contributions.

La validation de la transformation est une étape critique pour toute transformation pour s'assurer de la préservation des propriétés à travers la traduction. Différentes techniques peuvent

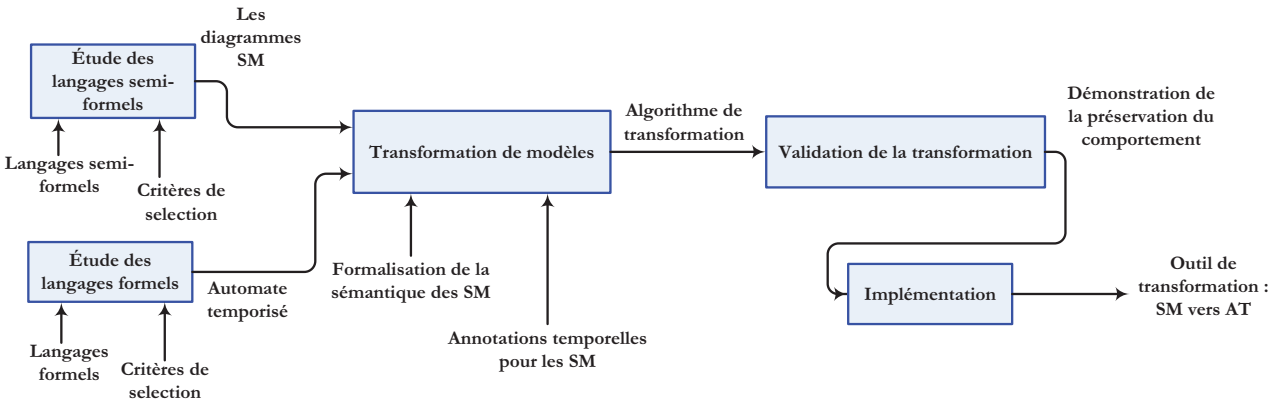

FIGURE 1.5: Étapes pour la phase de modélisation du comportement

être utilisées pour prouver différents niveaux de correspondance. Ici, nous avons adopté une démonstration de la bisimulation temporelle forte entre SM d'UML et automate temporisé (AT).

1.2.3 Étape de vérification

1.2.3.1 Vérification par observateur

Nous allons nous intéresser maintenant à l'étape de vérification et validation. Celle-ci repose sur les deux étapes présentées ci-dessus à savoir la spécification des exigences et la modélisation du comportement. Pour garantir que le processus de vérification des exigences n'influencera pas la modélisation du système, nous avons adopté une technique qui repose sur les observateurs. En effet, des patterns (génériques) d'observation ont été créés. Chaque pattern est spécifique à un type particulier d'exigence. Ainsi, le processus de vérification des exigences d'un système donné, dans notre approche, se fera par l'instanciation des patterns d'observation relatifs aux exigences du système en question. Ainsi les instances créées, appelées observateurs sont concrètement des modèles qui viendront se greffer au modèle du système et dont le rôle est de surveiller les exigences identifiées [Mekki 2009b, Mekki 2009a].

Concrètement, un observateur est un modèle en automate temporisé qui est composé avec le modèle à analyser par un produit de synchronisation sur des événements précis. Notre étude repose sur des modèles d'automates (le système et les propriétés sont agrégés sous forme d'automates). L'observateur dispose d'un ensemble de nœuds modélisant les erreurs (***KO***). Si un nœud de cet ensemble est accessible dans le modèle obtenu par la composition de l'observateur avec le modèle à vérifier, alors l'exigence correspondante peut être violée. Par exemple, pour vérifier un système Σ avec un observateur $\mathcal{O}$, le processus est comme suit : l'observateur $\mathcal{O}$ est composé avec Σ au moyen d'une composition synchrone ($\Sigma \parallel \mathcal{O}$). Une analyse d'accessibilité des états d'erreur est effectuée sur le produit de cette composition ($\Sigma \parallel \mathcal{O} \rightsquigarrow$ erreur). Si l'un de ses états (***KO***) est accessible, alors la propriété n'est pas vérifiée. Si, au contraire, aucun de ses états (***KO***) n'est accessible, le modèle satisfait bien la propriété.

1.2.3.2 Base de patterns d'observation

En se basant sur la nouvelle classification, nous avons développé une base de patterns d'observation [Mekki 2011b, Mekki 2012a]. La notion de "*Pattern*" est un concept du génie logiciel décrivant une solution à un problème récurrent dans un contexte donné, de telle manière que

cette solution soit réutilisable à chaque fois qu'on rencontre ce même problème [Gamma 1995]. Il s'agit d'un niveau d'abstraction (modélisation) supplémentaire qui vise à éviter la dispersion et à capitaliser le savoir-faire lors du développement d'applications complexes. Les "*Patterns*" proviennent à la base du domaine de l'architecture et ont été transposés au domaine du génie logiciel avec l'apparition de l'Orienté Objet.

La base développée comprend des patterns relatifs à chacun des types d'exigences temporelles identifiées. En effet, pour chaque classe d'exigences identifiées dans la nouvelle classification, nous avons défini un "*pattern* (*observateur*)" qui est concrètement un modèle automate temporisé générique. L'instanciation du pattern consiste à le dupliquer en utilisant les bonnes valeurs des paramètres relativement à l'exigence à surveiller. La base de patterns est développée une fois pour toute et elle est extensible i.e. peut être étendue pour prendre en compte de nouvelles classes d'exigences. Les différentes étapes de cette contribution sont données par la figure 1.6.

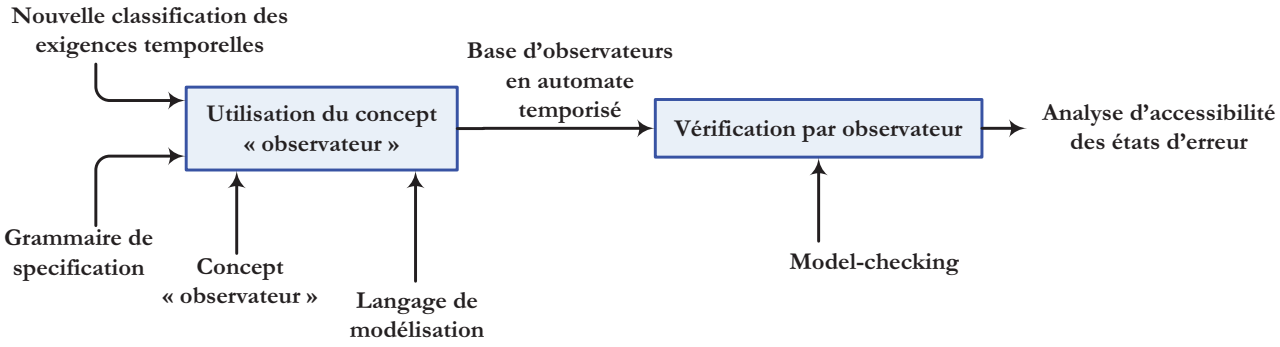

FIGURE 1.6: Étapes pour la phase de vérification

1.2.4 Proposition d'intégration du passage à niveau aux spécifications d'ERTMS niveau 2

ERTMS (European Rail Traffic Management System) est un système de contrôle commande et de signalisation européen qui a pour objectif de standardiser l'exploitation ferroviaire au sein de l'Europe, et ainsi assurer l'interopérabilité, augmenter la sécurité et favoriser la concurrence dans le secteur ferroviaire. Cependant, les SRS (System Requirement Specifications) ne prennent pas en compte le passage à niveau (PN). Cela laisse la porte ouverte aux acteurs ferroviaires à l'échelle nationale de chaque pays pour mettre en œuvre leur propres architectures pour la protection des PNs. L'idée est de proposer les bases d'une spécification des PNs dans le cadre d'ERTMS niveau 2 à travers la proposition d'une nouvelle architecture fonctionnelle pour le système de protection automatique. Nous allons pour cela mettre en œuvre les principaux mécanismes développés en termes de spécification et de vérification à travers ce cas d'étude.

1.3 Plan de la thèse

Le reste du manuscrit est structuré en 4 parties découpées en 6 chapitres :

Le chapitre 2 dresse le contexte de nos travaux. Nous introduisons, dans un premier temps, les défis posés par les systèmes complexes et critiques. Puis, nous orientons la discussion vers les propriétés temporelles rencontrées dans ces systèmes. Ensuite, nous passons en revue les langages associés à l'expression de propriétés/exigences temporelles. Plus généralement nous

nous intéressons dans ce chapitre à l'étape de spécification des exigences temporelles. Par la suite, nous présentons un aperçu d'une partie de nos contributions pour l'étape de spécification. Nous commençons par présenter une nouvelle classification des exigences temporelles les plus communément utilisées. Ensuite et afin de cadrer l'utilisateur lors de la phase d'expression des exigences, nous développons une grammaire en langage naturel pour l'expression des propriétés temporelles en langage naturel. Ensuite, nous développons un algorithme de vérification de la cohérence des exigences. Cette analyse est très intéressante du point de vue pratique puisqu'il n'est pas possible qu'une implémentation satisfasse un ensemble non cohérent d'exigences. Ce chapitre forme la première partie de ce rapport qui est dédiée à la spécification.

La deuxième partie de ce manuscrit est consacrée à l'étape de modélisation du comportement et elle est formée par les chapitres 3 et 4. En effet, afin d'assister l'utilisateur durant la phase de modélisation du comportement du système à concevoir nous allons adopter une approche dirigée par les modèles. Dans le chapitre 3 nous présenterons les concepts clés de ce type d'approche, à savoir le modèle, le méta-modèle et l'étape de la transformation de modèles.

Après avoir présenté les concepts clés de cette approche, nous entamons dans le chapitre 4 notre contribution relative à l'étape de modélisation ; tout d'abord nous présenterons l'idée générale que nous avons développée, avant de détailler les différents mécanismes mis en œuvre. Concrètement, l'idée est de permettre à l'utilisateur de manipuler une notation graphique et assez-intuitive (dans notre cas les SM d'UML) et d'en générer automatiquement des spécifications formelles, en automates temporisés. Par la suite, nous détaillons l'algorithme de transformation que nous avons élaboré. Finalement, nous démontrons la préservation du comportement à travers la transformation.

Le chapitre 5 est consacré à l'étape de vérification et validation (V&V). L'objectif de cette étape est de garantir la conformité du système par rapport à son cahier des charges (exigences). D'abord, nous passons en revue les différentes techniques de V&V tout en pointant les avantages et les inconvénients de chacune de ces méthodes. Par la suite nous donnons un aperçu de notre contribution au niveau de l'étape de V&V. Comme nous l'avons indiqué plus tôt dans ce manuscrit, l'un des objectif de notre travail est de rendre "transparents" certains aspects formels dans les étapes de spécification et de vérification. Ainsi, grâce à la grammaire de spécification définie, les exigences formulées sont automatiquement reconnues et les observateurs adéquats peuvent être générés. Ainsi, comme nous allons le montrer plus loin, la vérification est réduite à une analyse d'accessibilité. L'approche de vérification que nous proposons repose sur une base de patterns d'observation qui sera détaillée dans le chapitre 5 ainsi que sur la technique de vérification par model-checking. Ce chapitre forme la troisième partie de cette thèse qui est dédiée à la vérification.

La dernière partie de ce manuscrit est formée par le chapitre 6. Dans ce chapitre, une illustration de nos travaux relatifs à la spécification et à la vérification est proposée à travers un cas d'étude industriel, ERTMS. En effet, nous proposons une nouvelle architecture fonctionnelle du système de protection automatique d'un passage à niveau dans le cadre d'une exploitation en ERTMS niveau 2. Après avoir brièvement donné une description et les objectifs de ce cas d'étude, nous détaillons la procédure de spécification et de vérification formelle d'un ensemble d'exigences temporelles à valider sur cette nouvelle architecture que nous proposons.

Le chapitre 7 conclut cette thèse en donnant un bilan de nos contributions et présente les perspectives de notre travail.

Première partie

Spécification des exigences temporelles

Chapitre 2

Assistance à la spécification des exigences temporelles

Développement d'une grammaire de spécification et vérification de la cohérence

Résumé : *Ce chapitre est consacré à l'étape de spécification des exigences temporelles. D'abord un tour d'horizon des techniques et des notations utilisées lors de cette étape est proposé. Par la suite, en deuxième partie de ce chapitre, nous détaillons nos contributions. Nous commençons par présenter une nouvelle classification des exigences temporelles. Ensuite, afin de cadrer l'utilisateur lors de la phase d'expression des exigences, nous développons une grammaire de spécification à base de motifs en langage naturel. Puis, nous proposons un algorithme de vérification de la cohérence des exigences exprimées.*

Sommaire

2.1 Introduction

Toute étape de vérification et de validation nécessite au préalable une étape de spécification des exigences qu'on cherche à vérifier. La spécification des exigences est l'objet de ce chapitre. De manière générale, la spécification est une activité d'expression des besoins relatifs à un système. Ces besoins expriment des propriétés que le système en question doit vérifier. Comme mentionné dans le chapitre 1 de ce manuscrit, dans le cadre de nos travaux, nous nous intéressons aux exigences temporelles. Pour commencer, nous jugeons indispensable la définition des concepts suivants : propriété, exigence et exigence temporelle (section 2.2). Une fois ces concepts définis, nous abordons dans la première partie de ce chapitre l'état de l'art sur la spécification des exigences (section 2.3). Nous mettons l'accent sur les avantages et les inconvénients des différentes techniques de spécification. Par la suite, en deuxième partie de ce chapitre (section 2.4), nous détaillons nos contributions pour cette étape. Il est à noter que contrairement aux approches classiques où le point de départ est un cahier des charges déjà existant, la spécificité de notre approche est qu'elle permet de guider et d'assister l'utilisateur durant la phase d'expression et de spécification des exigences. La deuxième partie, dédiée à nos contributions, est structurée comme suit : d'abord nous proposons, en section 2.4.1, une nouvelle classification des exigences temporelles. Par la suite, afin de cadrer l'utilisateur lors de la phase d'expression de ces exigences, nous développons une grammaire à base de motifs en langage naturel, en section 2.4.2. Nous allons montrer que les exigences générées sur la base de notre grammaire sont précises et rigoureuses quand elles sont prises individuellement. Néanmoins, la phase d'expression d'exigences ne garantit en aucun cas la cohérence de l'ensemble des exigences générées. Afin de remédier à cela, un algorithme d'analyse de la cohérence des exigences est proposé en section 2.5. La section 2.6 est consacré aux conclusions.

2.2 Ingénierie système : propriétés et exigences temporelles

L'ingénierie système est un processus collaboratif qui s'appuie sur le savoir-faire, des méthodes et des outils et qui possède comme objectif la mise en œuvre (ou la définition) d'une solution qui satisfait et répond aux besoins qui ont été fixés pour un système donné. Ce processus doit prendre en compte tous les aspects ainsi que toutes les phases de la vie du système [Seidner 2009, 14th IFAC Symposium on Information Control problems in Manufacturing 2012]. Le point de départ de ce processus est l'étape d'analyse des exigences. Cette étape d'analyse permet d'identifier les exigences du système à concevoir qui peuvent être très variées et qui expriment et reflètent des propriétés également variées. Dans le cadre de nos travaux nous nous focalisons sur les exigences temporelles (voir chapitre 1). Ainsi, les systèmes que nous considérons sont des systèmes dont le comportement doit répondre à certaines contraintes de temps. Dans cette partie, nous commençons par lister les propriétés liées à notre étude (Section 2.2.1). Ensuite, nous introduisons les exigences dans la section 2.2.2 avant de présenter certaines classifications des exigences temporelles qu'on retrouve dans la littérature en section 2.2.3.

2.2.1 Propriétés

Définition 2.1 *Une propriété est définie dans [Chapurlat 2007] comme "une qualité propre, une caractéristique intrinsèque (fonctionnelle, comportementale, structurelle ou organique dépendante du temps ou non) que doit posséder une entité. Toute propriété traduit une attente, une exigence, une finalité à laquelle l'entité doit répondre".*

Nous distinguons plusieurs types de propriétés qu'on peut rencontrer dans les travaux de spécification des systèmes à contraintes de temps. Elles peuvent être classés de différentes façons. Nous présentons dans ce paragraphe une classification générique des propriétés et qu'on retrouve dans différents travaux [Schnoebelen 1999].

1. **La sûreté ou invariance** énonce que, sous certaines conditions, quelque chose de mauvais ne se produit jamais. Exemple : on ne doit jamais avoir le cas où un train est sur le passage à niveau alors que les barrières sont ouvertes.
2. **L'atteignabilité** énonce qu'un certain état peut être atteint ou non. Exemples : (1) on peut revenir à l'état initial ou (2) on ne peut pas atteindre l'état alarme.
3. **La vivacité** énonce que sous certaines conditions, quelque chose de bien finira par avoir lieu. Nous distinguons la vivacité simple ou de progrès, et la vivacité bornée :
   (a) **le progrès ou vivacité simple** énonce qu'un état est toujours atteignable. Exemple : le système peut toujours retourner à l'état initial.
   (b) **la réponse bornée ou vivacité bornée** impose un délai maximal avant lequel la situation souhaitée finira par avoir lieu. Exemple : la machine reprendra un fonctionnement normal dans au plus 5 min suite à une reconfiguration.
   (c) **l'équité ou vivacité générale** énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois. Exemple : la barrière sera levée souvent.

Ces propriétés sont généralement liées aux exigences extraites du cahier des charges du système. Dans la section suivante, nous définissons le concept d'exigences et nous nous focalisons sur les exigences temporelles.

2.2.2 Exigences et exigences temporelles

Définition 2.2 *"Une exigence correspond à une expression de besoin bien formulée émanant du client ou de toutes autres parties prenantes en lien avec le système à développer. Elle transmet un besoin en fonctionnalité (exigence fonctionnelle) ou en qualité (exigence non-fonctionnelle) que doit satisfaire le système en cours de conception" [Guillerm 2011].*

Les exigences d'un système sont généralement répertoriées en deux classes, à savoir les exigences fonctionnelles et les exigences non-fonctionnelles [Guillerm 2011, Hull 2005] :

- Les exigences fonctionnelles sont généralement liées à l'aspect opérationnel du système [Marangé 2009, Marangé 2010]. En effet, ces exigences représentent les fonctionnalités que devra satisfaire le système. Dans cette classe d'exigences on retrouve des propriétés standard telles que l'absence de blocage, l'absence de boucle infinie ou encore la possibilité de revenir à l'état initial.
- Les exigences non-fonctionnelles sont généralement liées aux critères de qualité attendus du système. En effet, ces exigences représentent des besoins complémentaires liés au fonctionnement du système. Dans cette classe d'exigences on retrouve par exemple, des contraintes de qualité de service, fiabilité, ...

Dans le cadre de notre étude, nous nous intéressons aux exigences temporelles. En effet, les exigences temporelles peuvent refléter des exigences fonctionnelles (retour à l'état initial au plus tard 3 min après une alerte), comme des exigences non-fonctionnelles (le temps de réponse, les délais min et/ou max, etc [1]). De plus, comme nous l'avons expliqué dans l'introduction (chapitre

1. Je précise ici que les contraintes temporelles peuvent être utilisées pour exprimer des exigences fonctionnelles ainsi que des exigences non-fonctionnelles.

1), de telles exigences peuvent refléter des propriétés de sûreté (invariance), de disponibilité, etc. Par exemple : *quelque chose de "mauvais" ne se produira jamais*. Ces exigences peuvent aussi refléter des propriétés de vivacité telle que la fatalité indiquant que quelque chose de bien finira par arriver, ou d'équité exprimant que si une action est possible alors le système doit l'exécuter. Les exigences temporelles sont généralement répertoriées en deux classes d'exigences [Wahl 1994] :

- Les exigences qualitatives : dans lesquelles le temps est exprimé de manière qualitative. Dans cette classe d'exigences on ne représente que les relations d'ordre entre les entités (états et/ou événements et/ou processus), telles que par exemple, le train doit être à l'arrêt avant de pouvoir ouvrir les portes.
- Les exigences quantitatives : dans lesquelles le temps est exprimé de manière quantitative. Dans cette classe d'exigences on peut exprimer des relations d'ordre entre les entités mais également spécifier ces relations par des quantités de temps. Dans cette classe on peut exprimer des exigences telles que "l'ouverture des portes doit être déclenchée 3 secondes après l'arrêt du train". Dans cette classe d'exigences, le temps peut être présenté de deux manières, à savoir discrète ou continue.

Plusieurs classifications de relations temporelles ont été proposées dans la littérature. Dans la section qui suit, nous passons en revue les plus connues.

2.2.3 Classifications existantes

Différents travaux se sont penchés sur le développement de classifications des propriétés temporelles [Allen 1984, Dasarathy 1985, Delfieu 1995, Dwyer 1999, Konrad 2005, Sadani 2007, Fontan 2008]. Nous détaillerons dans la suite les classifications les plus citées dans les travaux de recherche, à savoir Allen [Allen 1984], Dwyer [Dwyer 1999], Konrad [Konrad 2005] et Sadani [Sadani 2007].

- La logique d'intervalle d'Allen [Allen 1984] offre des opérateurs temporels pour la spécification de propriétés qui portent sur des intervalles. Allen a défini des relations de base (7 relations) sur les intervalles comme opérateurs d'ordre temporel. Les contraintes d'intervalles sont spécifiées en utilisant l'union de ces relations de base. Par exemple, la relation de précédence est l'union des relations *Meets* et *Before*, l'intersection de deux intervalles est l'union des 7 relations de base, etc. Bien que cette classification figure dans presque tous les travaux de logiques temporelles, elle présente une limite majeure : elle ne permet de présenter ni la causalité temporelle ni les exigences sur les événements.
- Dwyer [Dwyer 1999] propose une classification des exigences temporelles qualitatives dans laquelle chaque classe d'exigence est formellement exprimée à l'aide d'expressions en logiques temporelles : **CTL** [Clarke 1986] [2], **LTL** [Manna 1992] [3], **QRE** [Olender 1990] [4] ou **GIL** [Ramakrishna 1996] [5]. À l'aide de cette classification, on peut exprimer des exigences portant sur les états ou sur les événements. Dwyer a introduit également la notion de **«Scope»** (portée/contexte) dans l'expression des exigences. En effet, le **Scope** est utilisé pour indiquer l'intervalle dans lequel les exigences doivent être vérifiées. Cinq **Scopes** sont définis : **Globally**, **Before an event/state occurs**, **After an event/state occurs**, **Between two events/states** et **Until an event/state**. Néanmoins, à l'aide de cette classification on ne peut exprimer ni les exigences quantitatives ni la ponctualité.

2. **C**omputation **T**ree **L**ogic
3. **L**inear **T**emporal **L**ogic
4. **Q**uantified **R**egular **E**xpressions
5. **G**raphical **I**nterval **L**ogic

- Konrad [Konrad 2005] propose une extension temps-réel à la classification de Dwyer. Cette classification utilise les logiques temporelles **MTL** [Alur 1993b][6], **TCTL** [Alur 1992][7], et **RTGIL** [Moser 1997][8] pour exprimer formellement les exigences temporelles quantitatives. Comme Dwyer, Konrad propose l'utilisation des cinq **Scope** mentionnés ci-dessus. Néanmoins, sa classification souffre de l'absence de l'exigence ponctualité ainsi que l'absence des exigences portant sur des intervalles (du temps) bornés.
- La logique d'intervalles de Sadani [Sadani 2007] offre la possibilité d'exprimer des exigences portant sur les processus. Néanmoins, bien qu'elle offre la possibilité d'exprimer l'exigence de ponctualité, les exigences portant sur les événements ne peuvent pas être exprimées en utilisant cette classification. De plus, contrairement à Dwyer [Dwyer 1999] et Konrad [Konrad 2005] où des notations formelles (logiques temporelles) sont utilisées pour exprimer les exigences, Sadani utilise une notation graphique pour définir ses classes d'exigences.

Malgré le grand nombre de travaux sur les classifications on ne trouve pas de classification assez complète des exigences temporelles. Dans la suite, nous passerons en revue les différentes méthodes et notations utilisées pour mener le processus de spécification des exigences temporelles.

2.3 Spécification des exigences

Dans cette section, nous nous intéressons à l'étape de spécification des exigences. Nous commençons par introduire l'idée et les objectifs de cette étape en section 2.3.1. Ensuite, nous passerons en revue les techniques qui ont été proposées pour cette étape. En effet, dans la littérature on trouve deux classes de méthodes, à savoir les méthodes reposant sur des langages formels et celles qui reposent sur des langages semi-formels. Tout d'abord, nous commençons par introduire un aperçu de différentes familles de langages de spécification. Ensuite, nous enchaînons par une synthèse sur les méthodes qui reposent sur les notations formelles ainsi que les méthodes qui reposent sur les notations semi-formelles tout en détaillant les avantages et les inconvénients de chaque classe de méthodes.

2.3.1 Définition et objectifs de la spécification

La spécification est une étape de définition et d'expression des exigences de systèmes. Cette étape génère un ensemble de documents qui expriment des propriétés que le système doit satisfaire. Afin de faciliter (voire automatiser) leur exploitation par les étapes suivantes (développement, vérification et validation), les spécifications doivent être exprimées d'une manière claire, précise et non ambiguë. Différentes classes de langages de spécification peuvent être utilisées pour exprimer les exigences. Dans la section suivante (Section 2.3.2), nous passons en revue ces différentes classes.

2.3.2 Langages de spécification

Les langages de spécification peuvent être classés en trois classes : les langages naturels, les langages semi-formels et les langages formels [Chapurlat 2007].

6. **M**etric **T**emporal **L**ogic
7. **T**imed **C**omputational **T**ree **L**ogic
8. **R**eal-**T**ime **G**raphical **I**nterval **L**ogic

2.3.2.1 Langage naturel

Bien que cette classe de langages soit facile à utiliser pour les échanges entre les différents acteurs (concepteurs, développeurs, ...), le langage naturel reste souvent ambigu, non précis et très souvent différentes interprétations peuvent découler d'une même exigence exprimée en langage naturel. En conséquence, il est, par exemple, inimaginable de développer des techniques de vérification automatiques pour des spécifications en langage naturel.

2.3.2.2 Langages semi-formels

Les langages semi-formels reposent généralement sur des notations graphiques et/ou textuelles standardisées, c'est-à-dire dotées d'une syntaxe. Malgré l'utilité de ces langages pour la communication entre acteurs, les langages semi-formels souffrent souvent de l'absence d'une sémantique précise. Cela rend souvent les spécifications obtenues en utilisant cette classe de langages, non suffisamment précises. Dans cette classe on trouve des notations comme UML, BPMN, SART et SADT.

- UML (Unified Modeling Language) est un langage de modélisation objet standard possédant une popularité à la fois dans les milieux industriel et académique. UML est constitué d'un ensemble de notations graphiques. En plus, il propose une variété de diagrammes afin d'exprimer différentes vues du système : (1) une vue statique du système à travers le diagramme de classes, le diagramme d'objets, le diagramme de composants et le diagramme de déploiement, (2) une vue dynamique à travers le diagramme de séquences, le diagramme d'états-transitions (ou **S**tate **M**achine (**SM**)), le diagramme d'activités et le diagramme de collaboration et (3) une vue fonctionnelle à travers le diagramme de cas d'utilisation (UML version 2.4). Néanmoins, l'inconvénient majeur du langage UML est qu'il ne permet pas de vérifier formellement les diagrammes obtenus même si de récents travaux sont des tentatives pour formaliser certains diagrammes UML, notamment avec la proposition du F-UML [OMG 2011c].
- SADT (Structured Analysis and Design Technique) est un formalisme (associé à une notation graphique) pluridisciplinaire, d'analyse et de conception des systèmes, qui a été développée en 1977 par Douglas T. Ross [9]. SADT est une méthode de représentation structurée conçue à partir de concepts simples, et basée sur un formalisme graphique et textuel facile à apprendre et qui favorise la communication entre les utilisateurs et les concepteurs [Demri 2009]. Néanmoins, l'inconvénient majeur de cette méthode est qu'elle ne permet pas de représenter les aspects dynamiques, et s'arrête au niveau fonctionnel.

2.3.2.3 Langages formels

Les langages formels reposent sur une syntaxe bien définie et une sémantique précise, claire et non ambiguë basée sur des concepts mathématiques. Cela permet d'appliquer des techniques automatiques de vérification/validation sur les spécifications obtenues à partir de ces langages. Nous allons présenter dans la suite des exemples de langages formels les plus couramment utilisés :

1. Les logiques temporelles : ces sont des notations permettant d'exprimer, d'une manière précise et abstraite, des propriétés sur le comportement du système. La première logique temporelle a été proposée en 1977 par A. Pnueli [Pnueli 1977] afin de pouvoir exprimer

9. Douglas T. Ross est le fondateur de la société américaine Softech

des propriétés sur le comportement dynamique sur la base d'un modèle états/transitions. Elles ne peuvent pas être exprimées en utilisant la logique classique.

Les logiques temporelles sont des formalismes adaptés pour exprimer des propriétés temporelles faisant intervenir la notion d'ordonnancement, de délais, de ponctualité, ... Deux principales variantes de logiques temporelles existent dans la littérature : logiques linéaires (Linear Temporal Logic (LTL) [Pnueli 1977], Propositional Linear Temporal Logic (PLTL) [Pnueli 1981]) et les logiques branchantes (Computation Tree Logic (CTL) [Clarke 1986], CTL* [Emerson 1989, Pnueli 1977]). Certaines extensions temps-réel ont également été proposées, comme par exemple TCTL) (Timed Computation Tree Logic [Alur 1990, Alur 1993a]) qui est une extension temps-réel de CTL. Un bref aperçu de LTL et CTL est donné dans la suite de ce paragraphe.

(a) LTL : la logique temporelle linéaire (LTL) est une extension de la logique classique. Elle a été proposée pour la première fois, en 1977 par A. Pnueli, afin de spécifier des propriétés qui font référence au temps de manière qualitative. Concrètement, LTL propose des opérateurs temporels, tels que G (signifie "globalement") et F (signifie "dans le futur") lors de l'expression de formules logiques. Ces dernières peuvent être interprétées comme des séquences infinies (ou chemins) d'états. Les formules LTL sont à vérifier sur le comportement du système considéré comme un ensemble d'exécutions. Une exécution (ou *run* en anglais) est une séquence infinie d'états où chaque état a un et un seul successeur, et qui commence à un état de départ donné du système. Ainsi, pour qu'une propriété LTL soit vérifiées, il faut que l'ensemble des exécutions du système satisfasse cette propriété.

(b) CTL : une formule CTL exprime des propriétés qui doivent être vérifiées par toutes ou une partie des exécutions du système, en prenant en compte les différentes évolutions futures possibles (branchement). Deux types d'opérateurs peuvent être utilisés en CTL : les opérateurs d'états et les opérateurs de chemin. Les formules utilisant un opérateur d'état sont des expressions qui sont évaluées pour un instant donné, tandis que celles utilisant un opérateur de chemin sont évaluées sur toutes (à l'aide de l'opérateur **A**) ou une partie (à l'aide de l'opérateur **E**) des exécutions possibles.

2. Les Réseaux de Petri (RDP) : ils ont été introduits pour la première fois par Carl Adam Petri en 1962 [Petri 1962]. Ces sont des notations graphiques qui reposent sur des concepts mathématiques permettant de modéliser le comportement dynamique des systèmes concurrents [Murata 1989]. En effet, les réseaux de Petri sont des graphes orientés constituant un support pour la spécification et la conception des systèmes manufacturiers, systèmes de télécommunications [Bouali 2012], systèmes de transport : (1) d'un côté, leur notation graphique permet de représenter d'une manière intuitive la synchronisation, le conflit, le parallélisme, l'alternative ..., (2) de l'autre côté, leurs bases théoriques solides permettent d'évaluer les propriétés du système modélisé ainsi que son comportement [Murata 1989].

3. L'automate est l'un des formalismes les plus utilisés dans la spécification formelle des systèmes. Un automate est composé d'un ensemble d'états et d'un ensemble de transitions, étiquetées, entre les états. Les étiquettes de transitions modélisent les événements déclenchant ces transitions. Un automate permet de générer un langage qui consiste en l'ensemble des mots (séquence d'événements) pouvant être reconnus par l'automate. Ainsi, pour vérifier si un mot [10] donné est accepté ou non par un automate, le processus est déroulé comme suit : le mot est rentré à l'automate afin de le lire. Lorsque il est entièrement

10. Dans ce cas, un mot est un sous-ensemble de l'alphabet accepté par l'automate

lu, l'automate s'arrête dans un état dit final. Si ce dernier est un état *accepteur* alors le mot est accepté, sinon il est refusé [Hopcroft 2006]. Dans ce paragraphe nous nous focalisons sur deux variantes d'automates, à savoir l'automate à états fini (AEF) [McCulloch 1943] et l'automate temporisé [Alur 1994]. Un AEF est un modèle comportemental composé d'un nombre fini d'états. Parmi ces états, on distingue un seul état dit *initial* et un sous-ensemble d'états finaux, en plus bien sûr d'un sous-ensemble d'états ordinaux. Cette notation a été proposée par McCulloch et Pitts afin d'expliciter le comportement des neurones du cerveau humain [McCulloch 1943]. Par la suite, cette notation a été transposée dans plusieurs domaines dont la modélisation du comportement des systèmes manufacturiers. Ainsi plusieurs extensions ont été proposées pour élargir la capacité d'expression des AEF. Parmi ces extensions nous nous focalisons sur les automates temporisés. Les automates temporisés (**AT**) [Alur 1994, Bouyer 2006] sont des automates à états finis étendus par un ensemble d'horloges dont les valeurs croissent uniformément avec le passage du temps et qui peuvent être remises à zéro lors du franchissement de transitions.

L'inconvénient majeur du formalisme AEF est l'explosion combinatoire du nombre d'états du graphe modélisant le système étudié. Afin de faire face à ce problème, des techniques de réduction de l'espace d'états, d'abstraction, de vérification compositionnelle et d'ordre partiel ont été développées. [Merz 2006] passe en revue ces différentes techniques de réduction.

4. D'autres langages formels sont disponibles comme Z [Spivey 1992] et B [Abrial 1996].
   (a) *Z* : introduit par J. R. Abrial au cours des années 70 et implémenté par le "Programming Research Group" de l'Université d'Oxford pendant les années 1980, est un langage de spécification formelle de systèmes. La notation *Z* est basée sur la théorie des ensembles ainsi que sur la logique du premier ordre. Elle permet de définir ce que le système doit faire sans dire comment le faire. Plusieurs outils ont été développés pour la vérification de la syntaxe et des spécifications en *Z* [Junior 1997].
   (b) *B* : est une méthode formelle développée par J. R. Abrial *et al.* *B* couvre toutes les étapes de développement du logiciel ou de systèmes séquentiels, de la spécification jusqu'à l'implémentation qui est basée sur des machines abstraites. La méthode *B* offre la possibilité de raffiner une spécification de haut niveau en une spécification suffisamment détaillée pour pouvoir être automatiquement traduite dans un langage de programmation. Des outils comme *B-Tool* [Abrial 1996] et *Atelier-B* ont été proposés afin de favoriser l'usage du langage *B* notamment en milieu industriel, par exemple le domaine ferroviaire. Cependant, la manipulation de la méthode B reste souvent ardue. En effet, la vérification des contraintes dynamiques en *B* est réalisée en utilisant une technique de preuve. Cela exige que l'utilisateur soit un expert en spécification et en plus un expert dans l'utilisation des techniques de preuve.

Après avoir introduit certains langages de spécification, nous présenterons par la suite les méthodes de spécification. Nous donnons, dans la section suivante, une brève synthèse des méthodes de spécification les plus connues.

2.3.3 Synthèse

Une méthode de spécification qui repose sur un langage semi-formel (graphique et/ou textuel) hérite les avantages ainsi que des inconvénients du langage de spécification utilisé. En effet, la syntaxe et la notation de langages semi-formels se situent souvent à mi-chemin entre le langage naturel et le langage formel. Certains langages s'appuient sur des notations graphiques pour simplifier et rendre intuitive leur utilisation. Cependant, ce type de modélisation

présente une limitation majeure qui est l'absence d'une sémantique précise. Par conséquent, la vérification des modèles exprimés en utilisant ces méthodes n'est pas garantie. Cela est fortement pénalisant pour la mise en œuvre et l'utilisation de ces méthodes pour des applications critiques.

Une deuxième classe de méthodes utilisées lors de l'étape de spécification est celle qui repose sur des notations formelles possédant des sémantiques claires et précises. Les spécifications formelles se distinguent des autres spécifications par (1) leur pouvoir d'expression, (2) la précision des concepts utilisés et, par conséquent, (3) l'exactitude des résultats obtenus. En effet, les spécifications formelles permettent d'identifier, de repérer et de réduire les impacts des erreurs à travers une spécification complète [Hall 1990]. Selon [Bowen 1995], l'utilisation et la mise en œuvre des spécifications formelles, dans le processus de développement, permet de :

- détecter les erreurs assez tôt durant le processus de développement,
- réduire considérablement l'étape de vérification (automatisation à l'aide d'outils),
- aider dans la compréhension du système.

Et par conséquent, cela garantit une réduction significative des coûts ainsi que des délais [Bowen 1995] dans le cycle de développement. En effet, à travers leurs syntaxes formelles et leurs sémantiques claires, les spécifications formelles permettent d'exprimer précisément les propriétés qui doivent être respectées par le système. Cela est très important vu qu'une spécification rigoureuse facilite les étapes suivantes à savoir la conception, l'implémentation, la vérification et la validation. Selon le type de langage sur lequel elles reposent, les méthodes de spécification peuvent être répertoriées en quatre catégories : les méthodes algébriques, logiques, dynamiques et ensemblistes [Meyer 2001].

1. Les méthodes logiques reposent sur la théorie des types et des logiques d'ordre supérieur. Cette classe de méthodes fait appel aux théories de démonstration automatique ou semi-automatique de théorèmes. L'assistant de preuve Coq est un exemple d'outils formels qui supportent cette classe de méthodes.
2. Les méthodes à base de modèles dynamiques reposent sur une modélisation des interactions entre les processus. Dans cette classe on retrouve les systèmes de transitions (automates temporisés, réseaux de Petri), les algèbres de processus (CSP) et les logiques temporelles.
3. Les méthodes ensemblistes reposent sur des concepts (souvent appelés types) abstraits prédéfinis afin de modéliser l'état du système. La syntaxe des spécifications produites correspond aux types et aux opérations de ce modèle abstrait, alors que leur sémantique repose sur la théorie correspondante à ce modèle abstrait : théorie des ensembles, théorie des types, logique du premier ordre, etc. Dans cette classe de méthodes on retrouve VDM [Jones 1990], Z [Spivey 1992] et B [Abrial 1996].
4. Les méthodes hybrides sont des combinaisons des trois méthodes citées ci-dessus afin de profiter au mieux de leurs avantages respectifs.

Pour finir cette section, nous pouvons dire que l'utilisation de langages formels permet de générer des représentations/spécifications rigoureuses et qui sont par conséquent exploitables automatiquement. Cependant, leur manipulation nécessite certaines compétences théoriques, ainsi que la maîtrise d'une variété de notations spécifiques. De leur côté, les langages semi-formels sont généralement des notations intuitives et facile à manipuler. Cependant, les représentations obtenues en utilisant cette classe de notations souffrent d'une sémantique ambiguë qui rend difficile leur exploitation directe. Une partie de nos contributions est de proposer des

mécanismes qui tirent profit à la fois de la facilité et l'intuitivité des notations semi-formelles, et en même temps de la précision des notations formelles. En d'autres termes, l'idée est de proposer une approche qui permet d'assister l'utilisateur pour produire des spécifications rigoureuses qui reflètent les exigences temporelles qu'il veut exprimer sur un système donné, tout en évitant la manipulation des concepts formels abstraits sur lesquels repose notre processus de spécification. Nous détaillons dans la suite de ce chapitre nos contributions pour l'étape de spécification. D'abord, nous commençons par donner en section 2.4.1 une nouvelle classification des exigences temporelles. Par la suite, afin de cadrer l'utilisateur lors de la phase d'expression des exigences, nous proposons une grammaire de spécification à base de motifs en langage naturel, en section 2.4.2. Ainsi, sur la base de cette grammaire, l'utilisateur peut formuler des exigences selon une syntaxe prédéfinie. Dans la dernière partie de ce chapitre, section 2.5, nous nous intéressons au problème de cohérence entre les exigences.

2.4 Spécification des exigences temporelles

Dans le cadre de nos travaux, nous nous intéressons aux systèmes complexes à contraintes de temps, en particulier les systèmes de transport ferroviaire. La réussite de la mise en œuvre de ces systèmes repose, en grande partie, sur la communication entre ses différents sous-systèmes [Rutten 2009, Rut]. En pratique, les temps de réponse et les délais des ces communications peuvent être contraints voire critiques dans certaines situations, d'où l'intérêt porté aux exigences temporelles.

L'étude des exigences temporelles passe nécessairement par une analyse des différents types de contraintes temporelles qu'on trouve souvent dans les cahiers des charges. Dans le paragraphe suivant, nous présenterons une nouvelle classification des exigences temporelles qui comprend les principaux types d'exigences que nous avons identifiés.

2.4.1 Classification des exigences

Comme indiqué précédemment en section 2.2.3, beaucoup de travaux proposent des classifications des propriétés temporelles [Allen 1984, Dasarathy 1985, Delfieu 1995, Dwyer 1999, Konrad 2005, Sadani 2007, Fontan 2008]. Mais chacun de ces travaux manque de certains types d'exigences qu'on retrouve dans les autres travaux. Dans cette section, nous présentons une nouvelle classification des exigences temporelles, une des contributions de nos travaux. Les caractéristiques de cette nouvelle classification sont détaillées ci-dessous.

- La première caractéristique est que la classification que nous proposons permet d'exprimer la quasi-totalité[11] des classes identifiées dans les travaux précédents. En effet, les types d'exigences définies dans les classifications mentionnées plus tôt peuvent être exprimés en utilisant nos classes d'exigences, soit directement à l'aide des classes élémentaires, soit à travers une composition de ces classes. Les classes que nous proposons peuvent modéliser :
  - La réponse : ces propriétés expriment qu'un événement surveillé doit se produire ou ne peut pas se produire en réponse à un événement stimulus. Deux variantes pour ces propriétés sont définies : des propriétés qualitatives et des propriétés quantitatives (temporisées). Les différentes classes de ce type d'exigences sont données dans le tableau 2.1.

11. Toutes, sauf la classe ponctualité.

Classe	Sous-Classe	Catégorie	Nom	Description
Réponse	Obligation-Réponse	Quantitative	**Réponse-Entre**	on exige qu'un événement (E_{mon}) doit apparaitre dans un intervalle $[t_{Begin}\,;\,t_{End}]$ par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Réponse-Délai Min	on exige qu'un événement (E_{mon}) doit apparaitre après un délai min T_{min} par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Réponse-Délai Max	on exige qu'un événement (E_{mon}) doit apparaitre avant un délai max T_{max} par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Réponse Ponctualité	on exige qu'un événement (E_{mon}) doit apparaitre exactement à l'instant T par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
		Qualitative	**Réaction**	on exige qu'un événement (E_{mon})doit apparaitre après la "i^{eme}" apparition de l'événement E_{Ref}
	Interdiction-Réponse	Quantitative	**Interdiction Réponse-Entre**	on exige qu'un événement (E_{mon}) ne peut pas apparaitre dans un intervalle $[t_{Begin}\,;\,t_{End}]$ par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction Réponse-Délai Min	on exige qu'un événement (E_{mon}) ne peut pas apparaitre après un délai min T_{min} par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction Réponse-Délai Max	on exige qu'un événement (E_{mon}) ne peut pas apparaitre avant un délai max T_{max} par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction-Réponse Ponctualité	on exige qu'un événement (E_{mon}) ne peut pas apparaitre exactement à l'instant T par rapport à la "i^{eme}" apparition de l'événement E_{Ref}
		Qualitative	**Réponse-Interdite**	on exige qu'un événement (E_{mon}) ne peut pas apparaitre après la "i^{eme}" apparition de l'événement E_{Ref}

Table 2.1: Exigences temporelles : réponse

  - La précédence (qualitative et quantitative) entre les entités : ces propriétés expriment qu'un événement surveillé doit être précédé -ou ne peut pas être précédé- par un événement de référence. Comme pour la première classe, deux variantes pour ces propriétés sont définies : temporisées et non temporisées. Les différentes classes de ce type d'exigences sont données dans le tableau 2.2.
  - L'apparition et l'absence absolues : ces propriétés expriment qu'un événement surveillé doit se produire -ou ne peut pas se produire- dans l'absolu c'est-à-dire sans se référer à un stimulus ou à une référence. Les différentes classes de ce type d'exigences sont données dans le tableau 2.3.

- Le deuxième avantage de notre classification est que nous traitons de manière homogène les exigences qui portent sur les événements et les exigences qui portent sur les états. L'idée est d'associer à chaque état deux événements fictifs pour le représenter : un événement pour modéliser son activation et un événement pour modéliser sa désactivation. Ainsi en pratique, les exigences portant sur les états utilisent les événements d'activation

ainsi que les événements de désactivation de ces états.

- Nous traitons les événements répétitifs. L'idée est de raisonner par rapport à la i^{eme} apparition de l'événement de référence. En pratique, l'expression d'une exigence fait appel à une référence (événement ou processus). Contrairement aux travaux mentionnés précédemment où la référence est une unique occurrence d'un événement ou processus, nous donnons la possibilité d'exprimer des contraintes dont la référence est la n^{me} occurrence d'un événement.

Classe	Sous-Classe	Catégorie	Nom	Description
Antériorité	Obligation-Antériorité	Quantitative	**Antériorité Entre**	on exige qu'un événement (E_{mon}) doit apparaitre dans un intervalle $[t_{Begin}\,;\,t_{End}]$ avant la "i^{eme}" apparition de l'événement E_{Ref}
			Antériorité Délai Min	on exige qu'un événement (E_{mon}) doit apparaitre après un délai min T_{min} avant la "i^{eme}" apparition de l'événement E_{Ref}
			Antériorité Délai Max	on exige qu'un événement (E_{mon}) doit apparaitre avant un délai max T_{min} avant la "i^{eme}" apparition de l'événement E_{Ref}
			Antériorité Ponctualité	on exige qu'un événement (E_{mon}) doit apparaitre exactement à l'instant T avant la "i^{eme}" apparition de l'événement E_{Ref}
		Qualitative	**Antériorité**	on exige qu'un événement (E_{mon})doit apparaitre avant la "i^{eme}" apparition de l'événement E_{Ref}
	Interdiction-Antériorité	Quantitative	**Interdiction-Antériorité Entre**	on exige qu'un événement (E_{mon}) ne peut pas apparaitre dans un intervalle $[t_{Begin}\,;\,t_{End}]$ avant la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction-Antériorité Délai Min	on exige qu'un événement (E_{mon}) ne peut pas apparaitre après un délai min T_{min} avant la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction-Antériorité Délai Max	on exige qu'un événement (E_{mon}) ne peut pas apparaitre avant un délai max T_{max} avant la "i^{eme}" apparition de l'événement E_{Ref}
			Interdiction-Antériorité Ponctualité	on exige qu'un événement (E_{mon}) ne peut pas apparaitre exactement à l'instant T avant la "i^{eme}" apparition de l'événement E_{Ref}
		Qualitative	**Interdiction-Antériorité**	on exige qu'un événement (E_{mon}) ne peut pas apparaitre avant la "i^{eme}" apparition de l'événement E_{Ref}

TABLE 2.2: Exigences temporelles : précédence

Comme dans [Dwyer 1999] et [Konrad 2005], nous avons utilisé la notion de «**Scope**» (portée) dans l'expression des exigences. En effet, le **Scope** est utilisé pour indiquer l'intervalle dans lequel les exigences doivent être vérifiées. Nous avons repris quatre **Scopes** qui sont : **Globally**, **Before an event occurs**, **After an event occurs**, et **Between two events**.

Nous présentons dans la section suivante une grammaire que nous avons développé pour supporter l'expression des exigences sur la base de notre classification.

Classe	Sous-Classe	Catégorie	Nom	Description
Occurrence			**Absence**	on exige qu'un événement (E_{mon}) ne peut pas apparaitre
			Présence	on exige qu'un événement (E_{mon}) doit apparaitre

TABLE 2.3: Exigences temporelles : occurrence

2.4.2 Grammaire

Bien que l'utilisation de notations formelles pour l'expression des exigences temporelles permet d'obtenir des spécifications rigoureuses, la manipulation de telles notations reste ardue vu leur niveau d'abstraction en particulier. Ici, nous allons développer des mécanismes qui permettent d'assister l'expression des exigences temporelles, et qui permettent de générer des spécifications rigoureuses. En effet, les exigences seront exprimées sur la base d'une grammaire de spécification que nous avons développée. Ainsi les exigences formulées sur la base de notre grammaire sont des assertions composées de motifs pré-établis en langage naturel et qui ont un sens bien précis. Les assertions ainsi composées sont à la fois simples et rigoureuses. Par ailleurs, grâce à l'usage de cette grammaire, l'identification du type des exigences temporelles produites devient automatisable. La grammaire définie est ainsi utilisée pour assister et en même temps, cadrer l'utilisateur pendant la phase d'expression des exigences. Chaque phrase générée à partir de cette grammaire identifie (correspond à) une classe unique dans la classification présentée précédemment.

Ci-dessous, nous présentons la grammaire développée en utilisant la notation BNF (Backus-Naur Form) [Salmon 1992].

<Property> : := <scope> " :" <specification> ;
<scope> : := "globally" | "before" < entity > | "after" < entity> | "between"' <entity > "and"' <entity> ;
<Specification> : := <entity> <obligation> <occurrenceType> ;
<entity> : := <event> | activate(<state>) | deactivate(<state>) ;
<obligation> : := "must" | "cannot" ;
<occurrenceType> : := ("occur" [<referenceOccur>])|("precede" <referencePrecede>) ;
<referenceOccur> : := ([<punctuality > | <fromAdelay> | <BeforeAdelay> | <Between>] "after") (<Repetition> | <LastOcc>) ;
<referencePrecede> : := ([<punctuality > | <fromAdelay> | <BeforeAdelay> | <Between>] "before") <OccurrenceOfRef> ;
<punctuality> : := "exactly" <delay> ;
<fromAdelay> : := "after a minimum delay of" <delay> ;
<BeforeAdelay> : := "before a maximum delay of" <delay> ;
<Between> : := "between a minimum delay of" <delay> "and a maximum delay of" <delay> ;
<Repetition> : := <OccurrenceOfRef> and (("Regardless following occurrences") | ("before the next occurrence")) ;
<OccurrenceOfRef> := "i th occurrence of" <entity> ;
<LastOcc> : := "the last occurrence of" <entity> ;

<**delay**> : := <integer> “timeunit” ;
<**integer**> : := <i>+ ;
<**digit**> : := {0|1|2|3|4|5|6|7|8|9} ;
<**i**> : := <number> | <i> <digit> ;
<**number**> : := {1|2|3|4|5|6|7|8|9} ;

Concrètement, pour exprimer une exigence, l'utilisateur doit définir 5 éléments pour chaque contrainte. Il commence par choisir le **scope**. Ensuite, il introduit l'événement surveillé. Par la suite, il choisit le type de l'exigence (obligation ou interdiction). Puis, il définit le type d'occurrence (délai min, délai max, etc) et finalement, il introduit l'événement de référence.

Exemple 2.1 *Soit par exemple une exigence $\mathcal{E}$ qui exprime que le signal arrêt d'urgence* urg *doit être déclenché au plus tard 3 unités de temps après l'avoir signalé au conducteur en envoyant un signal* sig *au contrôleur à bord du train. Cette exigence est exprimée en utilisant notre grammaire comme suit :* ***Globally*** urg ***must occur before a maximum delay of*** 3 ***t.u after*** sig*. Dans cet exemple, l'assertion obtenue identifie la classe “Réponse Délai Max” (voir tableau 2.1).*

En pratique, l'utilisation de la grammaire permet (1) d'aider et de cadrer l'utilisateur lors de la phase d'expression des exigences et (2) permet d'identifier la classe de l'exigence exprimée. Néanmoins, comme mentionné précédemment, la phase d'expression d'exigences ne garantit pas la cohérence de l'ensemble des exigences obtenues. Pour cela, un algorithme d'analyse de la cohérence des exigences est proposé dans la section suivante.

2.5 Cohérence des exigences

Comme nous l'avons détaillé dans les parties précédentes, l'étape de spécification des exigences consiste à exprimer l'ensemble des contraintes que le système doit satisfaire. De son côté, l'étape de vérification a pour but de déterminer si une conception $\mathcal{M}$ satisfait l'ensemble des exigences $\mathcal{S}$ (on note $\mathcal{M} \vDash \mathcal{S}$). Cependant, dans le cas où les exigences exprimées sont incohérentes les unes avec les autres, aucune implémentation ne pourra satisfaire l'ensemble des exigences données. En pratique, le processus de vérification n'examine pas la cohérence de l'ensemble d'exigences, mais il vérifie plutôt si une implémentation donnée respecte un ensemble d'exigences. Par conséquent, cette analyse de cohérence permet un gain important puisqu'elle permet d'identifier (assez tôt) les erreurs de spécification avant d'exécuter le processus de vérification sur le modèle de l'implémentation.

Une des contributions de nos travaux est de définir et de développer des mécanismes automatiques pour vérifier la cohérence des exigences temporelles. La mise en œuvre de ces mécanismes d'analyse de cohérence nécessite une étape de formalisation du problème. Cette formalisation fera l'objet de la section 2.5.1. Ensuite, nous donnerons un algorithme (section 2.5.2) qui permet d'analyser et d'identifier les sources d'incohérence entre les exigences temporelles.

2.5.1 Représentation des exigences temporelles

L'objectif de cette étape consiste à représenter nos exigences temporelles sous une forme exploitable pour l'étape d'analyse de cohérence. Nous rappelons ici que notre objectif est de vérifier la cohérence d'un ensemble d'exigences temporelles exprimées sur la base de notre grammaire de spécification définie précédemment. Nous rappelons également que la définition

de cette grammaire se base sur la classification des exigences temporelles déjà proposée en section 2.4.1 de ce chapitre.

Les relations peuvent être des relations qualitatives (réponse, ordre, etc) ou des relations quantitatives caractérisées par une quantité de temps (délai max, délai min, etc).

Ici, nous allons développer une représentation des exigences temporelles sous forme d'un graphe orienté pondéré. Dans ce graphe, les nœuds représentent les entités (dans notre cas, les événements) et les arcs orientés -et pondérés- représentent les relations temporelles entre les nœuds. Le sens d'un arc indique une relation de précédence ou d'ordre entre les événements : entre le nœud source et le nœud cible. Or, étant donné le grand nombre de relations qu'on peut définir, nous avons défini plusieurs types d'annotations qu'on associe aux arcs pour pouvoir exprimer ces différents types de relations. Par défaut, un arc sans annotation définit une relation d'obligation de succession entre les nœuds. Par ailleurs, les pondérations sont utilisées pour représenter la quantification (le poids) d'une relation (délai min, délai max, etc) et l'interdiction entre les événements (événement e1 ne peut pas se produire avant l'événement e2). Ces deux types de pondérations (la quantification et l'interdiction) peuvent être combinés. L'ensemble des pondérations utilisées est donné dans le tableau 2.5.

	Symbole	Signification
1		obligation (par défaut)
2	!	interdiction
3	$+\alpha$	après α unité de temps (délai min)
4	$-\alpha$	avant α unité de temps (délai max)

Table 2.5: Liste des pondérations

2.5.2 Algorithme

2.5.2.1 Types d'incohérence

Une analyse des incohérences possibles entre les exigences nous a permis d'identifier trois types d'incohérences qui sont :

1. Présence de cycle (ou incohérence d'ordre) : cela se produit quand les exigences exprimées donnent lieu à des situations comme la suivante :
   - a) l'événement e_1 doit se produire avant l'événement e_2
   - b) l'événement e_2 doit se produire avant l'événement e_3
   - c) l'événement e_3 doit se produire avant l'événement e_1

   Le graphe associé à cet ensemble d'exigences est formé comme suit : l'exigence (a) est exprimée par un arc orienté de e_1 vers e_2 et de la même façon l'exigence (b) (respectivement (c)) est exprimée par un arc orienté de e_2 vers e_3 (respectivement de e_3 vers e_1). Ainsi, l'ensemble (a) + (b) + (c) implique qu'il existe un cycle entre e_1, e_2 et e_3. Par conséquent, la présence de cycle a été identifiée comme une source d'incohérence entre les exigences.
2. Incohérence des délais : cela se produit quand les exigences exprimées définissent des délais incohérents entre certains événements. Un scénario qui mène à une incohérence des délais peut être défini comme suit : une contrainte (a) exige que le délai séparant l'apparition d'événements e_1 et e_2 doit être supérieur à 10 unités de temps et une contrainte (b) exige que ce délai doit être inférieur à 7 unités de temps. Ainsi, dans le graphique associé à ces exigences nous trouvons (i) un chemin du e_1 vers e_2 avec un poids "+10" et (ii) un autre

chemin du e_1 vers e_2 avec un poids "-7". Ce scénario définit le deuxième type d'incohérence qui est relatif à l'incohérence des pondérations des différents chemins entre deux mêmes événements.

3. Incohérence d'existence/interdiction : cela se produit lorsque certaines exigences (exigences d'interdiction) *interdisent* l'apparition d'un événement e_1 avant (respectivement après) un autre événement e_2 alors que d'autres exigences expriment que e_1 *doit apparaitre* avant (respectivement après) e_2. Cela se traduit dans le graphe associé par un arc ou chemin d'interdiction (étiqueté avec " !") reliant e_1 à e_2 et un autre arc ou chemin allant de e_1 vers e_2. Ce scénario définit le troisième type d'incohérence.

2.5.2.2 Algorithme

Après avoir donné les trois types d'incohérence que nous avons identifiés, nous proposons un algorithme qui permet d'identifier ces incohérences. Cet algorithme met en œuvre plusieurs mécanismes récursifs ainsi qu'un ensemble de structures de données afin de parcourir et explorer le graphe des exigences et de vérifier leur cohérence. En effet, pour chaque nœud N du graphe d'exigences, l'algorithme parcourt et explore tous les arcs (chemins) entrants et sortants. Ces chemins sont les exigences qui font référence directement ou indirectement à l'événement associé au nœud N. Une version très simplifiée de l'algorithme que nous proposons est donnée ci-dessous.

```
/* La fonction "main" */
Boolean check(List Requirement LR)
{
List Requirement R_o=oblig(LR) ;
List Requirement R_f =forbid(LR) ;
List Requirement GR=global(R_o) ;
si not checkConsistency(nodes(GR)) alors
  return false
pour tout i ∈ R_o faire
  si not checkGraph(nodes(scopeMatch(i,R_o))) alors
    return false
pour tout j ∈ R_f faire
  si not checkGraphForbid(j,nodes(scopeMatch(j,R_o))) alors
    return false
return true
}

List Requirement oblig(List Requirement LR)
{
/* retourner la liste d'exigences d'obligation exprimées dans LR */
}

List Requirement forbid(List Requirement LR)
{
/* retourner la liste d'exigences d'interdiction exprimées dans LR */
}
```

List Node nodes(List Requirement **LR**)
{
/ traduire la liste d'exigences **LR** dans un graphe orienté formé par une liste des nœuds (représentent les événements) avec ses arcs associés (représentent les exigences) */*
}

List Requirement global(List Requirement **LR**)
{
/ retourner la liste d'exigences avec un scope **global** exprimées dans **LR** */*
}

List Requirement scopeMatch(Requirement **r**, List Requirement **LR**)
{
/ retourner la liste d'exigences, dont le scope est celui de **r**, exprimées dans **LR**. Par exemple, si le scope de **r** est **after e** alors que toutes les autres exigences exprimées dans **LR** ont un scope **global**,* scopeMatch *retrourne le sous-graphe formé par les nœuds successeurs du nœud **e** */*
}

/ procédure d'analyse du graphe **G** exprimé sous forme d'une liste de nœuds **LN** avec leurs arcs associés */*
}
Boolean checkGraph(List Node LN)
{
pour tout $n \in LN$ **faire**
  List Node succ = succ(n) ;
  si succ = CYCLE **alors**
    return $false$
  si succ = $\varnothing$ **alors**
    continue
  pour tout $m \in succ$ **faire**
    boolean b := checkPathsWeights(listPath(n->paths,m)) ;
    si not b **alors**
      return $false$
return $true$;
}

succ(Node n, List Node LN)
{
/ permet de déterminer la liste des nœuds atteignables, à partir d'un nœud donné n, dans un graphe formé par une liste de nœuds **LN** et permet d'identifier tous les chemins possibles à partir de n. Cette procédure permet également de vérifie la présence de cycles et de renvoyer CYCLE dans le cas où elle l'identifie */*
}

listPath(list Path LP, node n)
{
/ retourner la liste de chemins, qui mènent au nœud **n**, exprimées dans **LP** */*

```
}

checkGraphForbid(Requirement fr, List Node LN)
{
/* vérifie la cohérence de l'exigence d'interdiction fr avec le graphe d'exigences formé par la
liste de nœuds LN */
}

Boolean checkPathsWeights(list Path LP)
{
maxPositive := 0 ;
maxNegative := 0 ;
pour tout p ∈ LP faire

  si w := p.weight > 0 alors

    si w > maxPositive alors
      maxPositive := w ;
  si w < 0 alors

    si |w| > maxPositive alors
      maxPositive := |w| ;
si maxPositive < maxNegative alors
  return true ;
sinon
  return false ;
}
```

L'algorithme procède comme suit : soit $\mathcal{E}$ un ensemble d'exigences dont on cherche à vérifier la cohérence. L'algorithme commence par générer un graphe orienté ($\mathcal{G}$) à partir de l'ensemble $\mathcal{E}$. En effet, toute exigence $e \in \mathcal{E}$ est représentée par un arc dans $\mathcal{G}$. Une fois dessiné, le graphe $\mathcal{G}$ est utilisé pour déterminer les incohérences entre les exigences exprimées dans $\mathcal{E}$. En effet, pour tout couple de nœuds (src,$dest$) dans $\mathcal{G}$ avec $dest$ est accessible à partir de src, l'algorithme commence par déterminer tous les chemins qui mènent de src vers $dest$. Lors de cette étape, tous les cycles sont détectés et par conséquent toutes les incohérences d'ordre (de succession) sont identifiées. L'ensemble des chemins menant de src vers $dest$ est noté $\mathcal{LP}_{(src,dest)}$. Chaque chemin dans $\mathcal{LP}_{(src,dest)}$ est composé uniquement d'arcs d'obligation (les exigences définies avec **must occur**). Ensuite, le "poids" de chaque chemin est déterminé en utilisant la fonction *checkPathsWeights ($\mathcal{LP}$)*.

Le poids d'un chemin p est déterminé en utilisant les annotations définies dans le tableau 2.5 comme suit :

- si pour chaque arc a dans p, a est étiqueté soit par l'annotation par défaut ($\varnothing$), ou par une annotation positive ($+\alpha$), le poids de p est la somme "s" de toutes les annotations positives. Ceci peut être interprété comme $dest$ **must occur at least s t.u after** src.
- si pour chaque arc a dans p, a est étiqueté soit par l'annotation par défaut ($\varnothing$), ou par une annotation négative ($-\alpha$), le poids du p est la somme s de toutes les annotations négatives. Ceci peut être interprété comme $dest$ **must occur at most s t.u after** src.

- dans le cas où certains arcs de p sont étiquetés avec des annotations positives ($+\alpha$), et d'autres arcs avec des annotations négatives ($-\alpha$), le poids de p est obtenu en additionnant uniquement les poids positifs ($\Sigma(+\alpha_i)$). En effet, seul l'information concernant le délai min entre *src* et *dest* pourra être déduite dans ce cas.

Ensuite, *checkPathsWeights()* compare $maxPositive$ (le max des poids des chemins positifs) et $maxNegative$ (le max des poids absolues des chemins négatifs). Une incohérence de délais est détectée si $maxPositive > maxNegative$. En effet, soient par exemple $maxPositive$ = 10 et $maxNegative = 7$. Cela signifie que *dest* doit apparaitre avant 7 u.t (unité de temps) avant *src* et en même temps *dest* doit apparaitre après 10 u.t après *src*, ce qui est évidemment incohérent.

Ensuite, les exigences d'interdiction $\mathcal{FR}$ sont extraites de l'ensemble des exigences $\mathcal{E}$. Pour chaque exigence d'interdiction $ee \in \mathcal{FR}$, l'algorithme commence par calculer $\mathcal{LP}_{(source(ee),target(ee))}$. Une incohérence d'existence est détectée si $Card(\mathcal{LP}_{(source(ee),target(ee))}) > 0$. En effet, $Card(\mathcal{LP}_{(source(ee),target(ee))}) > 0$ signifie qu'il existe au moins un chemin qui mène de $source(ee)$ vers $target(ee)$, ce qui est contradictoire avec l'exigence ee.

2.5.2.3 Exemple

La figure 2.1 représente un exemple de graphe orienté associé à un ensemble d'exigences. En fait, l'ensemble d'exigences traité dans cet exemple est inspiré d'un cas d'étude autour d'un système de contrôle de passage à niveau. Soit un système composé de trois modules "Train", "Contrôleur" et "Barrière", qui opèrent en parallèle et se synchronisent sur les événements : *Approach*, *Close* et *Exit* émis par "Train", *GoDown* et *GoUp* émis par "Contrôleur" et, finalement, *Down* et *Open* émis par la "Barrière". Soit l'ensemble suivant d'exigences :

1. $\mathcal{R}_1$: **Globally** *Close* **must occur after** *Approach*.
2. $\mathcal{R}_2$: **Globally** *Down* **must occur after a minimum delay of 6 tu after** *GoDown*.
3. $\mathcal{R}_3$: **Globally** *Down* **cannot occur before a maximum delay of 10 tu after** *Close*.
4. $\mathcal{R}_4$: **Globally** *GoUp* **must occur after a minimum delay of 6 tu after** *Exit*.
5. $\mathcal{R}_5$: **Globally** *Exit* **must occur after a minimum delay of 15 tu after** *Down*.
6. $\mathcal{R}_6$: **Globally** *GoUp* **must occur before a maximum delay of 20 tu after** *Down*.
7. $\mathcal{R}_7$: **Globally** *Open* **must occur after a minimum delay of 3 tu after** *GoUp*.
8. $\mathcal{R}_8$: **Globally** *Open* **cannot occur after** *Exit*.
9. $\mathcal{R}_9$: **Globally** *GoDown* **must occur after** *Open*.

Ainsi, le graphe qui permet de représenter ces exigences est donné en figure 2.1.

2.5.2.4 Outil logiciel

Un outil a été développé [Cherif 2012] afin d'assister l'étape d'expression des exigences et automatiser l'étape d'analyse de la cohérence. Certaines fonctionnalités et options proposées par cet outil sont détaillées dans la suite et sont illustrées par des captures d'écran de l'outil. En pratique, la première étape consiste à introduire les exigences. Cette étape est assurée à l'aide d'une interface graphique. En effet, cette interface repose sur la grammaire de spécification que nous avons proposée et permet à l'utilisateur de composer les assertions afin d'exprimer ses exigences. Les figures 2.2, 2.3 et 2.4 sont des captures d'écran qui illustrent le processus d'expression des exigences : assistance à la composition des assertions en figure 2.2 et 2.3 et visualisation sous forme d'un graphe des exigences introduites en figure 2.4.

L'outil donne le choix entre deux modes pour la vérification de la cohérence :

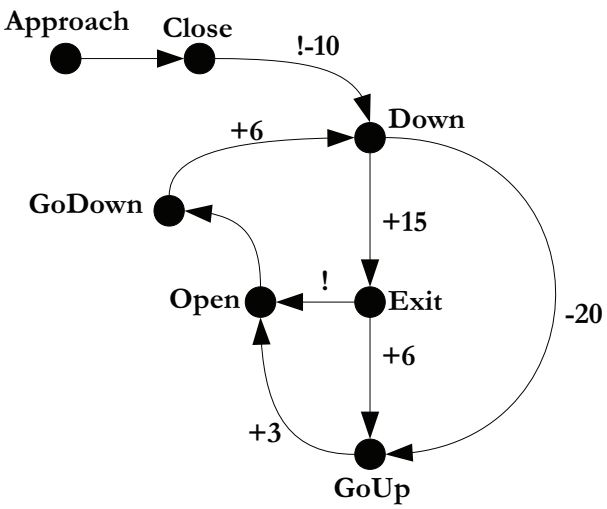

FIGURE 2.1: Graphe d'exigences

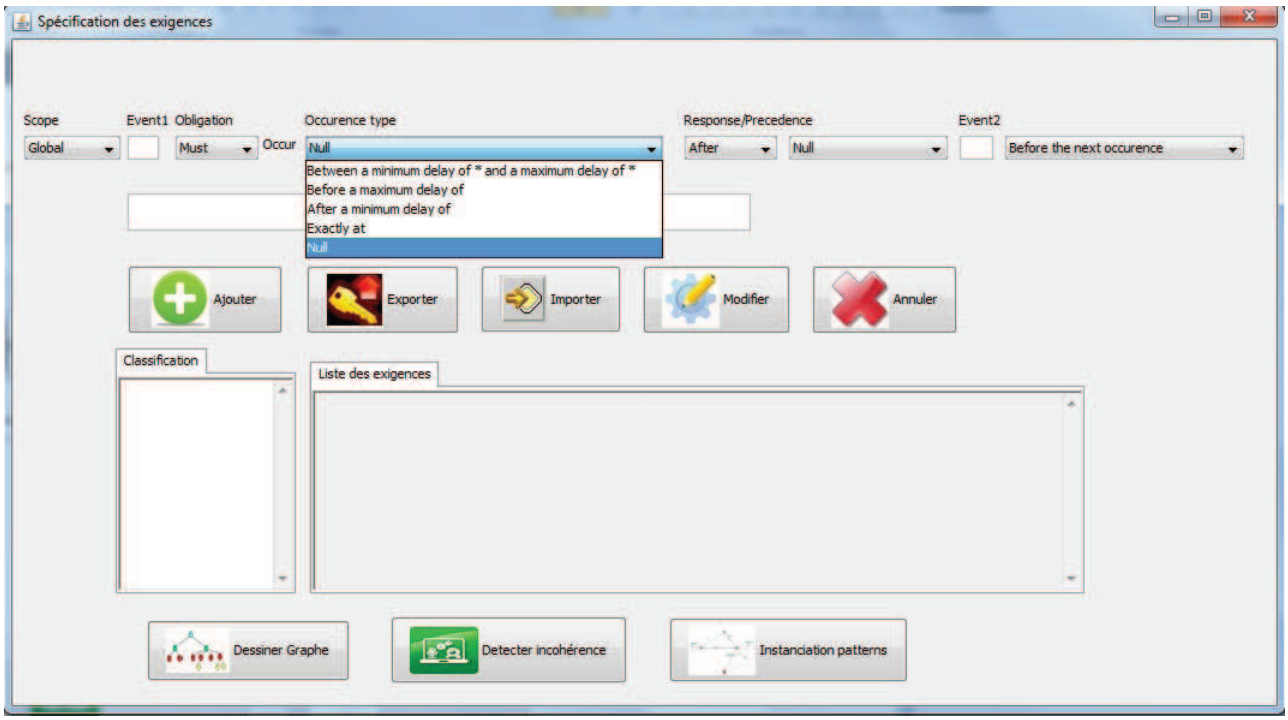

FIGURE 2.2: Expression d'exigence : composition d'assertions

– Suite à l'ajout de chaque exigence, le processus de vérification de cohérence est lancé sur l'ensemble courant des exigences.
– Á la fin, la vérification de la cohérence est lancée à la fin de l'introduction de l'ensemble des exigences

Cette analyse permet d'identifier les sources des incohérences présentées précédemment à savoir : la détection du cycle, l'incohérence de délais et l'incohérence d'apparition. Figure 2.5 illustre le déroulement de cette analyse.

La figure 2.6 donne le résultat de l'analyse de la cohérence de l'ensemble des exigences données en section 2.5.2.3. Cette analyse a identifié 3 incohérences entre les exigences exprimées, qui sont :

1. Une incohérence quantitative entre les chemins qui relient *Down* et *GoUp*

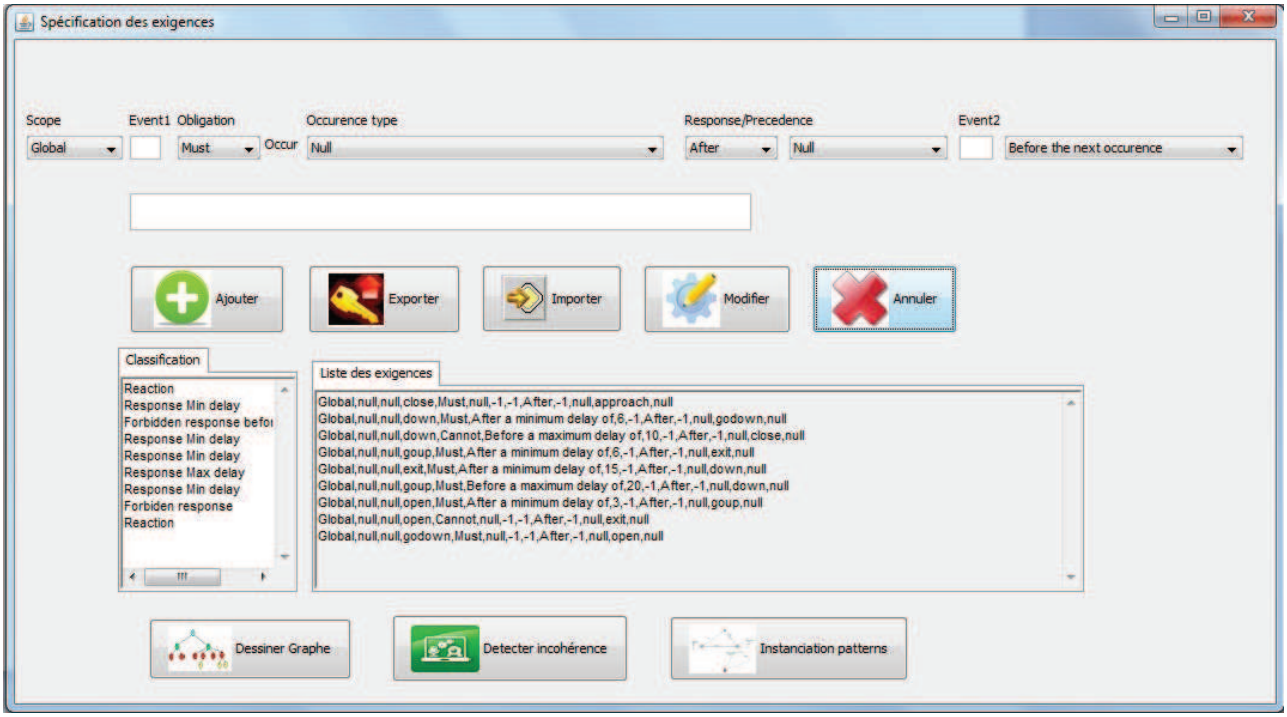

FIGURE 2.3: Expression d'exigence : Liste d'exigences

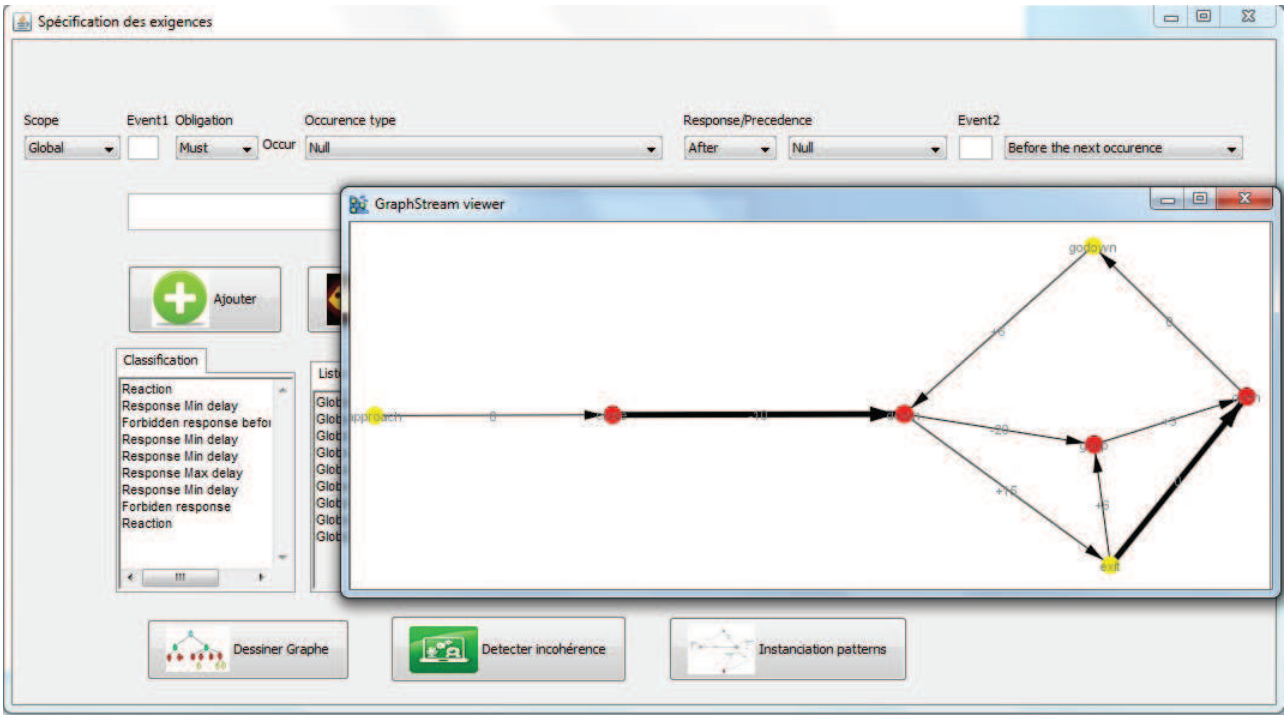

FIGURE 2.4: Expression d'exigence : visualisation sous forme d'un graphe des exigences introduites

2. Une incohérence d'apparition/interdiction entre les chemins reliant *Exit* et *Open*

3. Un cycle autour des nœuds : *Down*, *Exit*, *GoUp*, *Open* et *GoDown*.

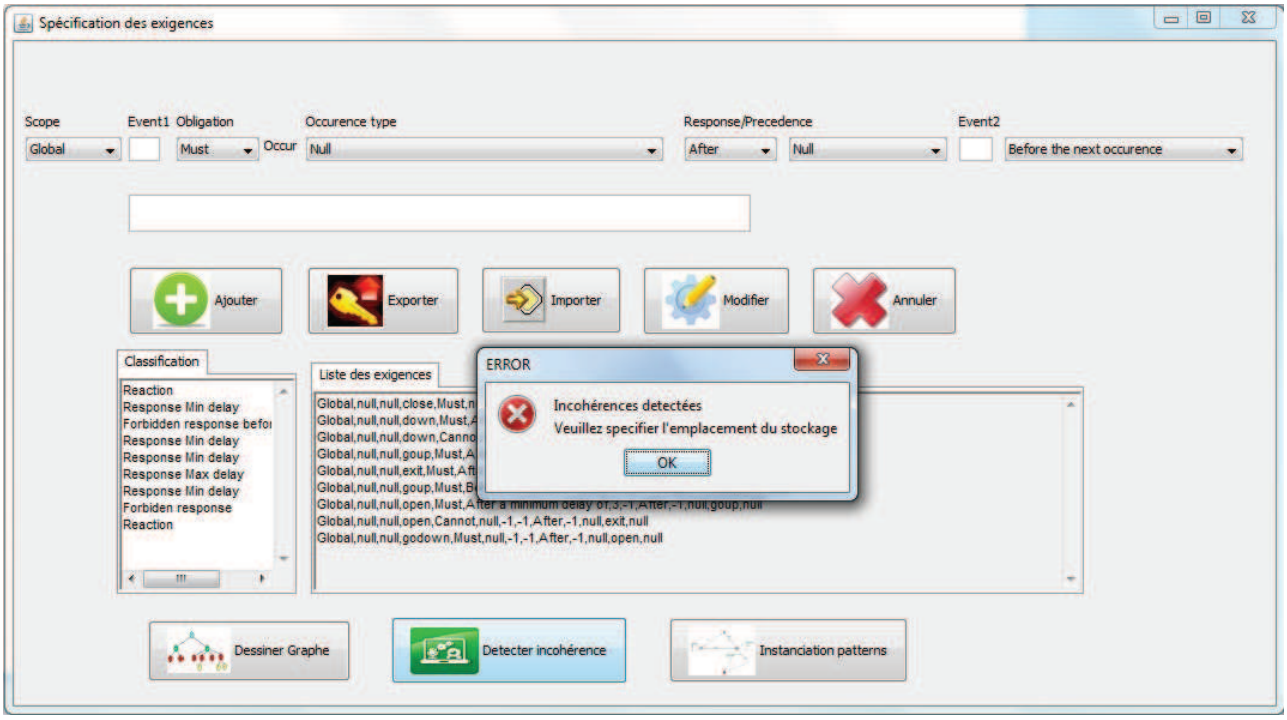

FIGURE 2.5: Analyse de la cohérence

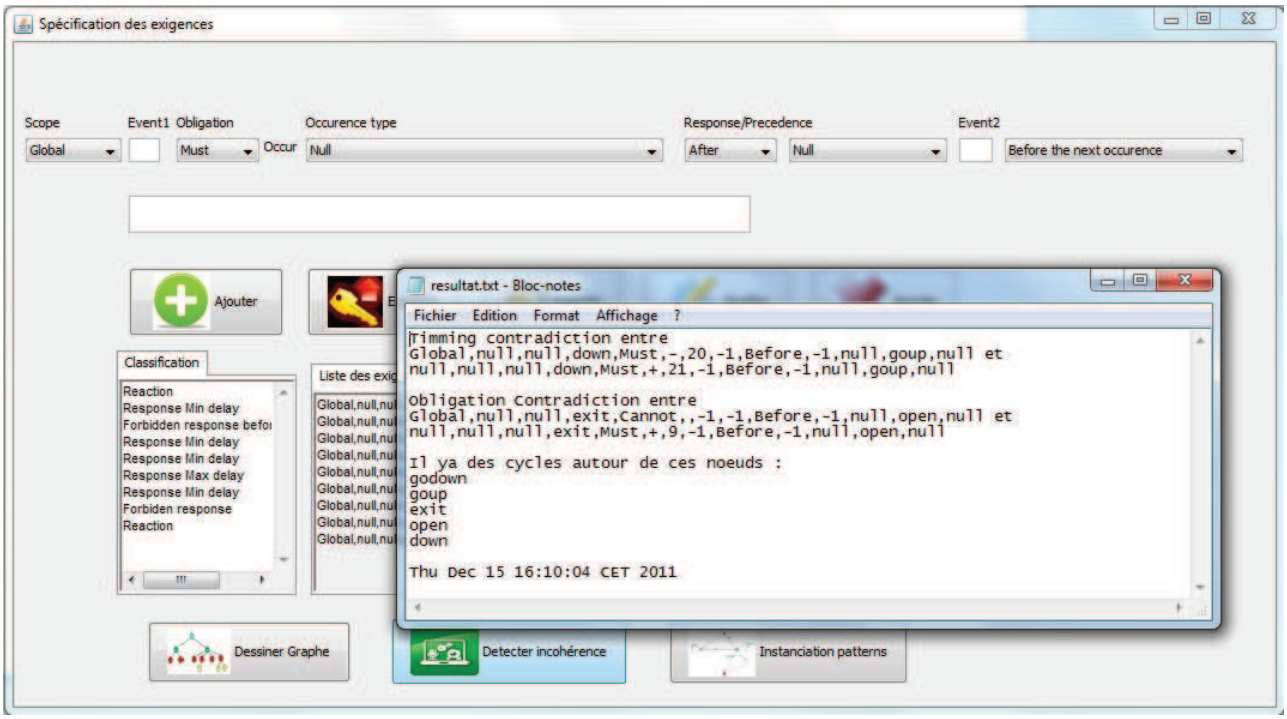

FIGURE 2.6: Résultat d'analyse de la cohérence

2.6 Conclusions

Comme nous l'avons expliqué au début de ce manuscrit, l'étape de spécification des exigences constitue une base pour les autres tâches : conception, implémentation et surtout pour la vérification et la validation. L'idéal est que cette étape fasse recours à un ensemble de langages permettant de disposer de descriptions précises et claires du système à étudier. Ces descriptions

permettent d'éviter les ambiguïtés lors de l'interprétation et rendent automatique la phase de vérification et validation. Néanmoins, bien que les langages formels aient des bases théoriques solides, leur manipulation reste une tâche difficile et une source importante d'erreurs. Notre idée est de proposer des mécanismes qui permettent de guider l'utilisateur lors de la phase de spécification des exigences, de manière à produire des spécifications rigoureuses. Nous avons commencé par définir une nouvelle classification des exigences temporelles. Cette classification présente l'avantage qu'elle permet de traiter les exigences qui portent sur les événements ainsi que les exigences qui portent sur les états. Ainsi, afin de cadrer l'utilisateur lors de la phase d'extraction des exigences, nous avons défini une grammaire à base de motifs en langage naturel. Par ailleurs, nous avons développé un algorithme de vérification de la cohérence des exigences exprimées. Cela est très intéressant d'un point d'un vue pratique puisqu'il n'est pas possible de satisfaire un ensemble d'exigences non-cohérentes.

Afin d'introduire la partie suivante, il est important de rappeler que le processus de vérification et validation d'un système quelconque repose sur deux éléments : la spécification des exigences et le modèle développé en réponse à la spécification. Dans ce chapitre nous avons passé en revue notre contribution pour l'étape de spécification. Dans la partie suivante, nous présenterons l'étape de modélisation.

Deuxième partie

Modélisation du Comportement

Chapitre 3

Phase de modélisation du comportement : introduction à l'ingénierie dirigée par les modèles

Résumé : *Dans cette deuxième partie du manuscrit, nous allons nous intéresser à la modélisation du comportement du système. Dans ce chapitre 3, nous allons présenter les principaux concepts relatifs à l'ingénierie dirigée par les modèles. Ainsi, nous allons préparer la présentation de notre contribution relative à la modélisation du comportement, et qui fera l'objet du chapitre 4.*

Sommaire

3.1 Introduction

Comme nous l'avons expliqué dans le chapitre introductif, le deuxième volet de notre contribution porte sur la phase de modélisation du comportement. Le modèle de comportement est une entrée à la phase de vérification qui confronte le modèle de l'implémentation aux exigences exprimées lors de la phase de spécification. Notre contribution porte principalement sur la transformation de modèles, et le travail réalisé peut être classé dans le contexte ADM (Architecture Dirigée par les Modèles)(ou MDA en anglais (Model Driven Architecture)).

Ce chapitre est ainsi consacré à la présentation des principaux concepts relatifs à l'ADM. Il est structuré comme suit : tout d'abord, nous présenterons les notions de "modèle" et de "modélisation". Ensuite, nous introduisons deux activités caractéristiques de l'ADM qui sont la "méta-modélisation" et la "transformation de modèles". Enfin, la dernière partie du chapitre est consacrée à l'étude d'équivalence entre les modèles. En particulier, nous mettrons l'accent sur la technique de bisimulation que nous allons utiliser dans le chapitre suivant.

3.2 Définition et objectifs de la modélisation

La transformation des besoins en produit final nécessite la mise en œuvre de processus dont le but est de transformer progressivement les concepts abstraits en une ou plusieurs représentations du système à concevoir. Ces représentations permettent d'avoir différentes vues du système. De point de vue pratique, cela permet de conceptualiser, concevoir, tester, simuler, prouver/valider et communiquer (entre les différents intervenants : client, concepteurs, développeurs, ...) le système à concevoir. C'est l'activité de modélisation qui permet de définir des représentations du système tout au long de son cycle de développement. Dans cette partie du manuscrit, nous nous focalisons sur la modélisation du comportement dont l'objectif dans le cas de notre étude, est de produire une représentation du comportement du système, et qui servira par la suite à vérifier les exigences comportementales définies lors de la phase de spécification. Or, en pratique, un ensemble de modèles (représentations) comportementaux peut être défini pour le même système. L'idée d'avoir plusieurs représentations est intéressante du point de vue chronologique (l'ajout et l'intégration des fonctions et/ou modules) ou technologique (passer d'une plateforme d'implémentation à une autre) dans le cycle de développement du système. Cependant passer d'une représentation à une autre du même produit reste une étape délicate à réaliser, d'où l'idée de développer des techniques pour accompagner la transformation de modèles.

3.3 L'architecture dirigée par les modèles : concepts de base

Le principe du développement classique est basé sur le couple "*technologie*, *architecture*" utilisé. En d'autres termes, ce principe de développement est centré sur le "code". Cette méthode de développement présente un inconvénient majeur, dû à sa rigidité et à son manque de flexibilité. En effet, à chaque changement de technologie ou modification de l'architecture, la totalité (ou une partie importante) du travail doit être refaite. Par conséquent, cela contraint la mise à jour des chaînes de production des entreprises ainsi que l'évolution de l'ensemble de logiciels et applications utilisées dans ces chaînes. Cela provoque souvent des retards et des coûts considérables et freine l'évolution des systèmes. Afin de suivre les avancées technologiques et maîtriser la complexité croissante du développement, de nouvelles approches ont été proposées. Ces approches ont fait émerger de nouveau concepts comme le paradigme objet. Le paradigme objet a introduit des contributions majeures à l'approche classique. Ces contributions

concernent notamment l'aspect structurel de l'architecture d'un système, ou l'amélioration de sa programmation. Elles ont permis d'introduire :

1. Les designs patterns pour permettre de décrire les solutions récurrentes et ainsi favoriser la reutilisabilité et la capitalisation,
2. La programmation par aspects qui est un paradigme de programmation transversal, et n'est pas lié à un langage de programmation particulier. Elle permet de séparer les considérations techniques des descriptions métier dans une application.

Malgré ces améliorations, l'approche classique souffre toujours des inconvénients cités ci-dessus. D'où l'idée d'introduire une approche de développement qui se concentre sur une vue plus abstraite par rapport au développement classique. L'objectif de cette nouvelle approche est d'améliorer le cycle de développement des systèmes tout en proposant des solutions aux inconvénients mentionnés ci-dessus. C'est dans ce contexte que les approches dirigées par les modèles (Model Driven - MD) ont été proposées comme solution pour pallier aux déficiences de l'approche traditionnelle. Dans la suite de ce mémoire nous nous focalisons sur l'approche la plus connue qui est la ADM [1] (**M**odel **D**riven **A**rchitecture - MDA). Les évolutions des approches de développement sont données dans la figure 3.1 [2].

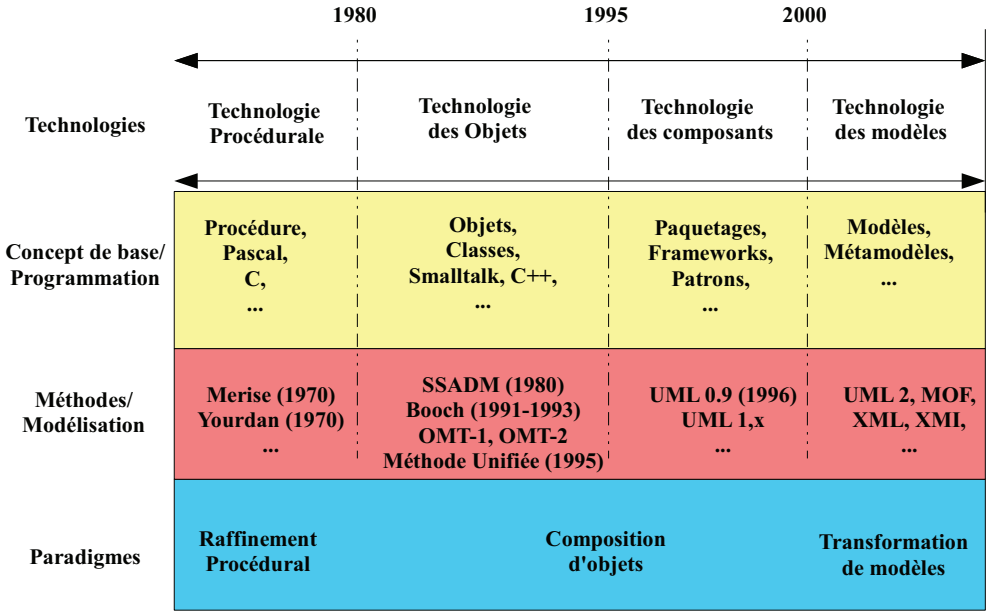

FIGURE 3.1: Évolution du Génie Logiciel

L'idée clé de l'ADM est de centrer le travail sur une vue plus abstraite que l'approche traditionnelle. Ainsi, la principale caractéristique de l'ADM est qu'elle est basée sur une ingénierie générative. En effet, la plus grande partie (voire la totalité) de l'architecture du produit à développer est obtenue par des transformations successives de modèles. Par ailleurs, un même produit peut avoir plusieurs représentations liées les unes aux autres. Cela se traduit par le fait

1. ADM est une approche proposée par l'OMG en 2001 en tant que vision d'OMG pour les approches Model Driven
2. http://combemale.perso.enseeiht.fr/teaching/metamodeling0708/IDM-Transfo_v1.1.pdf

qu'on peut avoir différentes vues et différents niveaux d'abstraction du même produit. Avoir une définition du langage de description (méta-modèle) de ces représentations est l'une des questions clés de ce type de technique. Par ailleurs, la génération des différents modèles est une autre question clé. Il s'agit de la transformation de modèles ayant pour objectif de rendre systématiques et opérationnelles ces différentes représentations du produit dans le processus de développement (génération des scénarios de test, génération de code, vérification/validation, ...).

Nous passons en revue dans cette section les concepts clés de l'ADM. Nous commençons par introduire dans la section 3.3.1 les notions de modèle et de méta-modélisation et leur utilisation dans l'approche ADM. Ensuite, nous présentons la transformation de modèles en Section 3.3.2.

3.3.1 Modèles et méta-modèles

Nombreuses sont les définitions données aux concepts clés de l'ADM. Certaines sont retenues et sont présentées dans la suite.

Définition 3.1 *[Minsky 1965] "M est un modèle du système S, si M aide un observateur O à répondre aux questions qu'il se pose sur S."*

Comme mentionné ci-dessus, les méthodologies de développement s'orientent de plus en plus de l'approche objet vers l'architecture dirigée par les modèles (ADM). Cette migration d'ingénierie (de l'objet vers le modèle) est accompagnée d'une évolution des concepts clés de l'ingénierie. En effet, avec l'approche objet où le principe est "tout est objet" on est passé vers l'ADM où le principe est "tout est modèle" [Bézivin 2004b, Bézivin 2005, Combemale 2008]. Cependant, l'idée clé de ADM est de fournir une représentation (modèle) pour chaque vue, c'est-à-dire que plusieurs modèles sont utilisés pour exprimer séparément les différentes vues des utilisateurs, des concepteurs, des développeurs, etc, et à différents niveaux d'abstraction. Cette séparation entre ces différents modèles est plus claire et plus franche [Combemale 2008] que les approches patterns [Gamma 1995] et les approches aspects [Kiczales 1997].

Le "modèle" qui est un concept clé de l'ADM ne possède pas une définition standard. Cependant de nombreux travaux [Bézivin 2001, Seidewitz 2003] ont proposé des définitions proches les unes des autres. Par la suite, nous considérons la définition suivante d'un modèle.

Définition 3.2 *[Blanc 2005] Un modèle est une représentation abstraite (abstraction pertinente) d'un système qui est capable de répondre aux questions qu'on se pose sur le système modélisé, exactement de la même façon que le système aurait répondu lui-même.*

À partir de cette définition, on déduit la première relation majeure de l'ADM qui relie le système modélisé à son modèle, appelé *"representation_De"* [Atkinson 2003, Seidewitz 2003, Bézivin 2004a, Combemale 2008]. Cette relation est représentée sur la figure 3.2. Néanmoins, pour qu'un modèle soit exploitable et manipulable par une machine, le langage utilisé pour exprimer le modèle doit être rigoureusement défini. Or vu que le principe de l'ADM est "tout est modèle", la définition du langage de modélisation est aussi considérée comme un modèle, appelé méta-modèle [Bézivin 2004b, Bézivin 2005, Combemale 2008].

Définition 3.3 *[Blanc 2005] Un méta-modèle est un modèle qui définit le langage d'expression d'un modèle [OMG 2004], c'est-à-dire le langage utilisé pour modéliser un modèle.*

Cette définition du concept méta-modèle définit un autre type de relation reliant le modèle et le langage utilisé pour le décrire. Cette nouvelle relation s'appelle *"conforme_A"*, elle est

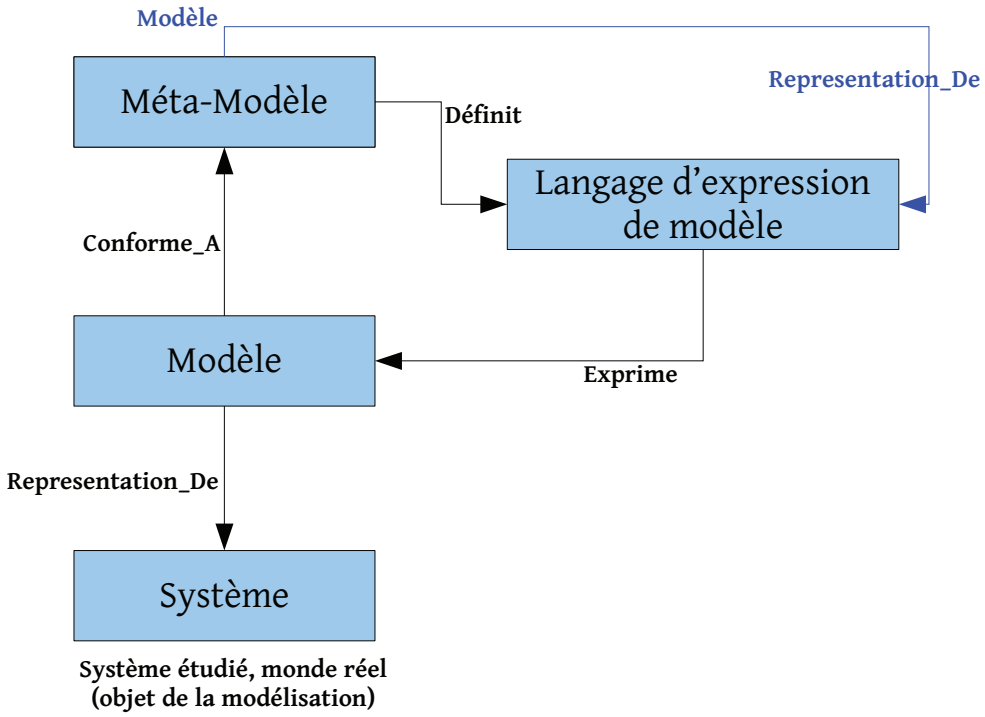

FIGURE 3.2: Concepts et Relations dans ADM

représentée sur la figure 3.2. Cependant, l'adoption du concept clé "méta-modèle" comme la définition du langage de description de modèles a donné lieu à plusieurs définitions et interprétations. Cela a poussé à donner une définition standard du concept de méta-modèle, par le biais de sa représentation appelée *méta-méta-modèle*.

Définition 3.4 *[Blanc 2005] Un méta-méta-modèle est un modèle qui définit d'une façon précise et claire un langage de méta-modélisation, c'est-à-dire qu'il décrit rigoureusement les éléments d'un langage de modélisation. Afin de limiter le nombre de niveaux d'abstraction, un méta-méta-modèle doit être auto-descriptif c'est-à-dire qu'il doit avoir la capacité de se décrire lui-même.*

Ces notions (*modèle*, *méta-modèle* et *méta-méta-modèle*) et leurs relations (*"representation_De"*, *"conforme_A"*) sont les concepts clés pour définir les standards de l'ADM. En effet, c'est en s'appuyant sur ces concepts et relations que l'Object Management Group (OMG) a défini :

1. L'ensemble de ses standards et en particulier le langage UML (Unified Modeling Language) [OMG 2011a, OMG 2011b],
2. Une organisation de modélisation à quatre couches (Figure 3.3).

Cette dernière est décrite sous une forme pyramidale dans la figure 3.3 [Bézivin 2003] dont l'architecture est décrite comme suit : à la base de la pyramide (niveau M0), on place le système modélisé, souvent appelé *monde réel*. Le niveau M1 est constitué de l'ensemble des modèles utilisés pour décrire le produit. Le niveau M2 est constitué de l'ensemble des méta-modèles utilisés pour la définition des modèles en M1. Dans cette couche M2, on trouve des notations comme l'UML. Finalement, le niveau M3 (sommet de la pyramide) représente le méta-méta-modèle utilisé pour décrire les méta-modèles. Le méta-méta-modèle présente la caractéristique

d'être auto-descriptif. Dans cette couche M3, on trouve des standards comme le Meta-Object Facility (MOF) [OMG 2004].

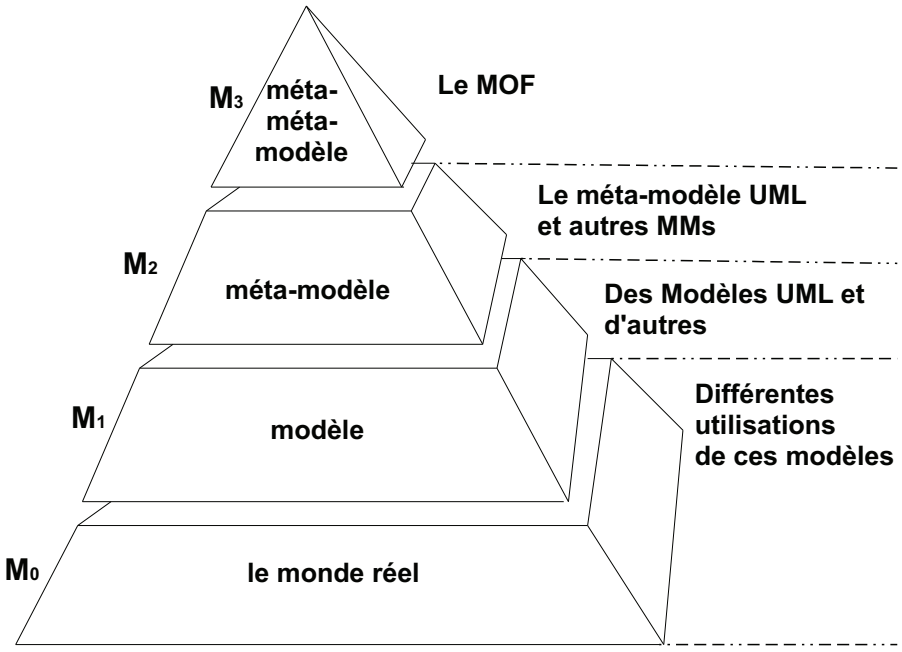

FIGURE 3.3: Pyramide de modélisation de l'OMG [Bézivin 2003]

Dans la section suivante (Section 3.3.2), nous abordons le deuxième concept clé de cette initiative qui concerne la "transformation de modèles".

3.3.2 Transformation de modèles

Nul ne peut nier les évolutions technologiques et la diversité de plateformes d'exécution de systèmes industriels ou informatiques. Afin d'avoir une robustesse et une indépendance par rapport aux technologies et par rapport aux plateformes, l'idée est de travailler à un niveau d'abstraction plus élevé. Par la suite, la migration ou bien l'évolution d'un espace technologique à un autre est assurée à l'aide des outils et/ou des langages de transformation.

3.3.2.1 Origines et objectifs

Le fait d'axer tout le cycle de développement sur les modèles tout en adoptant des niveaux d'abstraction différents a un intérêt double. En effet, cela permet de :

1. vérifier et valider suffisamment tôt certaines propriétés du produit final sur un niveau d'abstraction élevé,
2. fournir une base pour la conception et la validation des différents aspects du futur système. Parmi ces aspects on trouve la sécurité, la performance, etc.

Cela permet un gain considérable en termes de qualité, de coût et de temps. Par ailleurs, la variété des niveaux d'abstraction que l'ADM préconise, donne naissance à différents types de modèles comme :

- **CIM** (**C**omputational **I**ndependent **M**odels) : modèles utilisés pour décrire l'environnement du système ainsi que les propriétés et les exigences de cet environnement,
- **PIM** (**P**latform **I**ndependent **M**odels) : modèles utilisés pour décrire le savoir-faire métier sous forme de descriptions abstraites dont la principale caractéristique est qu'elles sont indépendantes des plates-formes et des technologies utilisées,
- **PSM** (**P**latform **S**pecific **M**odels) : modèles qui visent une plate-forme d'exécution spécifique. En effet, un PSM n'est que la projection d'un modèle PIM sur une plate-forme spécifique. Cela est très intéressant du point de vue pratique parce qu'il garantit la portabilité et l'évolution et augmente la productivité.

3.3.2.2 Idée et principe

La transformation de modèles (Figure 3.4) est la génération d'un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources [Bézivin 2004b]. Généralement, un algorithme de transformation est composé d'un ensemble de relations (souvent appelées règles de transformation) reliant les éléments (entités) des modèles sources avec celles des modèles cibles. Une règle est une correspondance entre un ou plusieurs éléments du modèle source avec un ou plusieurs éléments du modèle cible. Une règle de transformation est définie au niveau méta-modèle. En pratique le processus d'exécution d'une transformation de modèles est composé de deux étapes :

1. La première étape consiste à identifier les correspondances entre les éléments des modèles sources et cibles au niveau de leurs méta-modèles respectifs, et à définir les règles de transformation à appliquer. Cette étape génère une fonction de transformation qui sera appliquée à tous les éléments du modèle source.
2. La deuxième étape consiste à exécuter la fonction de transformation obtenue dans la première étape pour générer les modèles cibles. Cette étape de génération peut être réalisée automatiquement en utilisant un moteur de transformation.

Une problématique liée à l'activité de transformation concerne le langage utilisé pour exprimer les règles de transformation. Plusieurs types de langages sont utilisés pour décrire les transformations de modèles. Dans le paragraphe suivant nous passons en revue les différentes familles de langages de transformation en nous focalisant sur le standard OMG pour la transformation qui est le standard QVT [Blanc 2005, OMG 2005, Kurtev 2008].

3.3.2.3 Langages pour la transformation

Plusieurs types de langages peuvent être utilisés pour exprimer les transformations de modèles. Selon leurs types, ces langages peuvent être répertoriés en différentes familles : d'un côté les langages généralistes et de l'autre les langages dédiés à la transformation de modèles. D'autre part, dans [Blanc 2005] sont répertoriées les différentes approches de transformation : l'approche par programmation, l'approche par template et l'approche par modélisation.

- **L'approche par programmation** consiste à utiliser les langages de programmation en général, et plus particulièrement les langages orientés objet. Dans cette approche, la transformation est similaire à un programme informatique, mais elle a la particularité de manipuler des modèles. Vu que cette approche est souvent outillée, elle reste la plus utilisée. Comme langage de programmation concerné on trouve Java, C++, ...
- **L'approche par template** "*consiste à définir des canevas des modèles cibles souhaités*". En réalité, ces canevas sont des modèles cibles paramétrés ou des modèles templates. En pratique, l'exécution d'une transformation consiste à prendre un modèle template et à

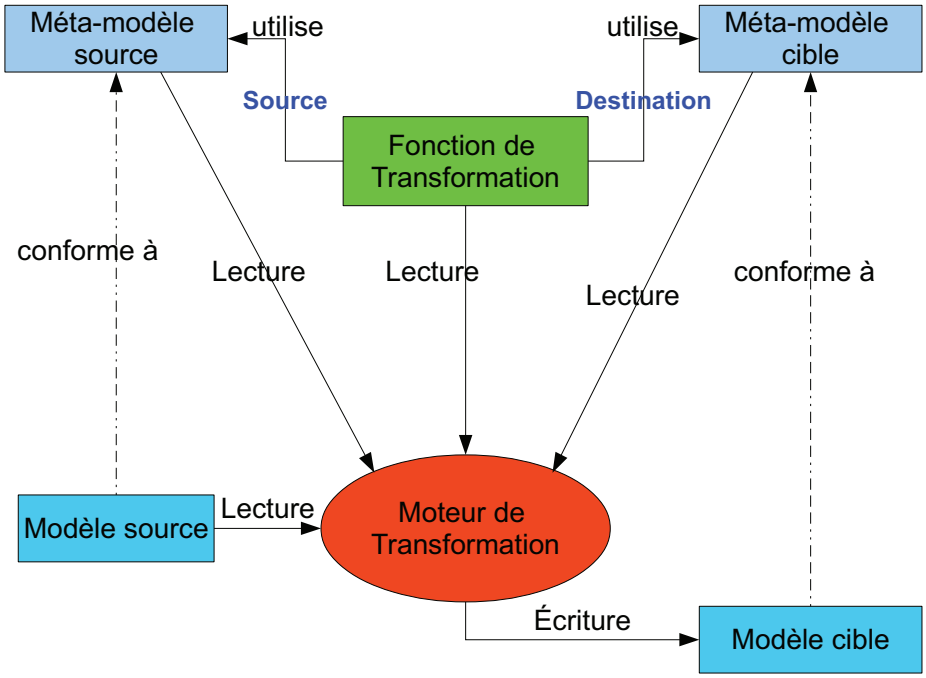

FIGURE 3.4: Schéma de base de la transformation de modèles [Blanc 2005, Staff 2008]

remplacer ses paramètres par les informations extraites des modèles sources. Comme support à ce deuxième type d'approches, on trouve des implémentations comme le "Softeam ADM Modeler".

- **L'approche par modélisation** "*consiste à appliquer les concepts de l'ingénierie des modèles aux transformations de modèles elles-mêmes. L'objectif est de décrire les transformations de modèles par des modèles de transformation pérennes et productifs, et d'exprimer leur indépendance vis-à-vis des plates-formes d'exécution*". Le standard MOF 2.0 QVT (Query, View, Transformation) [OMG 2005] de l'OMG a été élaboré dans ce cadre et a pour but de définir un langage standard pour la transformation de modèles. Dans cette classe de langages on trouve ATL (ATLAS Transformation Language) [Jouault 2005], QVT, ...

Néanmoins, ces langages permettent de décrire des correspondances structurelles entre modèles sources et modèles cibles, c'est-à-dire, qu'ils ne permettent que de définir et exprimer des transformations de structures et de syntaxes. Une composante aussi importante que la syntaxe est ignorée par la transformation ; c'est la préservation de la sémantique. En effet, chaque modèle est relié à deux éléments importants : sa syntaxe et sa sémantique. La syntaxe d'un modèle est une notation (graphique et/ou textuelle) qui décrit formellement un modèle. Tandis que la sémantique, en liaison forte avec la syntaxe, décrit d'une façon claire et précise le comportement du modèle défini. La préservation de la sémantique (comportement) à travers la transformation permet de garantir que le modèle généré garde bien les propriétés que nous cherchons à préserver. En d'autres termes, il s'agit d'une forme de validation de la transformation.

Cependant, il n'existe pas de standards pouvant être utilisés pour normaliser l'étape de validation de la transformation. Par conséquent, plusieurs approches ont été proposées dans la

littérature. Dans la section 3.3.3, nous passons en revue les approches les plus utilisées et les plus citées dans le domaine de la transformation de modèles.

3.3.3 Validation et ingénierie des modèles

Dans la littérature, on trouve deux niveaux de validation de transformations : un premier type de validation qui s'intéresse à la validation au niveau modèle généré (section 3.3.3.1), et un deuxième type de validation qui se focalise sur la transformation elle-même (section 3.3.3.2).

3.3.3.1 Validation des modèles

Ce type de validation présente la caractéristique d'être direct (sur le modèle généré) et ainsi à un niveau bas (modèle ou code). Dans la littérature on trouve deux approches pour la validation des modèles, à savoir la validation directe des modèles ainsi que la validation du code obtenu [Fleurey 2006]. En effet, la validation directe des modèles consiste à vérifier ou à tester les modèles. Cette approche présente un inconvénient majeur vu qu'elle exige la disposition d'une plate-forme de simulation ou d'exécution des modèles. La deuxième approche consiste à tester le code généré à partir des modèles plutôt que de tester les modèles directement. Cette approche n'exige pas la disposition d'une plate-forme de simulation ou d'exécution des modèles, mais nécessite néanmoins une étape de génération de code. Il est à noter que le processus de génération de code présente une source potentielle d'erreurs, ce qui complique énormément la tâche de validation.

3.3.3.2 Validation de la transformation de modèles

Vu que l'ADM propose un processus de développement centré sur les modèles, l'idée est de garder cette optique même pour la validation de la phase de transformation. En effet, le principe de l'ADM est que "tout est modèle" et par conséquent la transformation est considérée elle-même comme un modèle. Cela est très intéressant d'un point de vue pratique puisque la validation a lieu à un niveau d'abstraction supérieur à celui utilisé dans l'approche "validation des modèles". Dans cette approche on trouve principalement trois classes de validation, à savoir le test (cf. section 3.3.3.2.1), la vérification/préservation des propriétés liées à la transformation comme la terminaison (cf. section 3.3.3.2.2) et finalement la bisimulation (cf. section 3.3.3.2.3).

3.3.3.2.1 Test- L'idée de cette technique est de définir un ensemble de modèles à fournir à la transformation à tester. Ensuite, pour chaque modèle ou ensemble de modèles définis, la transformation est exécutée. Puis, les modèles générés sont comparés aux modèles attendus [Steel 2004]. Cela permet de détecter les erreurs de définition des correspondances.

Plusieurs travaux adoptent cette technique de validation de la transformation. Dans [Steel 2004] les auteurs présentent un retour d'expérience de l'utilisation du test pour la validation de la transformation. En particulier, les auteurs présentent les différents problèmes liés à l'utilisation du test pour valider une transformation. La première difficulté rencontrée concerne la complexité de manipulation des modèles étant donné que cette tâche exige la disposition d'outils d'édition ainsi que de sérialisation des modèles. La deuxième difficulté identifiée réside dans les critères de sélection des jeux de test. D'autres travaux ont été proposés comme [Küster 2006b] où les auteurs proposent une approche boîte blanche (test structurel). En effet, un langage de template qui repose sur la structure des règles de transformation a été proposé afin de générer automatiquement des modèles de test. Cependant, cette approche présente une limitation majeure puisqu'elle est fortement dépendante du langage de l'implémentation.

D'autres travaux ont proposé des mécanismes de mise en œuvre de la technique de test [Sen 2007, Sen 2008, Ehrig 2009, Fleurey 2009, Mottu 2010]. Cependant, tous ces travaux sont d'accord sur le fait que cette technique ne garantit pas la validation de la transformation puisqu'il est quasi impossible de garantir l'exhaustivité des jeux de test.

3.3.3.2.2 Préservation des propriétés- Une deuxième façon pour la validation de la transformation de modèles est d'utiliser une approche formelle. Il s'agit concrètement d'identifier et de vérifier certaines propriétés sur la transformation [Küster 2003, Küster 2004, Küster 2006a]. Ainsi, l'ensemble des propriétés définies forme la liste des caractéristiques que la transformation doit préserver. Abstraction faite des éventuelles autres propriétés qui n'auront pas été définies explicitement. En effet, l'approche repose sur une spécification formelle des règles de transformation. Ensuite, ces règles de transformation sont utilisées pour vérifier l'ensemble des propriétés (la correction syntaxique des règles, la convergence de la transformation, la terminaison de la transformation). D'autres approches formelles ont été proposées pour aborder la vérification de la transformation tout en vérifiant formellement certaines propriétés spécifiques [Varró 2003, Giese 2006, Poernomo 2008].

3.3.3.2.3 Bisimulation- Une troisième technique de vérification de la transformation de modèles consiste à valider la préservation du comportement entre le modèle source et le modèle généré. Cette relation d'équivalence comportementale définie entre les modèles est appelée bisimularité [Seidner 2009, Peres 2011]. L'idée de cette technique est comme suit : à partir de l'algorithme de transformation, définir une relation d'équivalence entre le modèle source et le modèle cible, ensuite montrer que cette relation est une bisimulation.

Dans la mesure où les modèles que nous traitons en chapitre 4 possèdent des sémantiques en systèmes de transitions, nous consacrerons la section suivante à la présentation des systèmes de transition, ainsi que des relations d'équivalence les concernant.

3.4 Système de transition et relations d'équivalence

Dans cette section, nous allons commencer par définir un système de transitions (ST) avec ses variantes *étiquetée* et *temporisé* (cf. section 3.4.1). Ensuite nous passons en revue différentes relations d'équivalence (cf. section 3.4.2) qu'on peut avoir entre deux systèmes de transitions.

3.4.1 Système de transitions

3.4.1.1 Système de transitions étiquetées

Le modèle états-transitions (RdP, AEF, etc) ont des sémantiques qui s'expriment généralement par un système de transitions étiquetées (STE) (en anglais Labelled Transition System (LTS)).

Définition 3.5 *Un système de transitions étiquetées (STE) [Cassez 2003, Seidner 2009] est un quadruplet* $STE = (S, S_0, \Sigma, \longrightarrow)$ *où :*

- S *est un ensemble d'états,*
- $S_0 \subseteq S$ *est l'ensemble des états initiaux,*
- Σ *est l'alphabet d'actions (l'ensemble des étiquettes) de transitions,*
- $\longrightarrow \subseteq S \times \Sigma \times S$ *est la relation de transition. Dans la suite nous utilisons la notation "*$q \xrightarrow{a} q'$*" à la place de "*$(q, a, q') \in \longrightarrow$*".*

Définition 3.6 *Une exécution est une séquence (finie et/ou infinie) de transitions, notée* $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$.

Définition 3.7 *La trace d'une exécution* $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots$ *correspond à la séquence des actions de cette exécution (la suite des étiquettes* a_i*), notée* $[a_1 a_2 \ldots]$.

En d'autres termes, une trace est un mot formé par concaténation des symboles de l'alphabet Σ. Un mot quelconque "m" est accepté par un STE si et seulement s'il existe une exécution à partir d'un état initial de STE dont la trace est "m".

Définition 3.8 *Le langage accepté par un système de transition* $\mathcal{S}$*, noté* $\mathscr{L}(\mathcal{S})$*, est l'ensemble des mots acceptés par* $\mathcal{S}$.

Exemple 3.1 *Soit le système de transitions* $\mathcal{S}$ *représenté en figure 3.5.* $\mathcal{S}$ *est défini par :*
- $S = \{p, q, r, s\}$
- $S_0 = \{p\}$
- $\Sigma = \{\alpha, \beta, \lambda\}$
- $\rightarrow = \{(p, \alpha, q); (q, \beta, r); (q, \lambda, s)\}$

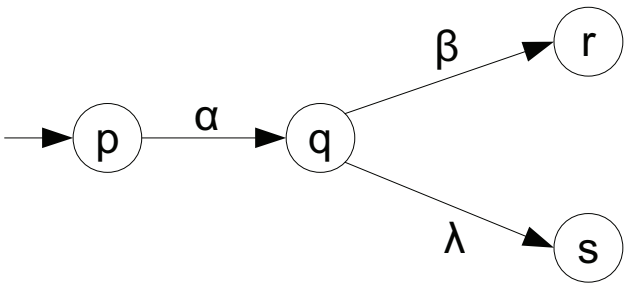

FIGURE 3.5: Un exemple de système de transitions étiquetées (STE)

L'ensemble des exécutions possibles est $\{(p \xrightarrow{\alpha} q); (p \xrightarrow{\alpha} q \xrightarrow{\beta} r); (p \xrightarrow{\alpha} q \xrightarrow{\lambda} s)\}$ *et le langage accepté* $\mathscr{L}(\mathcal{S}) = \{[\alpha]; [\alpha\beta]; [\alpha\lambda]\}$.

3.4.1.2 Système de transitions temporisé

Une extension de STE permettant de prendre en compte le temps explicitement est le système de transitions temporisé (STT) (en anglais, (Timed Transition Systems (TTS))). Deux types de transitions sont définis et utilisés dans un STT :
- les transitions d'action (ou discrètes) : ces transitions sont aussi utilisées dans les STE et permettent de représenter les évolutions discrètes du système.
- les transitions continues (ou temporelles) : ces transitions sont utilisées pour modéliser l'écoulement du temps.

Définition 3.9 *Un système de transitions temporisé (STT) [Cassez 2003, Seidner 2009] est un quadruplet* $STT = (S, S_0, \Sigma, \longrightarrow)$ *où :*
- S *est un ensemble d'états,*
- $S_0 \subseteq S$ *est l'ensemble des états initiaux,*
- Σ *est l'alphabet d'actions (l'ensemble des étiquettes) de transitions,*
- $\longrightarrow \subseteq S \times \{\Sigma \cup \mathbb{R}_{\geq 0}\} \times S$ *est la relation de transition formée par :*
  - *la transition discrète :* $\xrightarrow{a \epsilon \Sigma} \subseteq S \times \Sigma \times S$

– *la transition continue :* $\overset{\delta\epsilon\mathbb{R}_{\geq 0}}{\longrightarrow} \subseteq S \times \mathbb{R}_{\geq 0} \times S$

Définition 3.10 *Une exécution d'un STT est une séquence (finie et/ou infinie) de transitions discrètes et continues, notée* $q_0 \xrightarrow{\delta_1} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{\delta_2} q'_1 \xrightarrow{a_2} \ldots$.

Définition 3.11 *La trace (temporisée) d'une exécution* $q_0 \xrightarrow{\delta_1} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{\delta_2} q'_1 \xrightarrow{a_2} \ldots$ *correspond à la séquence des couples* (δ_i,a_i)*, notée* $[(\delta_1,a_1),(\delta_2,a_2)\ldots]$.

De la même façon que pour le cas du STE, une trace est un mot formé par concaténation de symboles de l'alphabet $\Sigma \times \mathbb{R}_{\geq 0}$. Un mot quelconque "m" est accepté par un STT si et seulement s'il existe une exécution à partir d'un état initial du STT et dont la trace est "m".

Définition 3.12 *Le langage accepté par un STT* $\mathcal{S}$*, noté* $\mathscr{L}_T(\mathcal{S})$*, est l'ensemble des mots acceptés par* $\mathcal{S}$.

3.4.2 Relations d'équivalence

Dans cette section, nous passons en revue les principales relations qu'on peut définir entre deux systèmes de transitions temporisés. En effet, en se basant sur le niveau d'équivalence entre deux systèmes de transitions, on peut définir quatre types de relations [Drissi 2000, Seidner 2009]. Ces quatre relations peuvent être répertoriées en allant de la plus faible relation d'équivalence qui est **l'égalité des langages** à la plus forte qui est **l'isomorphisme** tout en passant par des relations comme **la simulation** et **la bisimulation**.

3.4.2.1 Égalité de langages

Une première relation qui permet d'établir un premier niveau d'équivalence est la comparaison des traces ou, plus généralement, des langages acceptés par les deux systèmes.

Définition 3.13 *Soient* $\mathcal{S} = (S, S_0, \Sigma_S, \longrightarrow_S)$ *et* $\mathcal{Q} = (Q, Q_0, \Sigma_Q, \longrightarrow_Q)$ *deux STTs.* $\mathcal{S}$ *et* $\mathcal{Q}$ *sont équivalents en termes de langage temporisé accepté, noté* $\mathcal{S} \equiv_{trace} \mathcal{Q}$*, si et seulement si* $\mathscr{L}_T(\mathcal{S}) = \mathscr{L}_T(\mathcal{Q})$.

Exemple 3.2 *Les deux systèmes de transitions représentés en figure 3.6 acceptent le même langage temporisé* $\mathscr{L} = \{(0;\alpha);(0;\alpha)(1,75;\beta);(0;\alpha)(0;\lambda)\}$

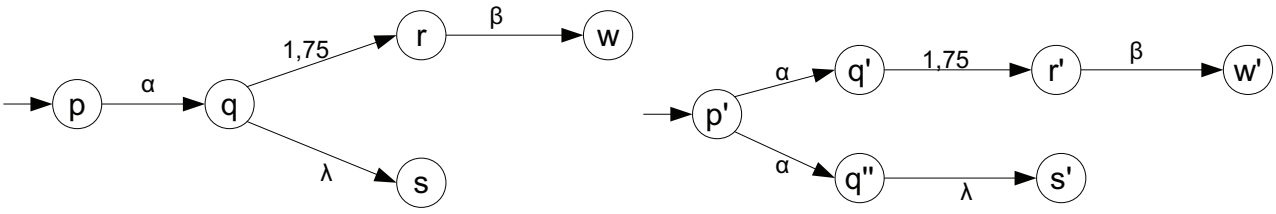

FIGURE 3.6: Équivalence de langages

Cependant, l'égalité des langages ne garantit pas l'équivalence de comportement respectif des systèmes comparés. Afin de pallier cette limite, une autre relation a été développée, la relation de simulation. En effet, la simulation est plus forte que l'égalité de langages et permet de comparer le comportement de deux systèmes.

3.4.2.2 Simulation (temporelle)

Définition 3.14 *Soient* $\mathcal{S} = (S, S_0, \Sigma_S, \longrightarrow_S)$ *et* $\mathcal{Q} = (Q, Q_0, \Sigma_Q, \longrightarrow_Q)$ *deux STTs et* $\lesssim_{st}$ *est une relation binaire sur* $\mathcal{S}$ *et* $\mathcal{Q}$. $\lesssim_{st}$ *est une simulation temporelle de* $\mathcal{S}$ *par* $\mathcal{Q}$, *noté* $\mathcal{S} \lesssim_{st} \mathcal{Q}$, *ssi :*

- *pour tout* $s_0 \in S_0$, *il existe* $q_0 \in Q_0$ *tel que* $s_0 \lesssim_{st} q_0$;
- *pour tout* $(s, q) \in S \times Q$ *tel que* $s \lesssim_{st} q$, *s'il existe* $s' \in S$ *et* $\delta \in \mathbb{R}_{\geq 0}$ *tels que* $s \xrightarrow{\delta}_S s'$, *alors il existe* $q' \in Q$ *tel que* $q \xrightarrow{\delta}_Q q'$ *et* $s' \lesssim_{st} q'$;
- *pour tout* $(s, q) \in S \times Q$ *tel que* $s \lesssim_{st} q$, *s'il existe* $s' \in S$ *et* $a \in \Sigma_S$ *tels que* $s \xrightarrow{a}_S s'$, *alors il existe* $q' \in Q$ *tel que* $q \xrightarrow{a}_Q q'$ *et* $s' \lesssim_{st} q'$;

Exemple 3.3 *Dans la figure 3.6, le système de gauche simule le système de droite mais l'inverse n'est pas vrai. Cependant, dans la figure 3.8, on peut vérifier que le premier système simule le deuxième et réciproquement.*

FIGURE 3.7: Simulation

3.4.2.3 Bisimulation (temporelle)

A partir de la définition de la relation de simulation temporelle découle une autre relation plus forte qui est la bisimulation temporelle. En effet, la bisimulation exige que toute action discrète et/ou continue, visible dans l'un des systèmes en relation doit être simulée par l'autre système et vice-versa.

Définition 3.15 *Soient* $\mathcal{S} = (S, S_0, \Sigma_S, \longrightarrow_S)$ *et* $\mathcal{Q} = (Q, Q_0, \Sigma_Q, \longrightarrow_Q)$ *deux STTs et* $\approx$ *une relation binaire sur* $\mathcal{S}$ *et* $\mathcal{Q}$. $\mathcal{S}$ *et* $\mathcal{Q}$ *sont bisimulaires, noté* $\mathcal{S} \approx \mathcal{Q}$, *ssi* $\approx$ *est une simulation temporelle de* $\mathcal{S}$ *par* $\mathcal{Q}$ *et* $\approx^{-1}$ *est une simulation temporelle de* $\mathcal{Q}$ *par* $\mathcal{S}$.

Exemple 3.4 *Dans la figure 3.7, bien que chacun des deux systèmes simule l'autre, leurs états ne sont pas en bijection*[3] *: ces deux systèmes ne sont pas en bisimulation.*

Remarque 3.1 *Deux systèmes en bisimulation temporelle acceptent le même langage temporisé.*

3.4.2.4 Isomorphisme

L'isomorphisme est la relation d'équivalence la plus forte entre deux systèmes de transitions. En effet, l'isomorphisme est une relation plus restrictive que les relations définies ci-dessus. Cette relation est définie comme une égalité[4] entre deux systèmes de transitions [Cheikh 2009].

3. Une bijection est une relation entre deux ensembles qui relie chaque élément de l'ensemble de départ à un élément unique dans l'ensemble d'arrivé et vice-versa.

4. L'égalité dans ce cas là est modulo une action de renommage des états d'un des systèmes par les états de l'autre système. C'est une égalité de structure.

FIGURE 3.8: Isomorphisme

3.5 Conclusions

Dans ce chapitre, nous nous sommes intéressés à l'activité de modélisation et les concepts liés. En particulier, nous avons présenté des notions relatives à la modélisation du comportement des systèmes. Nous avons ainsi préparé la présentation de notre contribution sur le volet modélisation, et qui sera détaillée au chapitre suivant. L'idée étant de partir d'un modèle de comportement assez-intuitif, ici des states machines d'UML avec des annotations temporelles, et d'en générer automatiquement des modèles formels, ici des automates temporisés, qui se prêtent à la vérification formelle. L'objectif derrière est de faire profiter l'utilisateur des avantages des SM (simplicité, flexibilité, etc.) tout en lui évitant les étapes ardues pour l'obtention d'un modèle formel.

Chapitre 4

Transformation des modèles : State Machine vers automate temporisé

Résumé : *Ce chapitre est consacré à notre contribution concernant la phase de modélisation du comportement. Dans ce chapitre, nous introduisons une technique d'assistance relative à cette phase de modélisation. Notre contribution repose sur l'idée de l'utilisation des modèles au cœur du cycle de développement dont le but d'obtenir des représentations de différents aspects, y compris les aspects temporels, du système à valider. Ici, l'idée est de permettre à l'utilisateur de manipuler une notation graphique et intuitive, le diagramme SM d'UML avec des annotations temporelles, et d'en générer automatiquement des spécifications formelles en automates temporisés. Concrètement, la mise en œuvre de la transformation de modèles nécessite de définir un certains nombre de concepts clés : la syntaxe et la sémantique respectives des modèles utilisés, l'algorithme de transformation ainsi que sa validation. Tous ces éléments sont progressivement présentés dans ce chapitre.*

Sommaire

4.1 Introduction

Dans le cadre de ce travail, nous avons choisi d'utiliser les state-machines (SM) d'UML comme langage durant la phase de modélisation du comportement. En effet, les SM sont un formalisme graphique qui permet à l'utilisateur de produire des modèles tout en manipulant (facilement) des objets graphiques et textuels. Les modèles obtenus seront ensuite transformés automatiquement afin d'obtenir des modèles formels (en automates temporisés). Dans ce chapitre, nous développons un algorithme de transformation qui prend en entrée des modèles SM avec des annotations temporelles, pour générer des modèles d'automates temporisés [Mekki 2010c].

Dans la suite, nous passons en revue les concepts clés de cette étape de transformation de modèles. D'abord, nous commençons par introduire dans la section 4.2 la problématique et les objectifs attendus. Nous détaillons ensuite les deux formalismes clés que sont les SM d'UML (section 4.3) et les automates temporisés (section 4.4). Pour chaque formalisme, une description du langage utilisé ainsi que sa sémantique sont présentées. L'algorithme de transformation de modèles est donné en section 4.5. La dernière étape concerne la démonstration d'équivalence comportementale entre le modèle source et le modèle généré. Ici, nous démontrons une relation de bisimulation entre ces deux modèles ce qui garantit la préservation des principales propriétés comportementales à travers notre transformation. La démonstration d'équivalence fera l'objet de la section 4.6.

4.2 Problématique et objectif

La vérification fait souvent appel à des modèles qui décrivent le comportement du système conçu. Or, comme nous l'avons expliqué, l'utilisation des langages formels permet d'avoir des spécifications précises qui sont sous certaines conditions, vérifiables automatiquement. Néanmoins, cette utilisation reste une source d'erreurs, étant donné que l'utilisateur doit souvent manipuler des notations spécifiques avec des concepts abstraits dans le but de décrire le comportement. Les SM d'UML offrent un bon compromis simplicité/précision/capacité d'expression quand on doit décrire le comportement des systèmes. En effet, grâce à leur notation graphique, l'interprétation des diagrammes SM reste assez intuitive. Par ailleurs, grâce aux différents éléments qu'ils intègrent (gardes, actions, etc), les SM offrent une certaine flexibilité en termes de modélisation. De l'autre côté, même en l'absence d'une sémantique clairement définie pour les SM, ces diagrammes restent relativement précis du fait de la similitude entre leur syntaxe graphique et celle de modèles formels de type état-transitions comme les automates à états finis. Néanmoins, on n'atteint pas un niveau de rigueur suffisant afin de dérouler des algorithmes de vérification directement sur les modèles UML, même si de récents travaux s'intéressent à l'utilisation de techniques formelles sur ces modèles [Boufenara 2010, Seidewitz 2011]. L'idée est donc de donner à l'utilisateur la possibilité de décrire le comportement du système par des SM avec des annotations temporelles pré-définies, et de nous charger de leur transformation en des modèles formels, ici des automates temporisés sur lesquels nous pouvons faire la vérification proprement dite. Afin de profiter de la flexibilité d'expression des SM et dans le but de cadrer l'utilisateur lors de la phase de modélisation, nous avons défini une liste d'annotations temporelles en langage naturel. Ces annotations seront utilisées pour exprimer des contraintes temporelles sur les transitions et aussi sur les états.

Exemple 4.1 *En se basant sur un ensemble riche de concepts comme les états hiérarchiques, orthogonaux ou séquentiels, les SM permettent de produire des représentations qui sont beaucoup*

plus compactes et intuitives que celles obtenues en utilisant, par exemple, les automates à états finis (AEF). Les figures 4.1 et 4.2 illustrent cette idée en donnant un modèle SM en figure 4.1 et son équivalent AEF en figure 4.2.

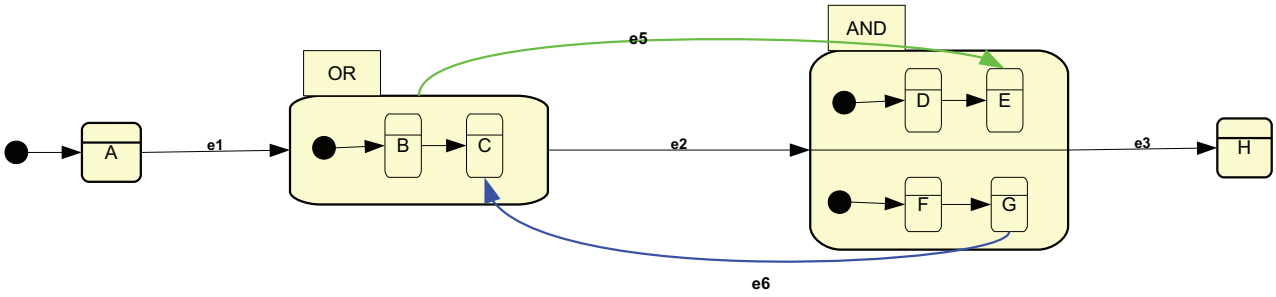

FIGURE 4.1: Modèle en SM

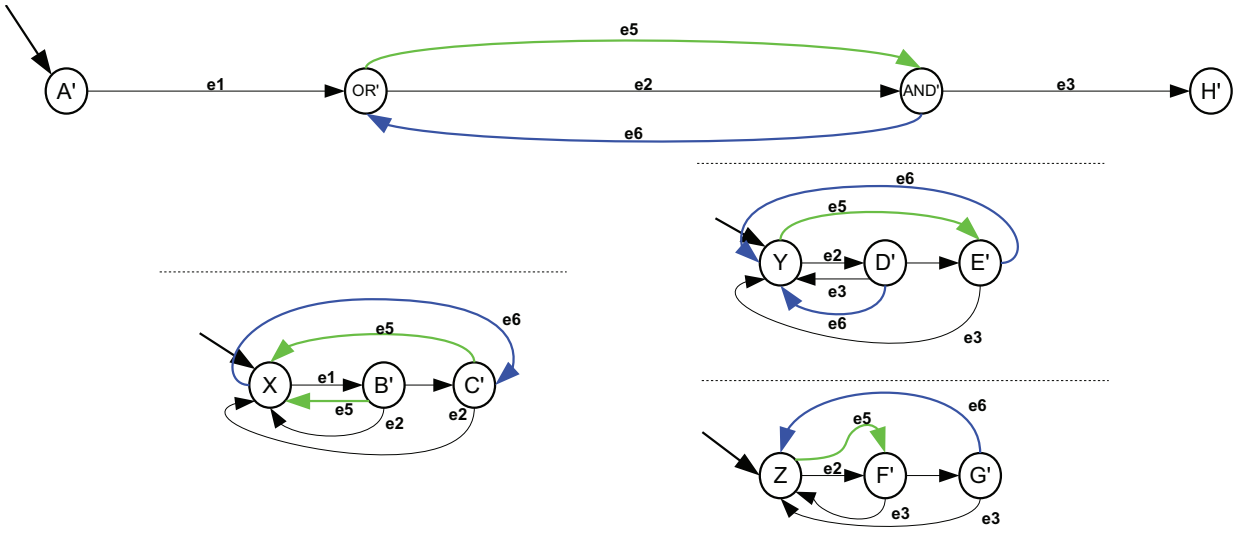

FIGURE 4.2: Modèle en AEF

Remarque 4.1 *On peut se demander ici pourquoi ne pas utiliser les automates hiérarchiques au lieu des automates (temporisés) pour avoir aussi le même avantage qu'avec les SM (modèles compacts) ? La raison est que les automates temporisés sont une notation bien formalisée (syntaxe et sémantique formellement définies) avec des outils très performants de simulation et de vérification. Ce qui n'est pas le cas pour l'automate hiérarchique [Mikk 1997] malgré le considérable effort qui a été mené dont l'objectif est la formalisation et l'outillage de cette extension [Beyer 1998, Lanotte 2000, Beyer 2001, David 2001, Lanotte 2001, Beyer 2003, David 2003, Lanotte 2006, Lanotte 2008]. [Waez 2011] propose un tour d'horizon très récent (14 aout 2011) des extensions proposées d'automate temporisé avec leur utilisation. Dans ce rapport,on peut noter un usage assez restreint des automates hiérarchiques.*

Néanmoins, la définition d'une transformation de modèles exige que la syntaxe ainsi que la sémantique des modèles source et cible soient définie d'une manière précise et claire, autrement dit de manière formelle. Ainsi, la syntaxe et la sémantique de deux formalismes, à savoir les SM et les automates, nécessitent d'être données avant d'aborder la transformation.

4.3 State machine d'UML : définition et sémantique

UML est un langage de modélisation graphique constitué d'un ensemble de diagrammes permettant de représenter différentes vues (aspects) du système : aspect statique, aspect dynamique et aspect fonctionnel. Dans le cadre de notre étude, nous nous intéressons à l'aspect dynamique et plus précisément au diagramme SM d'UML. Ce choix est justifié par le fait que nous considérons un point de vue discret (états-transitions) sur le comportement du système. Rappelons ici que notre objectif ultime est de vérifier des exigences temporelles sur le comportement du système. Dans cette partie, nous introduisons certains éléments de la syntaxe utilisée par les SM en section 4.3.1. Ensuite, nous proposons une syntaxe abstraite utilisant une représentation arborescente en section 4.3.2 ainsi qu'une sémantique formelle en section 4.3.3 pour les SM.

4.3.1 Diagramme SM : notations

Remarque 4.2 *Certaines définitions données dans la suite sont inspirées d'un ensemble de supports académiques sur UML disponibles en ligne*[1].

UML (***Unified Modeling Language***) est un formalisme graphique issu de la fusion de plusieurs langages de modélisation : OMT (Object Modeling Technique ; créé par Jim Rumbaugh), BOOCH (créé par Grady Booch) et OOSE (Object Oriented Software Engineering) et qui a été standardisé par l'OMG[2] en 1997. L'avantage d'UML est qu'il est un formalisme graphique et polyvalent. En effet, l'UML 2.4 définit 14 diagrammes afin de représenter les différentes vues d'un système parmi lesquelles nous nous focalisons sur le diagramme SM. Ce dernier décrit l'enchainement des états dans lesquels un objet peut se trouver en réponse à un ensemble de stimuli (aussi appelés événements). Les diagrammes SM ont été initialement développés par David Harel (Statecharts) [Harel 1987], puis repris par l'OMG afin de donner naissance aux SM d'UML [OMG 2011b, Rumbaugh 2004].

Définition 4.1 *Le diagramme SM est un automate à états finis décrivant l'ensemble des états que peut prendre une instance quelconque d'une classe en réponse à des stimuli.*

Or, un objet (une instance) est caractérisé(e) par un ensemble d'attributs qui changent de valeur au cours du temps. Ces valeurs forment l'état de l'objet.

Définition 4.2 *Un état représente une conjonction instantanée des valeurs des attributs d'un objet. Il se caractérise par sa durée et sa stabilité. Un état est donc donné par l'ensemble des valeurs des attributs de l'objet à un instant t.*

Par conséquent, un objet possède autant d'états que de combinaisons des valeurs possibles de ses attributs. Le nombre d'états peut être infini dans le cas où le domaine de certains attributs n'est pas fini. En UML, à un état peuvent être associés un ensemble d'actions, une activité et un invariant.

Définition 4.3 *Une action d'état est un traitement associé à un état. Le déclenchement d'une action d'état est lié soit à l'entrée ou à la sortie de l'état ou à l'apparition d'un événement.*

1. http://www.freewebs.com/fresma/Formation/UML/UML-08-DiagrammeEtatsTransitions.pdf et http://www.info.univ-tours.fr/~antoine/documents_enseignement/UML_GL_CM_UML.PDF
2. Object Management Group : http://www.omg.org/

Définition 4.4 *Une activité d'état est une séquence d'actions qui s'exécutent tant que l'objet est dans cet état.*

Dans les SM d'UML, on distingue trois types d'actions et une activité :

- Entry Action : chaque fois que l'objet entre dans l'état, l'action est exécutée,
- Exit Action : chaque fois que l'objet quitte l'état, l'action est exécutée,
- OnEvent Action : quand l'objet se trouve dans l'état, chaque fois que l'événement cité survient, l'action est exécutée,
- Do Activity : action exécutée en boucle dans l'état tant que ce dernier est actif.

Les états dans un SM peuvent être de trois types : état simple, état séquentiel et état concurrent (ou orthogonal). L'état simple, représenté à l'aide d'un rectangle à coins arrondis, est utilisé pour représenter une situation élémentaire dans laquelle l'objet peut se trouver (figure 4.3). Les deux autres types d'états (séquentiel et orthogonal) sont appelés "états composites". Dans un état séquentiel, noté OR (figure 4.4) et à un instant donné, un objet ne peut se trouver que dans un seul sous-état de l'état séquentiel. L'état séquentiel contient une seule **région**. L'état orthogonal, nommé AND (figure 4.5), permet de décrire deux ou plusieurs sous-états concurrents au sein d'un même état. Dans un état orthogonal et à un instant donné, un objet se trouve dans un seul sous-état direct dans chacun de ses états séquentiels (régions).

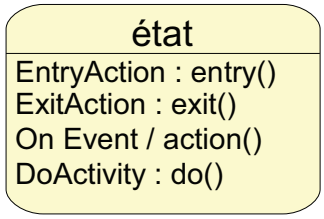

FIGURE 4.3: État Simple

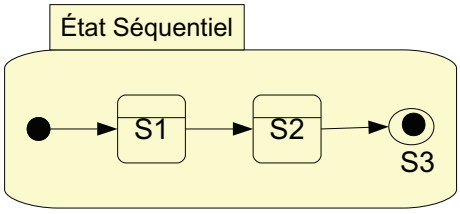

FIGURE 4.4: État Séquentiel

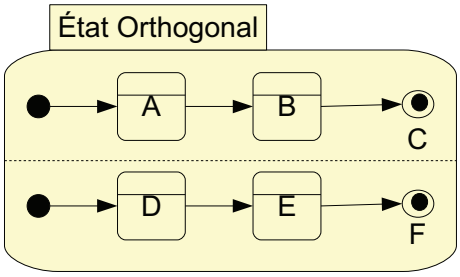

FIGURE 4.5: État Orthogonal

Définition 4.5 *Un "état composite" est un état composé par une ou plusieurs régions contenant chacune un ou plusieurs sous-états. Chaque région représente un flot d'exécution et peut contenir un état initial et final. D'un point de vue sémantique, toute transition dont la cible est un état composite est équivalente à une transition qui pointe les états initiaux de toutes les régions de celui-ci. Graphiquement, les différentes régions d'un état composite sont séparées par un trait horizontal en pointillé. Par conséquent si on reprend la définition d'un état simple, ce dernier peut être défini comme suit : un état simple est un état sans structure interne (autrement dit, composé d'un ensemble vide de sous-états).*

D'autres types d'états (pseudo-états) sont définis dans les SM comme, par exemple, l'état initial (•) qui indique le nœud par défaut dans une région, l'état final (⊙) qui indique la fin du cycle de l'instance en cours, le pseudo-état choix (◇) qui indique différents branchements possibles qu'on peut avoir selon des conditions prédéfinies. Les états et pseudo-états sont reliés par des transitions. Par définition, une transition (figure 4.6) est utilisée pour exprimer le passage instantané d'un état vers un autre. Une transition est déclenchée par un événement.

FIGURE 4.6: Transition UML

Définition 4.6 *Un événement est un stimulus dont l'occurrence est susceptible d'entraîner le déclenchement d'une réaction au sein du système modélisé.*

Deux types d'événements sont définis dans les SM : les événements externes échangés entre objets et les événements internes créés, émis et reçus au sein du même objet. Ces événements sont répertoriés en cinq classes :

1. *CallEvent* : indique la réception d'un message synchrone (un appel d'opération). Il résulte de l'exécution de l'opération ainsi que d'un changement d'état (ou configuration).
2. *AnyReceiveEvent* : déclenche la transition à la réception de tout message reçu.
3. *TimeEvent* : relatifs aux événements temporisés associés aux annotations *after* et *when*.
4. *ChangeEvent* : associé à une expression booléenne qui est continuellement évaluée.
5. *SignalEvent* : indique la réception d'un message asynchrone. Il résulte de l'exécution de l'opération ainsi que d'un changement d'état (ou configuration).

Définition 4.7 *Une transition est un arc orienté entre deux états conditionné par le déclenchement de l'événement auquel elle est associée. Le déclenchement d'une transition provoque le passage instantané d'un état à un autre.*

En plus des événements, il est aussi possible d'étiqueter une transition à l'aide de gardes et/ou d'actions.

Définition 4.8 *Une garde est une expression booléenne associée à une transition qui doit être vérifiée afin que celle-ci puisse être franchie.*

Définition 4.9 *Une action est un traitement instantané à exécuter lorsque la transition est franchie.*

Dans la suite de cette section et afin de simplifier la description de diagrammes SM, nous proposons d'utiliser une arborescence bipartie pour les décrire.

4.3.2 State machine sous forme d'une arborescence bipartie

Afin de décrire la sémantique des SM de manière claire et précise, nous proposons de les exprimer sous forme d'une arborescence bipartie. En pratique, cette représentation permet de définir une syntaxe abstraite qui pourra être utilisée par la suite lors de la définition de la transformation. Concrètement, chaque diagramme SM est ainsi associé à une arborescence contenant deux types de nœuds (un premier représente les régions SM et un deuxième représente les nœuds SM) et un ensemble de transitions pour expliciter la relation entre une région et ses sous-états. Le nœud racine de cette arborescence est la région "root" d'un diagramme SM. La figure 4.7 donne l'arborescence bipartie associée au diagramme SM de la figure 4.8. Afin de simplifier sa lecture, le nœud représentant une région est circulaire alors que celui représentant

un état est carré. Par ailleurs, deux types de filiations sont définis : entre une région et ses états fils ainsi qu'entre un état composite et ses régions filles. Ces liens de filiation sont représentées à l'aide d'une flèche orientée qui part d'une région (respectivement d'un état) vers un état fils (respectivement une région fille).

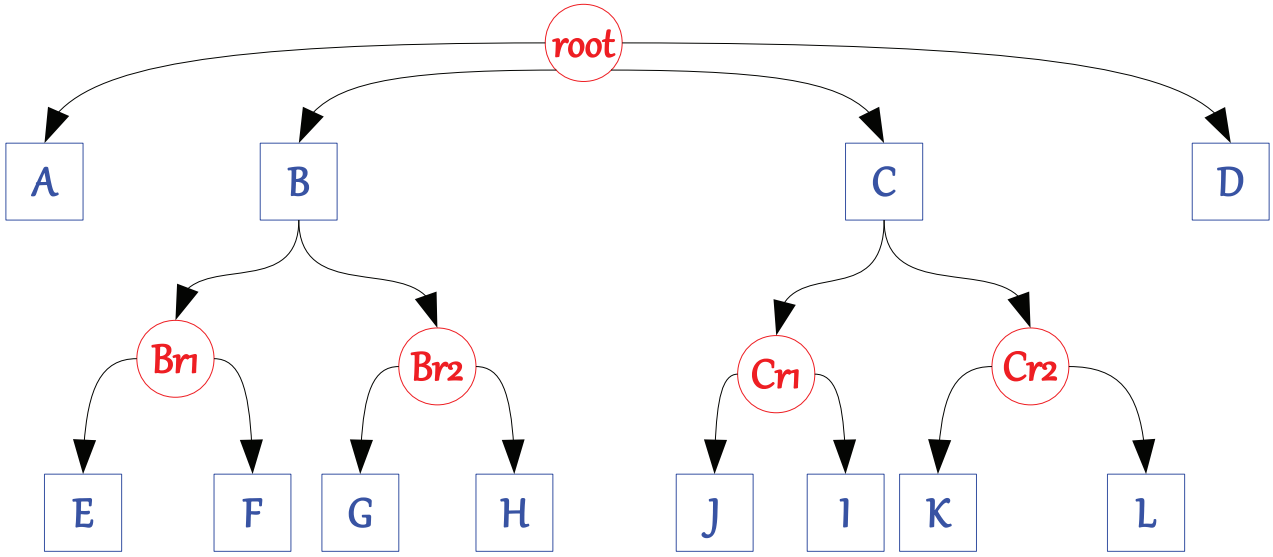

FIGURE 4.7: Graphe associé au diagramme SM de la figure 4.8

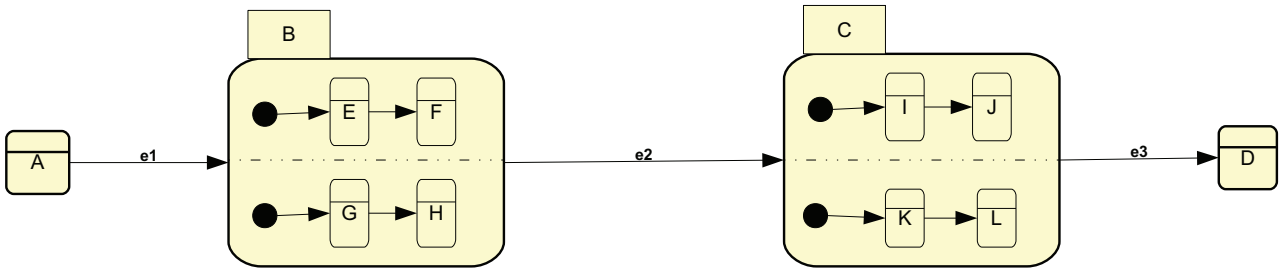

FIGURE 4.8: Diagramme SM de l'arborescence de la figure 4.7

Dans la suite, nous donnons une définition formelle de cette arborescence bipartie. La définition de celle-ci repose sur une fonction d'accessibilité, donnée ci-après.

Définition 4.10 *Soit E un ensemple de $\alpha-$tuplets tel que $\alpha \geqslant 2$. Soit $X\#n$ la projection retournant la n^{ime} composante d'un $m-$tuplet X avec $n \leqslant \alpha$. Soit E^* l'ensemble des mots obtenus par concaténation d'éléments de E.*

– $E^{\downarrow}_{root} = \{s | s \in E^* \wedge (\forall j \in [1, |s|-1](s^j\#\alpha = s^{j+1}\#1) \wedge s^1\#1 = root\})$

avec s^j une fonction qui retourne la j^{ime} "lettre" (qui est un tuplet) du mot s et $|c|$ une fonction qui retourne la longueur du mot c.

$E^{\downarrow}_{root}$ permet de retourner les mots accessibles à partir de la lettre "root".

Exemple 4.2 *Soit $\mathcal{F}$ un ensemble de trois $\alpha-$tuplets : ρ_1, ρ_2 et ρ_3.*

$$\mathcal{F} = \{\underbrace{\overbrace{(initial, \ldots, a)}^{\alpha-tuples}}_{\rho_1}, \underbrace{(\overbrace{a}^{\rho_2\#1}, \ldots, \overbrace{b}^{\rho_2\#\alpha})}_{\rho_2}, \underbrace{(\overbrace{b}^{\rho_3\#1}, \ldots, \overbrace{c}^{\rho_3\#\alpha})}_{\rho_3}\} \text{ avec } \rho_1\#1 = \text{initial et } \rho_1\#\alpha = \text{a}$$

Par définition, nous avons $\begin{pmatrix} \rho_1 \in \mathcal{F}^{\downarrow}_{initial} & \wedge \\ \rho_1.\rho_2 \in \mathcal{F}^{\downarrow}_{initial} & \wedge \\ \rho_1.\rho_2.\rho_3 \in \mathcal{F}^{\downarrow}_{initial} & \end{pmatrix}$.

$\mathcal{F}^{\downarrow}_{initial}$ *permet de retourner les mots accessibles à partir de la lettre "initial". Par analogie à la théorie de langage,* $\mathcal{F}^{\downarrow}_{initial}$ *est un sous-ensemble de* $\mathcal{F}^*$, $\mathcal{F}^{\downarrow}_{initial} \subseteq \mathcal{F}^*$

Définition 4.11 *Une arborescence bipartie* $\mathcal{A} = (N, R, H, root)$ *est un graphe dans lequel*
- N *et* R *sont deux ensembles de sommets distincts,*
- H *est une relation de hiérarchie ;* $H \subseteq (N \times R) \cup (R \times N)$,
- $root \in R$ *est utilisé afin de désigner la région racine*[3] *du SM,*

et en respectant les propriétés suivantes :

1. $(N \uplus R, H)$ *est un graphe orienté avec* $\uplus$ *est la fonction* ***Multiensemble*** *(****Multiset*** *en anglais),*
2. $(\forall p, p' \in A^{\downarrow}_{root})\ (\exists i)\ (i < |p| \wedge i < |p'| \wedge p^i = p'^i \wedge (\forall j)(i < j \leqslant Min(|p|, |p'|))(p^j \neq p'^j))$, *où* $Min(s) = \{n | n \in s \wedge (\forall x \in s)(\neg(x < n))\}$ *et la relation* $n > m$ *exprime que* n *est un ancêtre de* m. *Cette condition assure l'absence de cycles dans l'arborescence.*
3. $(\forall p \in A^{\downarrow}_{root})\ (\forall i \in [1, |p| - 1])\ (p^i \in N \Rightarrow p^{i+1} \in R \wedge p^i \in R \Rightarrow p^{i+1} \in N)$. *Cette condition assure l'alternance entre nœuds et régions.*

Remarque 4.3 *Ici, nous utilisons la notion d'arborescence bipartie et non pas la définition standard d'un graphe biparti. En effet, un graphe non orienté est dit biparti ssi il ne contient pas de cycle de longueur impaire. Alors qu'une arborescence est un graphe orienté dans lequel on ne peut jamais avoir de cycle.*

4.3.2.1 Notations et fonctions

Afin de rendre opérationnelle l'exploitation de cette représentation, nous définissons dans cette section un ensemble de fonctions qui seront par la suite utilisées lors de la définition de la syntaxe ainsi que pour la définition de la sémantique que nous proposons pour les SM.

Soit $\mathcal{A} = (N, R, H, root)$ une arborescence bipartie. L'ensemble des définitions données dans la suite sont toutes relatives à $\mathcal{A}$.

Définition 4.12 *Pour* $n, m \in N \uplus R$, $n \longrightarrow m$ *exprime que* m *est un fils de* n : $n \longrightarrow m \Leftrightarrow (n, m) \in H$.

Définition 4.13 *Deux régions* $r_1 \in R$ *et* $r_2 \in R$ *sont dites orthogonales si et seulement si* $(\exists!\ n \in N)(n \to r_1 \wedge n \to r_2)$.

Définition 4.14 *Pour* $x, y \in N$, $x \perp\!\!\!\perp y$ *dénote le fait que deux nœuds* x *et* y *appartiennent à deux régions orthogonales.*

Définition 4.15 *Pour* $n, m \in N \uplus R$, *la relation* $n > m$ *exprime que* n *est un ancêtre de* m.

$n > m$ *est la fermeture transitive de* $\to$ *et elle est définie comme suit :* $(\exists \sigma \in (N \uplus R)^*)(\sigma^1 = n \wedge \sigma^{|\sigma|} = m \wedge (\forall j \in [1, |\sigma| - 1])(\sigma^j \to \sigma^{j+1}))$

Remarque 4.4 *Un nœud peut avoir plusieurs ancêtres dont un seul direct qui est son père.*

Définition 4.16 *Les nœuds frères d'un nœud* r *sont les éléments de l'ensemble* $Siblings(r) = \{s | (\exists f)(f \to r \wedge s \neq r \wedge f \to s)\}$.

3. La région racine (*root*, en anglais) est une région spécifique qui ne possède pas de nœud parent

Exemple 4.3 *En se référant à la figure 4.7, $Siblings(A) = \{B, C, D\}$.*

Définition 4.17 *Par analogie, l'ensemble des nœuds fils (aussi appelé descendance) d'un nœud s est donné par $\mathcal{D}(s) = \{n \in N | n \leqslant s\}$.*

Exemple 4.4 *En se référant à la figure 4.7, $\mathcal{D}(B) = \{E, F, G, H, B\}$.*

Remarque 4.5 *La descendance stricte d'un ensemble de nœuds S est $\mathcal{D}^*(S) = \{n \in N | n < S\}$, autrement dit, $\mathcal{D}^*(S) = \mathcal{D}(S) \backslash \{S\}$*

Remarque 4.6 $x \perp\!\!\!\perp y \Leftrightarrow \neg(x < y \vee x > y)$.

Définition 4.18 *L'ensemble des ancêtres en commun (noté CA) d'un ensemble S de nœuds est $CA(S) = \bigcap_{i \in S} \{n | n > i\}$.*

Exemple 4.5 *En se référant à la figure 4.7, $CA(\{E, F, G\}) = \{B, root\}$.*

Définition 4.19 *$Min(S)$ est une fonction qui retourne les nœuds les plus proches des feuilles dans un ensemble S. $Min(S) = \{n | n \in S \wedge (\forall x \in S)(x > n \vee x \perp\!\!\!\perp n \vee x = n)\}$. Autrement dit, $Min(S) = \{n | n \in S \wedge (\forall x \in S)(\neg(x < n))\}$.*

Exemple 4.6 *En se référant à la figure 4.7, $Min(\{H, K, L, Cr1, Cr2, C\}) = \{H, Cr1, K, L\}$.*

Définition 4.20 *L'ancêtre commun le plus proche des feuilles est $\mathsf{LCA}(S) = Min(CA(S) \cap R)$.*

Exemple 4.7 *En se référant à la figure 4.7, $\mathsf{LCA}(\{E, F, G\}) = \{B\}$.*

Définition 4.21 *La fonction Max sur un ensemble de nœuds S est définie comme suit : $Max(S) = \{n | n \in S \wedge (\forall x \in S)(\neg(x > n))\}$.*

Remarque 4.7 *La fonction Max est la fonction duale de Min.*

Exemple 4.8 *En se référant à la figure 4.7, $Max(\{H, K, L, Cr1, Cr2, C\}) = \{H, C\}$.*

Définition 4.22 *Un chemin entre s et t noté $\mathcal{P}_s^t = \{n \in N | s \leqslant n \leqslant t\}$ est l'ensemble de nœuds formant un chemin de s vers t. Dans la suite de ce manuscrit, nous utilisons $[s, t]$ pour $\mathcal{P}_s^t$ et $[s, t[$ pour $\mathcal{P}_s^t \backslash \{t\}$, autrement dit, $[s, t[= \{n \in N | s \leqslant n < t\}$*

Exemple 4.9 *En se référant à la figure 4.7, $\mathcal{P}_G^{root} = \{n \in N | G \leqslant n \leqslant root\} = \{B, G\}$.*

4.3.2.2 SM en tant qu'arborescence bipartie

Nous définissons un diagramme *SM* comme un 12-tuplet $(N, R, H, Tr, I, O, C, root, Default, Inv, \Sigma, \mathcal{L})$ où

1. N : un ensemble de nœuds,
2. R : un ensemble de régions,
3. H : une relation de hiérarchie $(N \times R) \cup (R \times N)$, $(a, b) \in H$ est noté $a \rightarrow b$,
4. $Tr \subseteq N \times 2^I \times C \times 2^O \times N$ est l'ensemble de transitions,
5. I : ensemble d'événements d'entrée (*input*),
6. O : ensemble d'événements de sortie (*output*),

7. $C \subseteq ((\{upper, at\_least\} \times \mathbb{N}) \times (\{at\_most, lower\} \times \mathbb{N})) \cup (symbole \times \mathbb{N}) \cup \{\varnothing\}$ est l'ensemble des gardes des transitions, où *symbole* est l'ensemble des annotations temporelles proposées précédemment : $symbole = \{upper, at\_least, exactly, lower, at\_most\}$,
8. $root \in R$: est la région *racine*,
9. $Default \subseteq N$ respectant $(\forall r \in R)(\exists! n)(r \rightarrow n \wedge n \in Default)$ est l'ensemble des nœuds par défaut,
10. $Inv : N \rightarrow (\{at\_most, lower\} \times \mathbb{N}) \cup \{\varnothing\}$ est l'invariant d'un nœud SM,
11. Σ : alphabet d'étiquetage des transitions,
12. $\mathcal{L} : Tr \rightarrow \Sigma$ est la fonction d'étiquetage des transitions du SM.
13. $(N, R, H, root)$ une arborescence bipartie.
14. $(\forall (s,t) \in H \cap (N \times R))$ $(\exists u \in N | (t,u) \in H \cap (R \times N))$ (i.e chaque région doit contenir au moins un nœud).

Par la suite, nous donnons une fonction clé pour la sémantique des SM et qui sera utilisée dans la transformation que nous proposons. La fonction en question permet d'exprimer les effets du franchissement d'une transition. Ces effets sont divisés en deux parties : l'impact lié au départ depuis le nœud source : quels nœuds va-t-il falloir également quitter suite au tir de la transition ? Et l'impact lié à l'arrivée dans le nœud destination : dans quels nœuds autre que celui de la destination va-t-il falloir aller suite au tir de la transition ? En pratique, le franchissement d'une transition SM permet le passage d'une configuration SM vers une autre configuration SM. Concrètement, ce changement de configuration est composé de :

1. Retirer les états de l'impact-source de la configuration courante,
2. Mise à jour des variables contenues dans la transition,
3. Ajouter les états de l'impact-cible à la configuration obtenue en 1.

Ainsi, soit une transition t dont le nœud source est "src", et le nœud cible est "tgt". L'impact ${}^{\bullet}\mathcal{I}_{src}^{tgt}$ (respectivement $\mathcal{I}_{src}^{\bullet tgt}$) permet de déterminer les nœuds à retirer (respectivement à ajouter) suite au franchissement de t.

Exemple 4.10 *Soit la transition étiquetée avec le symbole "e_6" dans la figure 4.1. L'exécution de "e_6" implique*

1. *sur le nœud source :*
   - *de quitter le nœud "*G*"*
   - *de quitter le nœud "*AND*"*
2. *sur le nœud cible*
   - *d'activer le nœud "*OR*"*
   - *d'activer le nœud "*C*"*

Ainsi, les nœuds affectés par l'impact-source de "e_6" sont AND, E, D *et* G. *Par analogie, les nœuds affectés par l'impact-cible de "e_6" sont* OR *et* C.

Définition 4.23 *Soit $h_s = Max([src, \mathsf{LCA}(\{src, tgt\})[)$. L'impact-source de la transition permettant d'aller de src à tgt est défini comme suit :*

$${}^{\bullet}\mathcal{I}_{src}^{tgt} = \mathcal{D}(h_s) \setminus \bigcup_{(i \in [src, h_s[)} \mathcal{D}(Siblings(i))$$

Définition 4.24 *Soit $h_t = Max([tgt, \mathsf{LCA}(\{src, tgt\})[)$. L'impact-cible est défini comme suit :* $\mathcal{I}_{src}^{\bullet tgt} = [tgt, h_t[\cup ((\mathcal{D}(h_t) \setminus \bigcup \mathcal{D}(Siblings(i))) \cap Default)$.

Remarque 4.8 *L'utilisation d'une borne supérieure stricte dans les cas* $i \in [src, h_s[$ *et* $i \in [tgt, h_t[$ *trouve sa justification, respectivement, dans le fait que* $Siblings(h_s) \cap \mathcal{D}(h_s) = \varnothing$ *et* $Siblings(h_t) \cap \mathcal{D}(h_t) = \varnothing$

Définition 4.25 *Soit un vecteur de variables* $\alpha = (\alpha_1, \ldots, \alpha_n)$. *La fonction* $\alpha[\beta]$, *avec* $\beta = (\alpha_i, \ldots, \alpha_j)$ *et* $1 \leqslant i \leqslant j \leqslant n$, *retourne* β.

Remarque 4.9 *Dans la suite, nous utilisons le symbole* $\times$ *pour désigner le produit cartésien généralisé traditionnellement modélisé en utilisant le symbole* $\prod$.

4.3.3 Diagramme SM : sémantique

L'absence d'une sémantique formelle [Fecher 2005] a incité la communauté d'utilisateurs d'UML à proposer des sémantiques formelles plus ou moins conformes à la spécification d'origine des SM d'UML. Par exemple, [Crane 2005] cite 25 propositions de sémantique pour les diagrammes SM d'UML. La proposition de sémantique sur laquelle repose une partie de notre travail est celle proposée dans [Eshuis 2000]. Ce choix est justifié par le fait que cette sémantique (1) propose une structure de Kripke étiquetée (en anglais : Clocked Labelled Kripke Structure (CLKS)) pour définir une sémantique formelle des SM d'UML et (2) traite la majorité des concepts de la syntaxe des SM. Cette proposition de sémantique n'est pas totalement conforme à la spécification d'UML. En effet, lors de la définition des CLKS, [Eshuis 2000] ne fait pas la distinction entre les états stables (appartenant à une configuration stable) et les états intermédiaires (appartenant à une configuration non stable) [Crane 2005]. Dans la suite, nous nous basons sur une partie de cette formalisation afin de proposer une sémantique définissant un système de transitions étiquetées (STE).

4.3.3.1 Sémantique d'exécution : description textuelle

La sémantique des SM repose sur le concept d'*exécution jusqu'à terminaison* (Run To Completion (RTC)). Le RTC décrit le passage d'une configuration stable à une autre. Une machine à états est dite en configuration stable lorsque toutes les activités des états actifs sont terminées et qu'à partir de cette configuration, aucune transition n'est franchissable. Un élément clé lié au RTC est la liste des événements *"I"*. En effet, en SM plusieurs événements peuvent être envoyés simultanément à une SM. Contrairement à certaines extensions des automates temporisés, dans lesquelles la communication se fait à travers des canaux de communication, les SM utilisent une file, notée *"I"*, pour enregistrer les événements. Autrement dit, deux types de communication sont définis en automate temporisé : l'émetteur reste bloqué jusqu'à ce qu'un récepteur se synchronise avec lui, ou il envoie le signal sans attendre une synchronisation avec un récepteur. Dans ce dernier cas la synchronisation est perdue. Tandis qu'en UML chaque événement envoyé est enregistré dans une file d'attente pour une utilisation ultérieure.

Un élément fondamental dans la sémantique des SM exige que la machine à états soit toujours dans une configuration active : elle est en attente d'un événement pour franchir une transition. Concrètement, pour tirer une transition étiquetée avec un événement e, ce dernier doit être dans la liste *"I"*. En pratique, les SM procèdent comme suit : à partir des événements contenus dans *"I"*, l'ensemble de toutes les transitions franchissables à partir de la configuration courante est sélectionné sous réserve que leurs gardes associées (si elles en possèdent) soient évaluées à vrai. Cet ensemble définit la notion de *pas* en UML. Ensuite, le *pas* est exécuté, les événements associés à celui-ci sont retirés de *"I"* et les événements générés sont mis dans *"I"*. En outre, la machine à états passe d'une configuration stable à une autre. Ce processus se répète

tant qu'il y a des événements dans *"I"*. Bien évidement, il se peut qu'un *pas* ne consomme pas tous les événements. Cela est dû au fait qu'il y a des événements *différés*, c'est-à-dire qui n'activent aucune transition. Ce dernier type d'événements est ignoré [4] lors de l'exécution du *pas* : ils peuvent néanmoins être utilisés ultérieurement dans un futur *pas*.

Dans le cas où un événement active deux transitions concurrentes, les SM définissent plusieurs politiques de sélection : (1) la transition qui a la priorité la plus élevée ou (2) un choix indéterministe. Dans le cas où les transitions sortant d'un même état ont la même priorité, UML utilise des critères prédéfinis, comme, par exemple, les transitions étiquetées avec le même symbole provenant d'un sous-état, ont une priorité supérieure à celles provenant de l'un de ses ancêtres.

4.3.3.2 Annotations temporelles

Les diagrammes d'états définissent deux types d'annotations temporelles pour spécifier des contraintes temporelles pour le tir d'une transition. Ces deux annotations sont *after* et *when* qui permettent de spécifier respectivement de manière relative (temps écoulé depuis) et de manière absolue (date précise) une contrainte temporelle. De plus, UML donne la possibilité à l'utilisateur de spécifier ces contraintes de garde en utilisant OCL (Object Constraint Language) qui est un langage de spécification de contraintes étendant UML. Afin de profiter de la flexibilité d'expression pour les contraintes de garde, et dans le but de simplifier la tâche de spécification des contraintes temporelles, nous avons défini un ensemble d'annotations temporelles en langage naturel. Ainsi, nous mettons à disposition de l'utilisateur un ensemble prédéfini d'annotations temporelles, rendant ainsi possible l'expression des contraintes temporelles couramment utilisées. L'idée est de cadrer l'utilisateur durant la tâche de modélisation afin de produire une description claire et précise tout en manipulant des concepts simples et précis. L'ensemble des annotations est défini dans le tableau 4.1. Il est important de noter que les annotions prédéfinies d'UML, à savoir *after* et *when*, ne sont plus autorisées sur les modèles SM.

	Annotation Temporelle	Signification
1	at_most(T_{max})	la transition peut être tirée au plus tard T_{max} unités de temps après l'activation de l'état source
2	at_least(T_{min})	la transition peut être tirée à partir T_{min} unités de temps après l'activation de l'état source
3	at(d)	la transition peut être tirée exactement d unités de temps après l'activation de l'état source
4	upper(T_{min})	la transition peut être tirée après T_{min} unités de temps après l'activation de l'état source
5	lower(T_{max})	la transition peut être tirée avant T_{max} unités de temps après l'activation de l'état source

Table 4.1: Liste des annotations temporelles

Différents cas d'études autour de systèmes à contraintes de temps nous ont permis de définir cette liste d'annotations temporelles. Une horloge est associée à chaque nœud SM. Ainsi, une annotation donnée exprime une contrainte sur l'horloge associée au nœud source de la transition étiquetée par l'annotation.

4. Mais on les garde dans *"I"*

4.3.3.3 Sémantique d'exécution : description formelle

Comme nous l'avons expliqué précédemment, nos travaux reposent sur la sémantique proposée par [Eshuis 2000]. Néanmoins, afin de clarifier certains éléments clés dans cette sémantique, principalement la notion de configuration et les transitions entre configurations, nous avons choisi de définir notre propre définition formelle de la sémantique des SM. Cette sémantique sera utilisée tout au long du reste du mémoire.

Un système des machines à états (noté **SMS**) formées par n ($n \in \mathbb{N}^*$) diagrammes SM est défini comme un couple $\langle \mathbb{S}, \mathbb{E} \rangle$ où :

- $\mathbb{S} = \{ s_i \mid s_i = (N_i, R_i, H_i, Tr_i, I_i, O_i, C_i, root_i, Default_i, Inv_i) \}$ avec $i \in [1, n]$ est l'ensemble des machines à états,
- $\mathbb{E} = \bigcup_{i \in [1,n]} O_i \cup I_i$.

L'état dans lequel se trouve un système est donné par le triplet (a,b,c) :

1. *Une configuration nodale* **a**, donnant la liste des configurations nodales locales, c'est-à-dire, les nœuds actifs pour chacune des machines. **a** est un tuplet de dimension $Card(\mathbb{S})$ et $a \in \bigtimes_{i \mid s_i \in \mathbb{S}} (2^{N_i} \backslash \{\varnothing\})$.
2. *Une configuration temporelle* **b**,donnant la liste des configurations temporelles locales, c'est-à-dire, la valuation des horloges pour chacune des machines. $b \in \bigtimes_{i \mid s_i \in \mathbb{S}} \mathbb{R}_+^{\|N_i\|}$.
3. *Une configuration événementielle* **c**, donnant l'ensemble des événements en attente de traitement par les machines du système. $c \in 2^{\mathbb{E}}$

Définition 4.26 *Soit un tuplet $t \in \bigtimes_{i \in Y} \mathcal{Z}_i$. La fonction $\overbrace{t} : Y \to \mathcal{Z}_i$ retourne la composante du tuplet t associé à l'élément de Y passé en paramètre. Par exemple, si t est une configuration nodale : $t \in \bigtimes_{i \mid s_i \in \mathbb{S}} (2^{N_i} \backslash \{\varnothing\})$, alors $\overbrace{t} : \mathbb{S} \to 2^{N_i} \backslash \{\varnothing\}$ retourne la configuration nodale locale correspondant au système passé en paramètre.*

La configuration nodale locale initiale d'une machine i est donnée par $\mathcal{C}_i = \bigcup_{j=0}^{\infty} \mathcal{C}_i^j$ où :

- $\mathcal{C}_i^0 = \{n | root_i \to n \wedge n \in Default_i\}$
- $\mathcal{C}_i^j = \{n | (\exists x \in \mathcal{C}_i^{j-1})(\exists r \in R_i)(x \to r \to n \wedge n \in Default_i)\}$, pour $j \geqslant 1$.

À partir de la région *root*, la fonction retourne le nœud par défaut. Le nœud retourné est retenu dans $\mathcal{C}_i$. Ensuite, si le nœud retenu est de type *composite*, la fonction rajoute les nœuds par défaut pour chacune de ses régions à $\mathcal{C}_i$. Par la suite, un test de type est effectué sur ces nœuds retournés. La procédure est répétée jusqu'à ce que tous les éléments de $\mathcal{C}_i$ sont testés.

Remarque 4.10 *Existence d'un point fixe pour le calcul de n'importe quelle configuration :*
$\mathcal{C}_i$ est une fonction monotone : on part de la racine et à chaque étape on "descend" d'un cran vers les feuilles sans jamais remonter. De plus, une arborescence est un graphe orienté acyclique admettant une racine unique. Ceci implique donc que tout chemin de la racine vers les feuilles est fini : le nombre d'étapes nécessaire à $\mathcal{C}_i$ est donc borné par la profondeur maximale du graphe.

Soit $SMS = \langle \mathbb{S}, \mathbb{E} \rangle$ un système de machine à états avec :
$\mathbb{S} = \{ s_i \mid s_i = (N_i, R_i, H_i, Tr_i, I_i, O_i, C_i, root_i, Default_i, Inv_i, \Sigma_i, L_i) \}$ avec $i \in [1, n]$.
La sémantique de SMS est donné par le STT $S_{sm} = (S_{sm}, s_{sm}^0, \to_{sm}, \Sigma_{sm})$ où

– $\Sigma_{sm} = \bigcup_{i|s_i \in \mathbb{S}} \Sigma_i$.
– $s^0_{sm} = (\underset{i|s_i\in\mathbb{S}}{\times}\{\mathcal{C}_i\}, \underset{i|s_i\in\mathbb{S}}{\times}\{0^{N_i}\}, \varnothing)$ est la configuration initiale où $\underset{i|s_i\in\mathbb{S}}{\times}\{\mathcal{C}_i\}$ est la configuration nodale initiale, $\underset{i|s_i\in\mathbb{S}}{\times}\{0^{N_i}\}$ est la configuration temporelle initiale et $\varnothing$ est la configuration événementielle initiale.
– S_{sm} est l'ensemble des états accessibles à partir de s^0_{sm} en utilisant la relation $\rightarrow_{sm}$.
– La relation de transition $\rightarrow_{sm}$ est définie comme suit :

Comme pour les automates temporisés, nous définissons deux types de transitions : la transition d'action et la transition de temps.

A. Transition d'action

Le premier type est la transition d'action t étiquetée par un symbole $a = \mathcal{L}(t) \in \Sigma_{sm}$ définie comme suit :

$$(\mathcal{C}_n, \gamma, \lambda) \xrightarrow{a\in\Sigma} (\mathcal{C}'_n, \gamma', \lambda') \Leftrightarrow$$

$$(\exists! j)(\exists t = (n, i, c, o, n') \in Tr_j) \left(\begin{array}{ll} L_j(t) = a & \wedge \\ n \in \widehat{\mathcal{C}}_n(j) & \wedge \\ c = (s, m) \Rightarrow & \\ \left(\begin{array}{lll} s = upper & \Rightarrow & \overbrace{\widehat{\gamma}(j)}(n) > m \quad \wedge \\ s = at\_least & \Rightarrow & \overbrace{\widehat{\gamma}(j)}(n) \geqslant m \quad \wedge \\ s = at & \Rightarrow & \overbrace{\widehat{\gamma}(j)}(n) = m \quad \wedge \\ s = lower & \Rightarrow & \overbrace{\widehat{\gamma}(j)}(n) < m \quad \wedge \\ s = at\_most & \Rightarrow & \overbrace{\widehat{\gamma}(j)}(n) \leqslant m \end{array} \right) & \wedge \\ i \subseteq \lambda & \wedge \\ \lambda' = (\lambda \backslash i) \cup o & \wedge \\ \widehat{\mathcal{C}}'_n(j) = (\widehat{\mathcal{C}}_n(j) \backslash {}^{\bullet}\mathcal{I}^{n'}_n) \cup (\mathcal{I}^{\bullet n'}_n) & \wedge \\ (\forall n \in \mathcal{I}^{\bullet n'}_n)(\overbrace{\widehat{\gamma'}(j)}(n) = 0) & \wedge \\ (\forall n \notin \mathcal{I}^{\bullet n'}_n)(\overbrace{\widehat{\gamma'}(j)}(n) = \overbrace{\widehat{\gamma}(j)}(n)) & \end{array} \right)$$

En langage naturel, à partir d'une configuration de départ définie par le triplet : la configuration nodale $\mathcal{C}_n$, la configuration temporelle γ et la configuration événementielle λ, il existe une transition $t = (n, i, c, o, n')$ étiquetée par un symbole $a = \mathcal{L}(t) \in \Sigma_{sm}$ dont la cible est une configuration définie aussi par un triplet : $\mathcal{C}_{n'}$, γ' et λ'. Étant donné que $\mathbb{S}$ est composé par plusieurs machines à états, il existe une unique machine $SM_j \in \mathbb{S}$ avec $n, n' \in SM_j$. Pour qu'une transition puisse être tirée, il faut que la valuation locale des horloges associée au nœud n, $\overbrace{\widehat{\gamma}(j)}(n)$, vérifie la contrainte donnée par $c = (s, m)$ et il faut que l'événement i associé à t soit déjà dans la configuration événementielle, $i \subseteq \lambda$. Une fois que la transition est tirée, l'événement i est retiré de la configuration événementielle à laquelle l'événement o est ajouté, $\lambda' = (\lambda \backslash \{i\}) \cup \{o\}$. La nouvelle configuration nodale $\mathcal{C}'_n$ est obtenue en retirant les nœuds de l'impact-source et en ajoutant les nœuds de l'impact cible, $\widehat{\mathcal{C}}'_n(j) = (\widehat{\mathcal{C}}_n(j) \backslash {}^{\bullet}\mathcal{I}^{n'}_n) \cup (\mathcal{I}^{\bullet n'}_n)$. Pour la nouvelle configuration des hor-

loges, seules les horloges associées à l'impact-cible sont réinitialisées, $(\forall n \in \mathcal{I}_n^{\bullet n'})(\overbrace{\overbrace{\gamma'}(j)}(n) = 0)$.

La valuation des autres horloges reste la même, $(\forall n \notin \mathcal{I}_n^{\bullet n'})(\overbrace{\overbrace{\gamma'}(j)}(n) = \overbrace{\gamma}(j)(n))$.

B. Transition de temps

$$(\mathcal{C}_n, \gamma, \lambda) \xrightarrow{\theta \in \mathbb{R}_+^*} (\mathcal{C}'_n, \gamma', \lambda') \Leftrightarrow$$

$$\left(\begin{array}{ll} \theta \in \mathbb{R}_+^* & \wedge \\ \mathcal{C}_n = \mathcal{C}'_n & \wedge \\ \lambda' = \lambda & \wedge \\ (\forall j \in \mathbb{S})(\forall n \in N_j)(\overbrace{\overbrace{\gamma'}(j)}(n) = \overbrace{\overbrace{\gamma}(j)}(n) + \theta) & \wedge \\ (\forall n \in \mathcal{C}_n)\left(Inv(n) = (s,k) \Rightarrow \left[\begin{array}{lll} s = lower & \Rightarrow & \overbrace{\overbrace{\gamma}(j)}(n) + \theta < k \quad \wedge \\ s = at\_most & \Rightarrow & \overbrace{\overbrace{\gamma}(j)}(n) + \theta \leqslant k \end{array} \right] \right) & \end{array} \right)$$

En langage naturel, à partir d'une configuration de départ définie par le triplet : la configuration nodale $\mathcal{C}_n$, la configuration temporelle γ et la configuration événementielle λ, on laisse avancer le temps d'une valeur $\theta \in \mathbb{R}_+^*$ tant que cette valeur vérifie les invariants associés à la configuration nodale de départ, $\overbrace{\overbrace{\gamma}(j)}(n) + \theta \vDash Inv(n)$. Ainsi, seule la configuration des horloges sera modifiée. Autrement dit, la nouvelle configuration nodale ainsi que la configuration événementielle restent inchangées. Cependant, dans la nouvelle configuration des horloges, seules les horloges associées à la configuration nodale de départ vont être changées, $\overbrace{\overbrace{\gamma'}(j)}(n) = \overbrace{\overbrace{\gamma}(j)}(n) + \theta$.

4.4 Automate temporisé : définition et sémantique

Les automates temporisés (ATs) ont été introduits pour la première fois par Alur [Alur 1994] en 1994 et ont été par la suite étendus avec la notion d'invariant d'états par Henzinger [Henzinger 1994]. Certains travaux, comme [Berrada 2005], ainsi que certains outils d'analyse et de vérification, comme Uppaal [Larsen 1997], proposent d'étendre les AT par des variables. Ce type d'extension est très intéressant d'un point de vue pratique puisqu'il permet à l'utilisateur de considérer différents types de données afin de faciliter la tâche de modélisation.

4.4.1 Automate temporisé étendu avec des variables booléennes et des événements de synchronisation

Soit un système d'automates temporisés à variables booléennes synchronisables par canaux de diffusion (en anglais *broadcast*) $SAT - BD = \langle A, \mathbb{C}, V \rangle$ où :

1. V est un ensemble de variables booléennes,
2. $\mathbb{C}$ est l'ensemble de canaux de diffusion,
3. $A = \{AT_i | AT_i = (L_i; \ell_0^i; X_i; T_i; Inv^i; G_i; C_i; R_i; \mathbb{T}_i; \mathbb{F}_i; Sync_i; \Sigma_i)$, où pour tout AT_i :
   - L_i est un ensemble fini de localités,
   - $\ell_0^i \in L_i$ est l'état initial,
   - X_i est un ensemble fini d'horloges,

- T_i est un ensemble de transitions, $T_i \subseteq L_i \times \Sigma_i \times L_i$ et $(\ell, a, \ell') \in T_i$ est noté $\ell \xrightarrow{a} \ell'$,
- $Inv^i : L_i \to 2^{X_i \times \mathbb{R}_+ \times \{<,\leqslant\}}$ associe un invariant à certains états de contrôle
- $G_i : T_i \to 2^{X_i \times \mathbb{R}_+ \times \{<,\leqslant,=,\geqslant,>\}}$ associe une garde sur les horloges à certaines transitions,
- $C_i : T_i \to 2^V$ associe une condition sur les variables booléennes à certaines transitions
- $R_i : T_i \to 2^{X_i}$ est l'ensemble d'horloges à réinitialiser à zéro sur une transition
- $\mathbb{T}_i : T_i \to 2^V$ est l'ensemble des variables booléennes à mettre à vrai,
- $\mathbb{F}_i : T_i \to 2^V$ est l'ensemble des variables booléennes à mettre à faux, avec $\mathbb{T}_i \cap \mathbb{F}_i = \varnothing$,
- $Sync_i : T_i \to (\mathbb{C} \times \{!, ?\}) \cup \{\varnothing\}$ associe des canaux de synchronisation à certaines transitions. Les communications synchrones admettent deux types de participants : la transition émettrice (maître), marquée d'un " *!*" accolé au canal et les transitions réceptrices (esclaves), marquées d'un " *?*".
- Σ_i est un alphabet d'actions

La sémantique d'un $SAT - BD = \langle A, \mathbb{C}, V \rangle$ est le TTS $S = \langle Q, q_0, \rightharpoonup, \Sigma_s \rangle$, où

- $Q \subseteq \bigtimes_i L_i \times \bigtimes_i \mathbb{R}^{\|X_i\|} \times 2^V$,
- $q_0 = (\bigtimes_i \ell_0^i, \bigtimes_i 0^{\|X_i\|}, \varnothing)$,
- $\Sigma_s = \bigcup_i \Sigma_i$,
- $\rightharpoonup$ est un ensemble fini de transitions. Une transition est de la forme $(\mathcal{L}, \mathcal{C}, \mathcal{V}) \overset{a/\theta}{\rightharpoonup} (\mathcal{L}', \mathcal{C}', \mathcal{V}')$ représente une transition d'une configuration représentée par un triplet $(\mathcal{L}, \mathcal{C}, \mathcal{V})$ où les composantes représentent respectivement la configuration nodale, la configuration des horloges et la configuration des variables (le sous-ensemble des variables évaluées à vrai), vers une autre configuration. Deux types de transitions sont définis : transition d'action et transition de temps.

Transition d'action est un changement de configuration qui se déroule comme suit : à partir de la configuration courante une transition t $((\exists i)(t \in T_i))$ dont la source appartenant à cette configuration est franchissable. Deux cas sont possibles : la transition sélectionnée synchronise avec d'autres transitions (c'est-à-dire $(\exists c \in \mathbb{C})(Sync^i(t) = c!)$), ou elle ne synchronise pas $(Sync^i(t) = \varnothing)$.

Dans le cas où $Sync^i(t) = c!$, toute les transitions t' dont la garde est évaluée à vrai et dont la synchronisation est $Sync^j(t') = c?$ sont sélectionnées. Afin d'empêcher une synchronisation entre deux transitions appartenant à un même automate, une condition est ajoutée aux critères de sélection, $SYNC_c^i = S \backslash \{NoSync(c) \cup \{i\}\}$. Par la suite, l'ensemble des transitions synchronisables est tiré. Pour chaque transition tirée : on remet l'horloge associée à son nœud cible à zéro $(R(t'))$, on remet à "False" les variables booléennes utilisées dans l'expression de sa garde $(\mathbb{F}(t'))$, on remet à "True" l'ensemble des variables booléennes contenues dans $\mathbb{T}(t')$.

Dans le cas où il n'y a pas de synchronisation, seule la transition t sélectionnée au départ est franchie : l'horloge associée à son nœud cible est initialisée et les variables booléennes sont mises à jour ("False" ou "True").

À cela, on rajoute une condition qui exige qu'on ne tire pas une transition si la configuration (les valuations des horloges) ne vérifie pas l'invariant de la configuration nodale cible,$(\forall i \in A)(\widehat{\mathcal{C}'}(i) \vDash Inv^i(\widehat{\mathcal{L}'}(i)))$.

$(\mathcal{L}, \mathcal{C}, \mathcal{V}) \overset{a \in \Sigma_s}{\rightharpoonup} (\mathcal{L}', \mathcal{C}', \mathcal{V}')$ ssi :

$$
\left(\begin{array}{c}
(\exists t = \ell \overset{a\in\Sigma_s}{\rightarrow} \ell')(\exists i)(\exists c \in \mathbb{C}) \\
t \in T_i \quad \wedge \\
enabled(t,\mathcal{L},\mathcal{C},\mathcal{V}) \quad \wedge \\
(Sync^i(t) = c! \vee Sync^i(t) = \varnothing) \quad \wedge \\
Sync^i(t) = c! \Rightarrow (\exists \alpha)\alpha = \underset{j\in SYNC_c^i}{\times} Syncable(j,c) \neq \varnothing \Rightarrow \\
\left(\begin{array}{c}
(\exists y \in \alpha)\left[\begin{array}{c}
(\forall j \in SYNC_c^i) \\
(\widehat{y}(j) = \ell_1 \rightarrow \ell_2 \Rightarrow \widehat{\mathcal{L}'}(j) = \ell_2 \quad \wedge \\
\overbrace{(\forall k \in R_j(\widehat{y}(j)))(\widehat{\mathcal{C}'}(j)(k) = 0)} \quad \wedge \\
\mathcal{V}' = (\mathcal{V}\backslash(\underset{j\in SYNC_c^i}{\cup}\mathbb{F}_j(\widehat{y}(j)) \cup \mathbb{F}(t)))\cup(\underset{j\in SYNC_c^i}{\cup}\mathbb{T}_j(\widehat{y}(j)) \cup \mathbb{T}(t))
\end{array}\right] \quad \wedge \\
\widehat{\mathcal{L}'}(i) = \ell' \quad \wedge \\
\overbrace{(\forall k \in R_i(t))(\widehat{\mathcal{C}}(i)(k) = 0)}
\end{array}\right) \\
(Sync^i \neq c!) \Rightarrow \mathcal{V}' = (\mathcal{V}\backslash(\mathbb{F}(t)))\cup(\mathbb{T}(t)) \quad \wedge \\
(\forall i \in A)(\widehat{\mathcal{C}'}(i) \vDash Inv^i(\widehat{\mathcal{L}'}(i)))
\end{array}\right)
$$

Transition de temps est un changement de configuration qui se déroule comme suit : dans la configuration courante on laisse passer le temps tant qu'on respecte l'invariant associé aux nœuds de cette configuration, $(\forall i)(\widehat{\mathcal{C}}(i) \vDash Inv^i(\widehat{\mathcal{L}}(i)))$. Seule la valuation des horloges change dans ce type de transition. En d'autres termes, la configuration nodale et la configuration événementielle restent les mêmes, $\mathcal{L} = \mathcal{L}' \wedge \mathcal{V} = \mathcal{V}'$.

$$
\left(\begin{array}{c}
(\mathcal{L},\mathcal{C},\mathcal{V}) \overset{\theta\in\mathbb{R}_+^*}{\rightarrow} (\mathcal{L}',\mathcal{C}',\mathcal{V}') \Leftrightarrow \\
\mathcal{L} = \mathcal{L}' \quad \wedge \\
\mathcal{V} = \mathcal{V}' \quad \wedge \\
\overbrace{(\forall i)(\forall x \in X_i)(\widehat{\mathcal{C}}(i)(x) + \theta = \widehat{\mathcal{C}'}(i)(x))} \quad \wedge \\
(\forall i)(\widehat{\mathcal{C}}(i) \vDash Inv^i(\widehat{\mathcal{L}}(i)))
\end{array}\right)
$$

Les fonctions utilisées ci-dessus sont définies comme suit :

1. La fonction utilisée pour sélectionner les transitions potentiellement franchissables :
$$
enabled(t,\mathcal{L},\mathcal{C},\mathcal{V}) = (\exists i \in A)\left(\begin{array}{c}
t = \ell \overset{a}{\rightarrow} \ell' \in T_i \quad \wedge \\
\ell = \widehat{\mathcal{L}}(i) \quad \wedge \\
\widehat{\mathcal{C}}(i) \vDash G_i(t) \quad \wedge \\
\widehat{\mathcal{L}}(i) \vDash Inv^i(t) \quad \wedge \\
C_i(t) \subseteq \mathcal{V}
\end{array}\right)
$$
2. $Syncable(j,c) = \{t \in T_j | enabled(t,\mathcal{L},\mathcal{C},\mathcal{V}) \wedge Sync^j(t) = c?\}$ est la fonction utilisée pour

sélectionner les transitions synchronisables avec l'automate j sur le canal de broadcast c.

3. $NoSync(c) = \{j \in S | (\forall t \in T_j)(\neg(enabled(t, \mathcal{L}, \mathcal{C}, \mathcal{V}) \wedge Sync^j(t) = c?))\}$ retourne les automates qui ne possèdent pas de transitions synchronisables sur le canal de broadcast c.
4. $SYNC_c^i = S \backslash \{NoSync(c) \cup \{i\}\}$
5. La fonction $\vDash$ utilisée précédemment est définie comme suit : Soit $C \in \mathbb{R}^{\|X\|}, G \subseteq X \times \mathbb{R}^+ \times \{<, \leqslant, =, \geqslant, >\}$:

   $C \vDash G \Leftrightarrow (\forall i \in [1, n])(\forall g \in G)(g = (clk, cst, sym) \Rightarrow \overbrace{C}(clk) \quad sym \quad cst)$

4.5 Algorithme de transformation

Après avoir présenté les modèles les machines à états d'UML et les automates temporisés, nous passons à la transformation de modèles entre ces deux formalismes. Dans cette section, nous allons introduire l'algorithme de transformation. Il est à rappeler que nous avons adopté une approche dirigée par les modèles pour mener à bien cette étape. Il est donc intéressant de mettre l'accent sur les éléments clés de la mise en œuvre de cette approche. Ainsi, nous commençons cette section en présentant les deux méta-modèles utilisés pour la transformation. Ensuite, une définition formelle ainsi qu'une description textuelle et graphique de l'algorithme de transformation sont données. Par la suite, une démonstration de la préservation du comportement entre les deux formalismes est donnée dans la section 4.6. La préservation est démontrée par l'existence d'une relation de bisimulation temporelle entre les deux sémantiques.

4.5.1 Modèles et méta-modèles

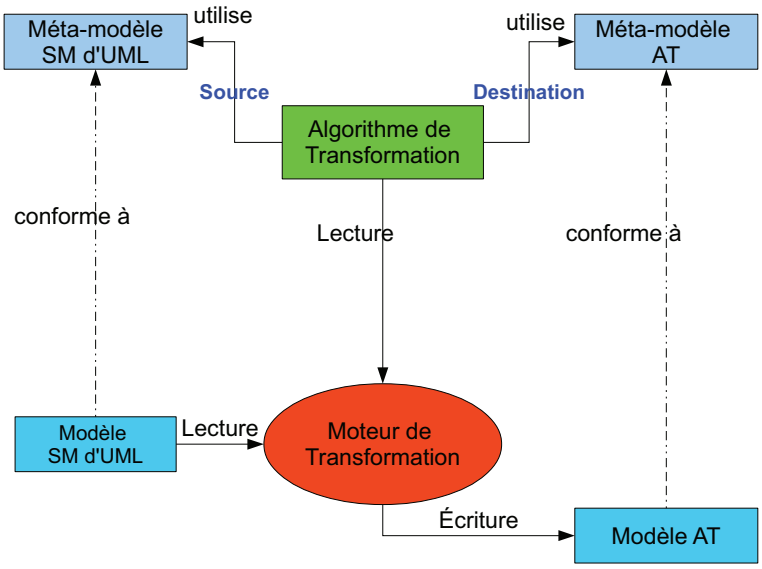

FIGURE 4.9: Transformation de modèles : UML SM vers AT

Conformément à l'approche AMD, la transformation que nous proposons est conforme à l'architecture donnée par la figure 4.9. En effet, nous proposons un algorithme de transformation dont l'idée est de définir, au niveau méta-modèle, des relations entre les éléments (entités) des modèles sources (SM) avec ceux des modèles cibles (AT). La transformation prend en entrée

un (ou plusieurs) modèle(s) SM et génère en sortie un (ou plusieurs) modèle(s) AT. L'implémentation de la transformation est faite à l'aide d'un langage généraliste (en l'occurrence un prototype en *Prolog* puis une version plus complète en *Scala*). Nous allons définir dans la suite les méta-modèles respectifs des SM et des AT. La définition de ces deux méta-modèles repose sur les syntaxes abstraites présentées ci-dessus.

4.5.1.1 Méta-modèle des SM d'UML

Le méta-modèle de SM d'UML que nous avons utilisé dans l'implémentation de l'algorithme de transformation est le méta-modèle défini dans la spécification des SM dans [OMG 2011b]. Ce méta-modèle est donné dans la figure 4.10.

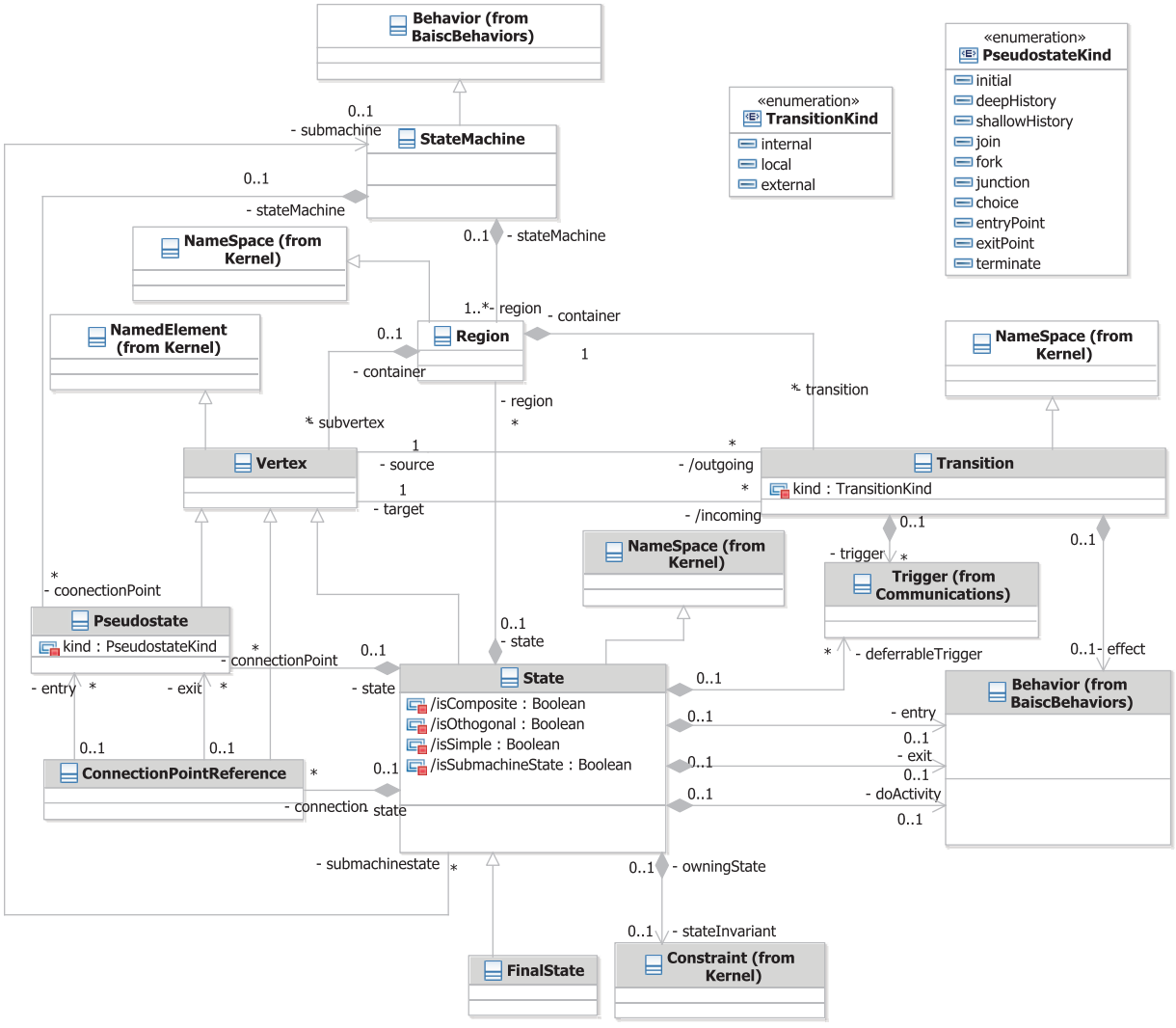

FIGURE 4.10: Méta-modèle : SM d'UML [OMG 2011b]

4.5.1.2 Méta-modèle des automates temporisés

Le deuxième méta-modèle que nous allons utiliser dans l'implémentation de l'algorithme de transformation est celui des automates temporisés. En effet, en nous basant sur la définition

formelle d'un automate temporisé donnée en section 4.4, nous avons défini nous-même un méta-modèle pour ce formalisme. Une version simplifiée de ce méta-modèle est donnée en figure 4.11.

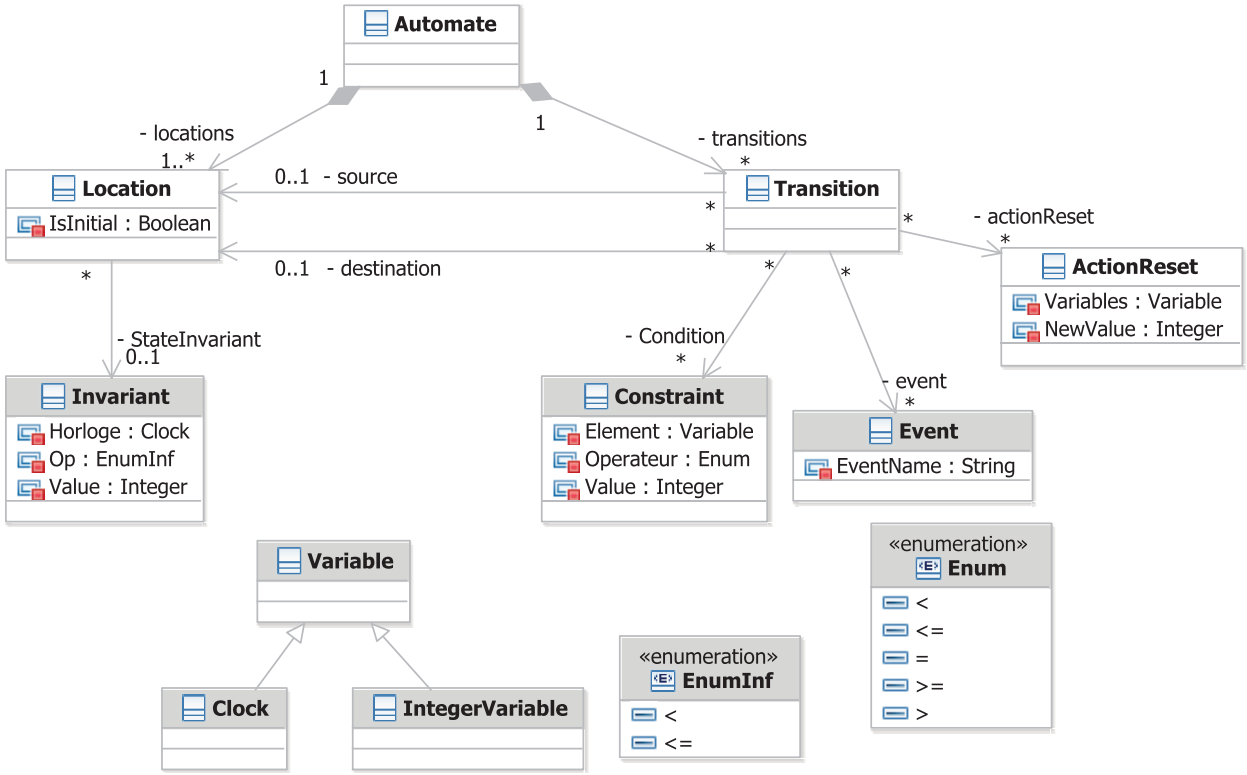

FIGURE 4.11: Méta-modèle : Automate Temporisé

4.5.1.3 Modèle asynchrone et modèle synchrone

Les SM admettent deux sémantiques présentes dans l'article de Eshuis & Wieringa [Eshuis 2000], basées sur celles de *statemate*. Ces sémantiques se différencient par le déclenchement du calcul d'un *pas* :

- soit synchrone, c'est-à-dire que le calcul est déclenché tous les tics d'horloge,
- soit asynchrone, c'est-à-dire déclenché par l'arrivée d'un nouvel événement.

Même dans le cas asynchrone, le calcul du pas reste synchrone. On a ainsi une cohabitation d'un aspect synchrone très local et d'un aspect asynchrone global (au niveau de la communication entre machines). C'est un mode assez courant dans les modèles "industriels" et qui porte le nom de GALS (Globally Asynchronous Locally Synchronous). Cependant, les automates temporisés ne permettent qu'un seul type de modélisation ; la modélisation asynchrone. C'est pourquoi nous avons eu besoin de limiter la sémantique des SM à la seule composante "GA". Un élément clé lié à cette restriction est la liste des événements *"I"*. Le formalisme des SM permet la synchronisation simultanée entre plusieurs événements de synchronisation différents. Ceci est possible grâce à la sémantique localement synchrone, qui à chaque pas inspecte l'état des événements sur lesquels il est possible de se synchroniser. Ce type de mécanisme n'est pas présent dans les automates temporisés. La synchronisation s'opère lors du tir d'une transition qui est tirable en fonction du respect de certains conditions, sans qu'il n'y ait à aucun moment besoin d'une notion de pas.

Ceci implique qu'il faille tirer l'une après l'autre les transitions qui respectent les conditions de tirs dans les automates temporisés tandis qu'un seul pas les franchit toutes en SM d'UML. Afin de n'être pas pénalisé par cette différence de paradigme, et garantir que la traduction donne des automates dont la sémantique est équivalente (à la bisimulation temporelle forte près) à la sémantique des SM d'origines, il nous faut changer la sémantique de SM afin de l'adapter à un paradigme complètement asynchrone.

Ainsi, la liste des événements *"I"* en SM sera associée en automate temporisé à un vecteur de variables booléennes. En effet, chaque événement du modèle SM sera associé à une variable booléenne dans les AT. La valeur par défaut des variables associées est **False**. Ainsi, à chaque apparition d'un événement **e**, sa variable associée v_e est mise à **True**. La transition en AT associée à cet événement doit être franchie et la variable associée à l'événement déclencheur est remise à **False**. Cette solution est facilement intégrable dans les automates temporisés avec variables.

Dans la section suivante, nous présenterons l'algorithme de transformation qui prend en entrée des représentations SM d'UML pour générer des modèles AT. Cet algorithme est défini formellement dans la section 4.5.2 et sera ensuite détaillé à l'aide de descriptions graphiques dans la section 4.5.3.

4.5.2 Algorithme de transformation : définition formelle

Soient

- $NtoL : Nodes \longrightarrow L$ est une bijection qui associe chaque nœud SM à une (et une seule) localité AT,
- $RtoL : Regions \longrightarrow L$ est une bijection qui associe chaque région SM à une unique localité AT,
- $NtoX : N \longrightarrow X$ est une bijection qui associe chaque nœud du SM source à une horloge de l'automate cible.

Soit $\langle \mathbb{S}, \mathbb{E} \rangle$ un SMS,

$$
\begin{array}{l}
(\forall i \in \mathbb{S})(\forall r \in R^i)(\exists A^i_r = (L^i_r; \ell^i_{0_r}; X^i_r; T^i_r; G^i_r; C^i_r; Inv^i_r; \mathbb{R}^i_r; \mathbb{T}^i_r; \mathbb{F}^i_r, Sync^i_r; \Sigma^i_r)) \\
\left(\begin{array}{ll}
L^i_r = \{\ell | (\exists x)(r \to x \wedge NtoL(x) = \ell)\} \cup \{RtoL(r)\} & \wedge \\
X^i_r = \{c | (\exists x)(r \to x \wedge NtoX(x) = c)\} & \wedge \\
(\exists n \in \{x | r \to x\})((n \in \mathscr{C}_i \Rightarrow \ell^i_{0_r} = NtoL(n)) \wedge (n \notin \mathscr{C}_i \Rightarrow \ell^i_{0_r} = RtoL(n))) & \wedge \\
(\exists n \in \{x | r \to x\}) \left(\begin{array}{l} Inv_i(n) = (s,k) \Rightarrow \left(\begin{array}{ll} s = lower & \Rightarrow Inv^i_r(NtoL(n)) = NtoX(n) < k \\ s = at\_most & \Rightarrow Inv^i_r(NtoL(n)) = NtoX(n) \leqslant k \end{array} \right) \wedge \\ Inv_i(n) = \varnothing \Rightarrow Inv^i_r(NtoL(n)) = \varnothing \end{array} \right) & \wedge \\
\Sigma^i_r = \bigcup\limits_{t \epsilon T^i_r} \mathcal{L}_i(t) & \wedge \\
(\forall tr \in Tr_i) \left(\begin{array}{l} tr = (src, in, c, out, tgt) \Rightarrow \\ (\exists g)(\exists v)(\exists d^\top)(\exists d^\perp) \left(\begin{array}{ll} c = (h,m) \Rightarrow \left[\begin{array}{lllll} h = upper & \Rightarrow & g = NtoX(n) > m & \wedge \\ h = at\_least & \Rightarrow & g = NtoX(n) \geqslant m & \wedge \\ h = at & \Rightarrow & g = NtoX(n) = m & \wedge \\ h = lower & \Rightarrow & g = NtoX(n) < m & \wedge \\ h = at\_most & \Rightarrow & g = NtoX(n) \leqslant m & \end{array} \right] & \wedge \\ c = \varnothing \Rightarrow g = \varnothing & \wedge \\ v = \{x == true | x \in in\} & \wedge \\ d^\top = out \wedge d^\perp = in & \wedge \\ ((\exists s = {}^\bullet\mathcal{I}^{tgt}_{src} \cap \{n | r \to n\})(\exists t = \mathcal{I}^{\bullet tgt}_{src} \cap \{n | r \to n\}))(T^i_r) & \end{array} \right) \end{array} \right) & \end{array} \right)
\end{array}
$$

avec T^i_r est définie comme suit : $(\exists s = {}^\bullet\mathcal{I}^{tgt}_{src} \cap \{n | r \to n\})(\exists t = \mathcal{I}^{\bullet tgt}_{src} \cap \{n | r \to n\})$

$$
\left(
\begin{array}{l}
T_r^i = \left\{
\begin{array}{l|lll}
\tau = toL(x) \xrightarrow{\alpha} toL(y) & (s = \varnothing & \Rightarrow & (x = r)) \wedge \\
 & (s \neq \varnothing & \Rightarrow & x \in s) \wedge \\
 & (t = \varnothing & \Rightarrow & y = r) \wedge \\
 & (t \neq \varnothing & \Rightarrow & y \in t) \wedge \neg(x = y = r) \wedge \\
 & (x \neq src & \Rightarrow & \left[\begin{array}{ll} \alpha = \varepsilon & \wedge \\ G_r^i(\tau) = C_r^i(\tau) = \mathbb{T}_r^i(\tau) = \mathbb{F}_r^i(\tau) = \varnothing & \wedge \\ Sync_r^i(\tau) = (lbl(tr), ?) & \end{array} \right] \wedge \\
 & (x = src & \Rightarrow & \left[\begin{array}{ll} \alpha = \mathcal{L}_i(tr) & \wedge \\ G_r^i(\tau) = g & \wedge \\ C_r^i(\tau) = v & \wedge \\ \mathbb{T}_r^i(\tau) = d^{\top} & \wedge \\ \mathbb{F}_r^i(\tau) = d^{\perp} & \wedge \\ Sync_r^i(\tau) = (lbl(tr), !) & \end{array} \right] \wedge \\
 & \mathbb{R}_r^i(\tau) = \{toL(y)\} &
\end{array}
\right\}
\end{array}
\right)
$$

avec $toL = (NtoL \cup RtoL)$ et lbl est une bijection associant un canal à chaque transition.

Afin d'illustrer l'algorithme que nous proposons, nous donnons dans la suite des motifs graphiques qui permettent d'aider à comprendre les différentes règles de transformation.

4.5.3 Illustration de l'algorithme de transformation : motifs graphiques

Avant de commencer à présenter les différents motifs graphiques utilisés pour illustrer la transformation, nous jugeons indispensable de présenter ici quelques éléments clés pour mieux comprendre ces motifs.

- La partie action (*effect*) d'une transition peut avoir deux types d'action (1) envoi des événements ou (2) traitement sur des données. Dans la suite, nous utilisons (a) *actionEventOf* **(eff)** comme une fonction qui retourne les événements associés à une action **eff** et (b) *actionDataOf* **(eff)** comme une fonction qui retourne les affectations (données et horloges) associées à une action **eff**.
- Le trigger d'une transition SM correspond à un événement dont le déclenchement permet de franchir la transition. Dans le cadre de nos travaux, nous traitons les événements UML de type *SignalEvent*.
- Afin de cadrer la partie de spécification de gardes de transitions SM, seules les annotations temporelles que nous avons définies sont considérées dans l'algorithme de transformation.

4.5.3.1 Motif : nœud basique

Les figures 4.12, 4.13 et 4.14 donnent les motifs de traduction de nœud basique. Un nœud basique est un nœud de type *"UML::State"* et dont le nombre de régions est égal à zéro ($self.region- > size() = 0$ *and not self.oclIsTypeOf(UML::FinalState)*). Concrètement, un nœud basique se traduit en une localité (ou état) en AT de type *"AT::Location"*. Les transitions (*"UML::Transition"*) entrantes vers ce nœud basique se traduisent en des transitions (*"TA::Transition"*) entrantes vers la localité associée à ce nœud basique. De la même façon, les transitions (*"UML::Transition"*) sortantes à partir de ce nœud basique se traduisent en des transitions (*"TA::Transition"*) sortantes de la localité associée à ce nœud basique.

Il est à noter que les motifs de traduction graphiques présentent tous les cas possibles pour un nœud basique. En pratique, un nœud basique peut être un état initial (figure 4.12) ou un état ordinaire (figure 4.13) ou les deux en même temps (figure 4.14).

Comme nous pouvons le noter tout au long de cette section et conformément à la définition donnée en section 4.5.2, la traduction des nœuds (basique, OR et AND) est une fonction bijective qui associe à chaque nœud SM *"UML::State"* une localité AT *"TA::Location"*. Cependant la traduction de transitions est beaucoup moins évidente que la traduction des nœuds. En

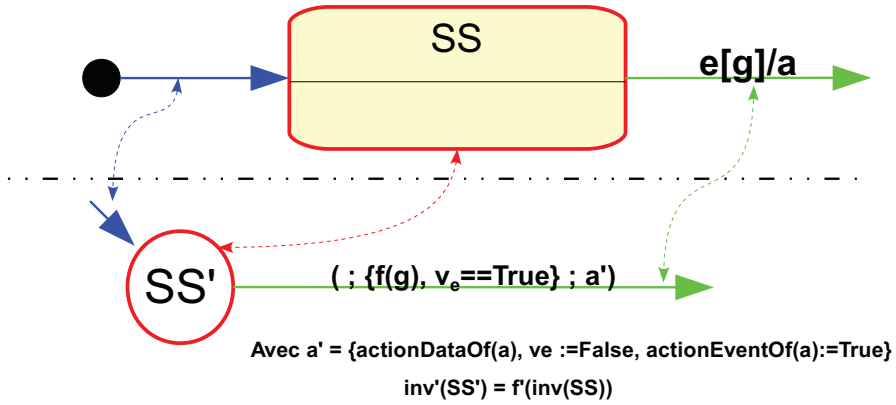

FIGURE 4.12: Nœud basique en tant qu'état initial

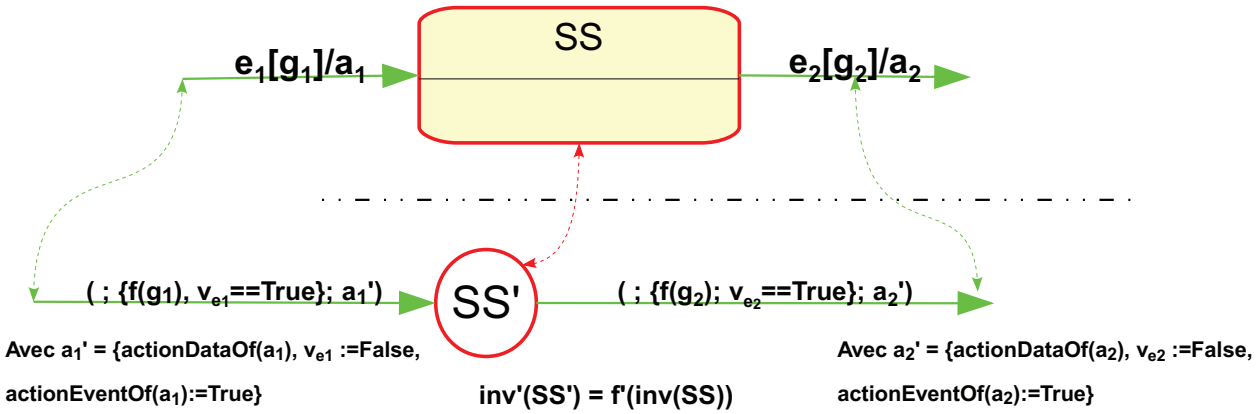

FIGURE 4.13: Nœud basique en tant qu'état ordinaire

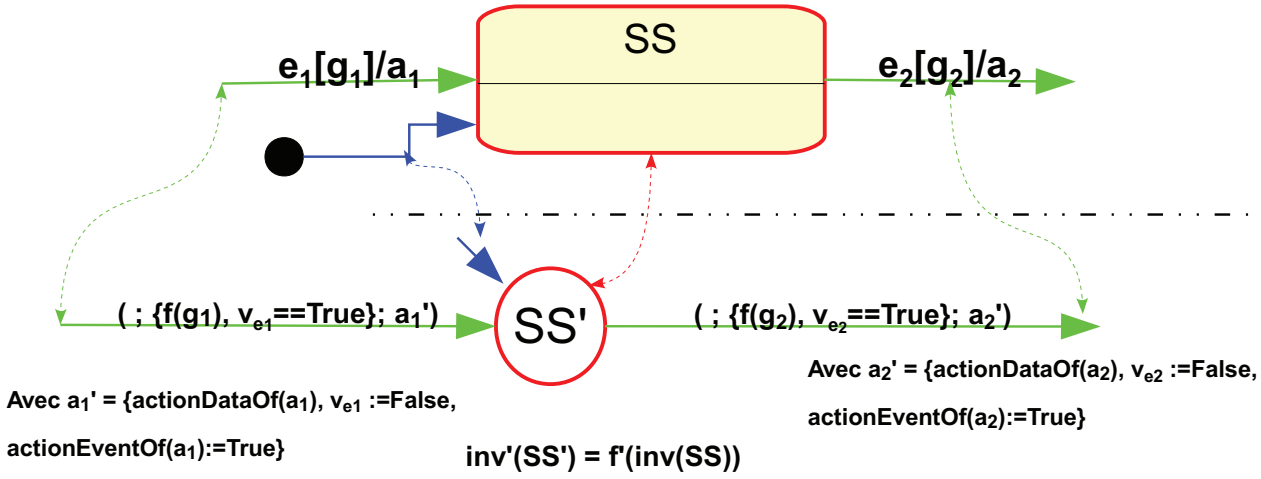

FIGURE 4.14: Nœud basique en tant qu'état initial et ordinaire

pratique, chaque transition en UML SM est traduite en une ou plusieurs transitions AT, selon plusieurs scénarios : (1) une transition qui pointe sur un nœud basique, un nœud OR ou un nœud AND, (2) une transition qui sort d'un nœud basique, un nœud OR ou un nœud AND et (3) une transition qui traverse la frontière. Ces scénarios sont détaillés tout au long de la présentation des différents motifs de traduction.

Le premier scénario est celui d'une transition dont la source et la cible sont deux nœuds basiques. Soit une transition SM, t_{sm}, dont la source (resp. la destination) est s_1 (resp. s_2) (s_1 et s_2 sont deux nœuds en SM). Soient e, g et a respectivement l'événement déclencheur de t_{sm}, la garde de t_{sm} et l'action de t_{sm}. La traduction de t_{sm} génère une transition AT t_{at} avec la source (resp. la destination) de t_{at} est s'_1 (resp. s'_2) (s'_1 et s'_2 sont obtenus respectivement de la traduction de s_1 et s_2). Ainsi, l'étiquette, la garde et l'action de t_{at} sont définies comme suit :

- L'étiquette de t_{at} est celle du nom de l'événement e de la transition t_{sm}.
- La garde de t_{at} est composée de deux parties : (1) une traduction de la garde g de la transition t_{sm} et (2) une condition sur la variable booléenne associée à l'événement e (test si la valeur de la variable est **True**).
- L'action de t_{at} est composée par : (1) l'action $actionDataOf(a)$ du t_{sm}, (2) ExitAction de s_1, (3) EntryAction de s_2, (4) mettre à **False** la variable associée à e et (5) mettre à **True** les variables associées aux événements de $actionEventOf(a)$.

4.5.3.2 Motif : état séquentiel

Le motif de traduction (figures 4.15, 4.16 et 4.17) du nœud OR est décrit comme suit : à chaque nœud OR est associé une localité en AT. Un nœud OR est un nœud de type *"UML::State"* et dont le nombre de régions est égal à un (*self.region->size()=1*). Concrètement, un nœud OR se traduit en une localité en AT de type *"AT::Location"*. Soit r la région fille du nœud OR, les nœuds fils de OR sont traduits de la même façon, selon leurs types, le motif de traduction approprié est appliqué. La traduction de ces nœuds fils génère un automate AT^r.

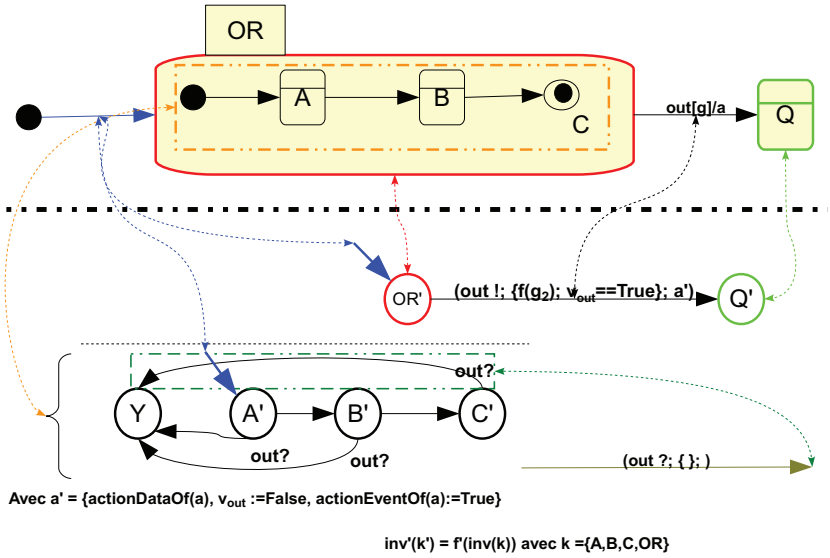

FIGURE 4.15: Nœud séquentiel en tant qu'état initial

Comme nous l'avons expliqué précédemment, la "région" fille *self.region* (de type *UML::Region*) du nœud OR est associée à l'automate AT^r. Cependant, afin de synchroniser cet automate AT^r avec l'activation et la désactivation du nœud OR, nous avons procédé comme suit :

- Ajouter une localité ℓ^r à l'automate AT^r et mettre cette localité avant la localité associée à l'état initial de la région. Par exemple, dans la figure 4.16, la localité Y est insérée avant la localité A' associée à l'état par défaut qui est A.
- Pour synchroniser l'activation de l'automate AT^r et le nœud OR, l'idée est d'associer à toute transition rentrante au nœud OR deux transitions : (1) une première transition qui

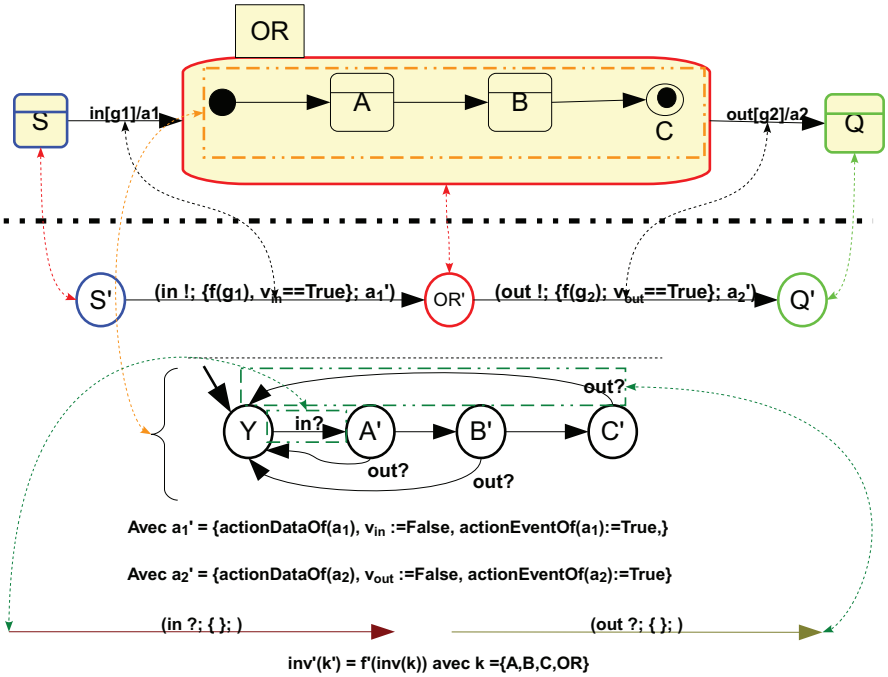

FIGURE 4.16: Nœud séquentiel en tant qu'état ordinaire

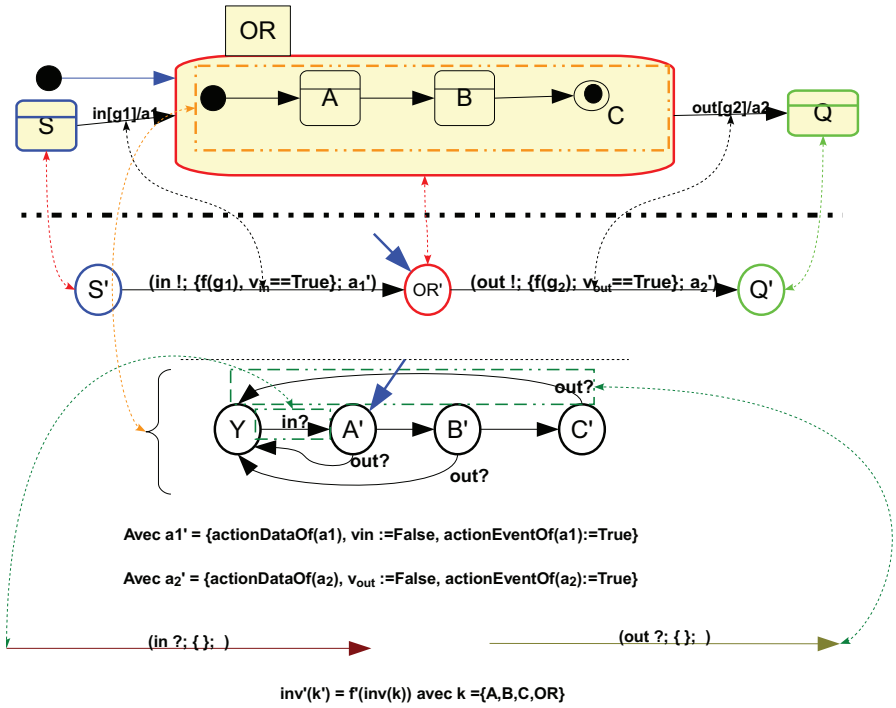

FIGURE 4.17: Nœud séquentiel en tant qu'état initial et ordinaire

pointe sur la localité associée au nœud OR et (2) une deuxième transition, étiquetée par le même événement pour synchroniser l'automate, qui sort de ℓ^r et pointe sur la localité associée à l'état par défaut dans la région à transformer. Par exemple, dans la figure 4.16, la transition t entrante au nœud OR est associée à (1) la transition t' entrante à la localité OR' et (2) à la transition t'', étiquetée avec le même événement, allant de Y vers A', étant donné que A' est la traduction de l'état par défaut (A) de la région. t' est obtenue en appliquant la procédure donnée en section 4.5.3.1 alors que t'' est une simple

transition sans garde et sans action étiquetée avec le même événement que t.

– Pour synchroniser la désactivation de l'automate AT^r et le nœud OR, l'idée est d'associer à toute transition sortante du nœud OR plusieurs transitions : (1) une transition sortante de la localité associée au nœud OR et (2) pour toutes les localités de l'automate associée à la région du nœud OR ($AT^r \backslash \{\ell^r\}$), une transition sortante, étiquetée par le même événement, dont la cible est la localité ℓ^r. Par exemple, dans la figure 4.16, la transition tt sortante du nœud OR est associée à (a) la transition tt' sortante de la localité OR' et (2) toutes les transitions tt''_i, étiquetée avec le même événement, allant de A', B' et C' vers Y. tt' est obtenue en appliquant la même procédure donnée dans la section 4.5.3.1, alors que les tt''_i sont des simples transitions avec une garde identique à la garde de tt', étiquetées par le même événement que tt et ont comme action l'ExitAction de nœud associé à la localité source de chaque transition.

Comme pour les motifs de traduction de nœud basique, le nœud OR peut avoir trois cas d'utilisation et par conséquent trois motifs de traduction. Ces cas possibles sont : (1) nœud OR en tant qu'état initial (figure 4.15) ou en tant qu'état ordinaire (figure 4.16) ou les deux en même temps (figure 4.17).

4.5.3.3 Motif : état orthogonal

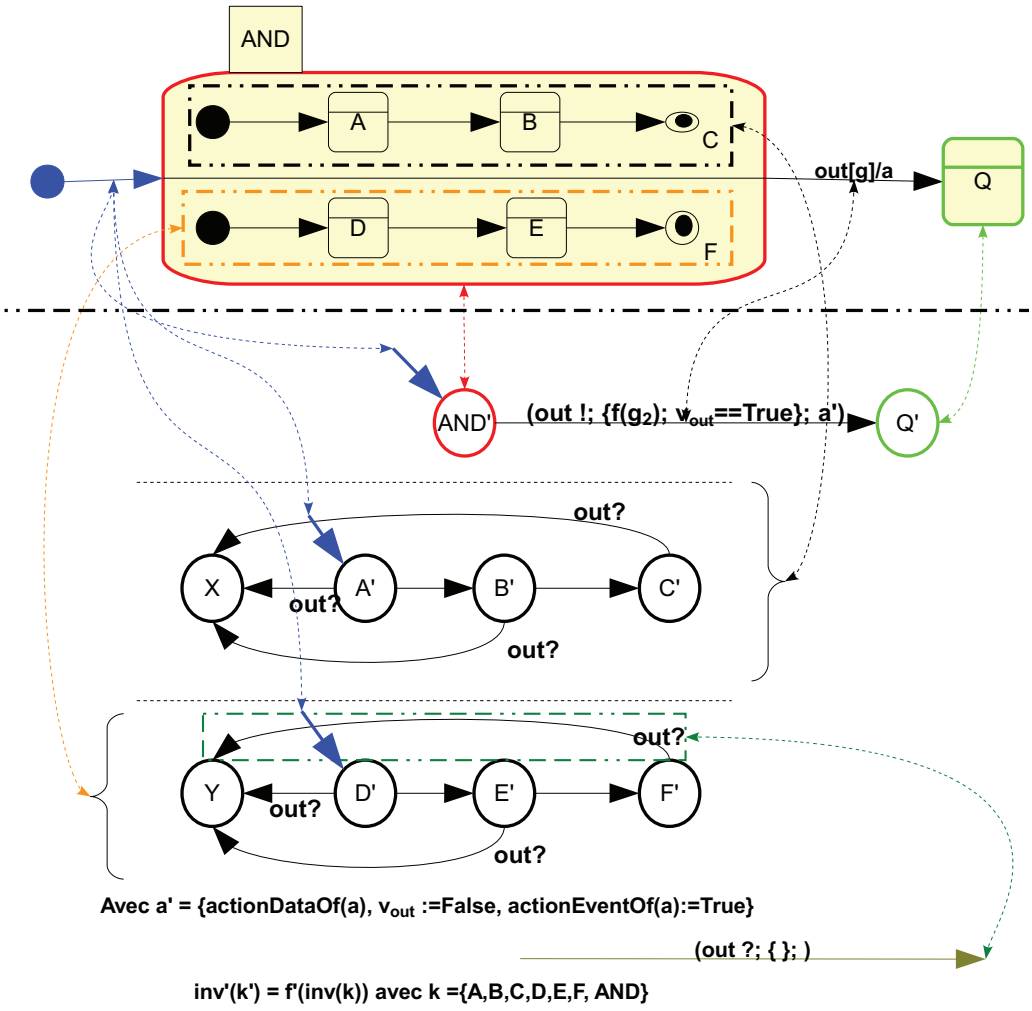

FIGURE 4.18: Nœud orthogonal en tant qu'état initial

Comme on peut le noter sur les figures 4.18, 4.19 et 4.20, les motifs de traduction des nœuds AND sont une généralisation des motifs de traduction des nœuds OR. Un nœud AND est un nœud de type *"UML::State"* dont le nombre de régions est au moins deux (*self.region->size()>1*). Comme pour les autres motifs de traduction, les règles de traduction présentent tous les cas possibles pour un nœud AND : en tant qu'état initial (figure 4.18) ou en tant qu'état ordinaire (figure 4.19) ou les deux en même temps (figure 4.20).

Il est important de préciser que la démarche présentée précédemment pour traduire la région fille d'un état OR et pour synchroniser le nœud OR avec l'automate associé à sa région fille, est utilisée de la même manière pour les nœuds AND. En effet, pour chaque région r_i fille appartenant à *self.region* (de type *UML::Region*) du nœud AND donné, nous appliquons la même procédure utilisée pour générer un automate A^{r_i}. Ensuite, pour chaque transition entrante au (respectivement sortante du) nœud AND, nous appliquons la démarche définie précédemment pour toutes les régions r_i du nœud AND.

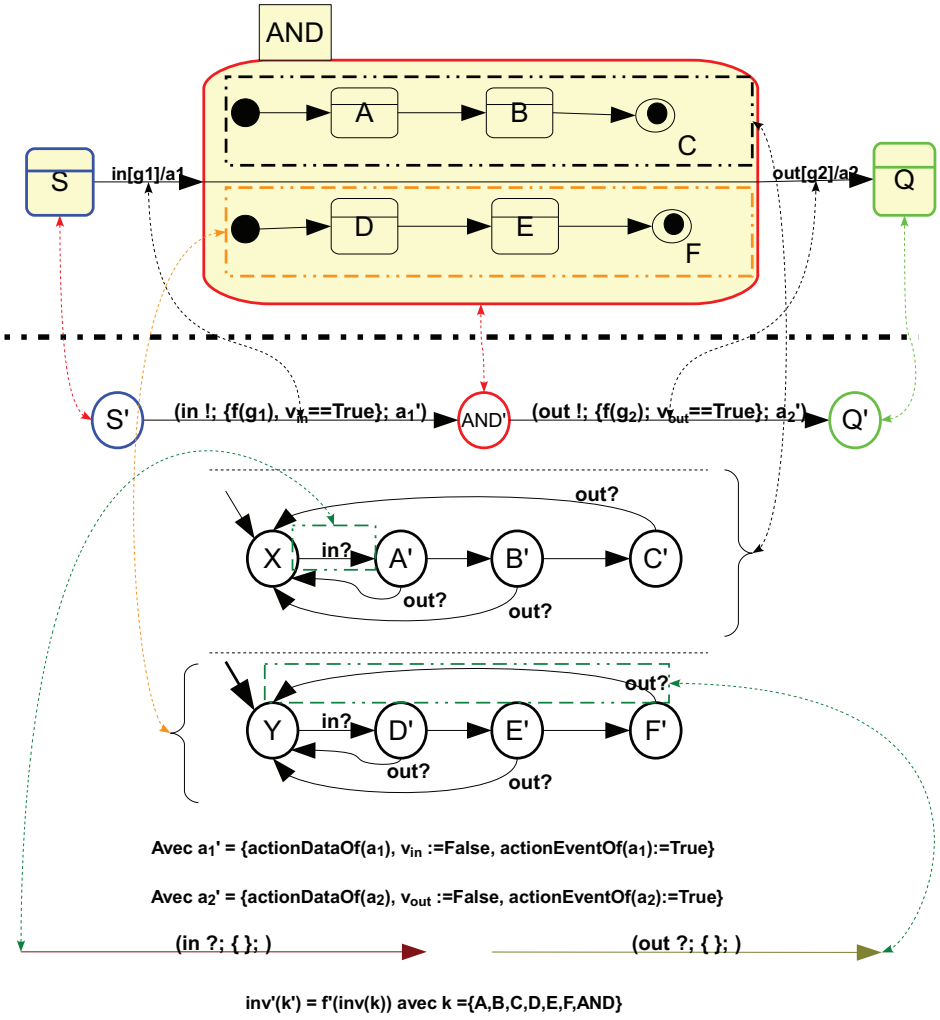

FIGURE 4.19: Nœud orthogonal en tant qu'état ordinaire

4.5.3.4 Nœud "final"

Le nœud "Final" n'est qu'une sous-classe de la classe *UML::State*. La spécification des SM (figure 4.10) définit le nœud *Final* comme un nœud basique sans l'*Entry Action*, l'*Exit Action*, et le *Do Activity* et sans transitions sortantes. L'algorithme de transformation associe à tout

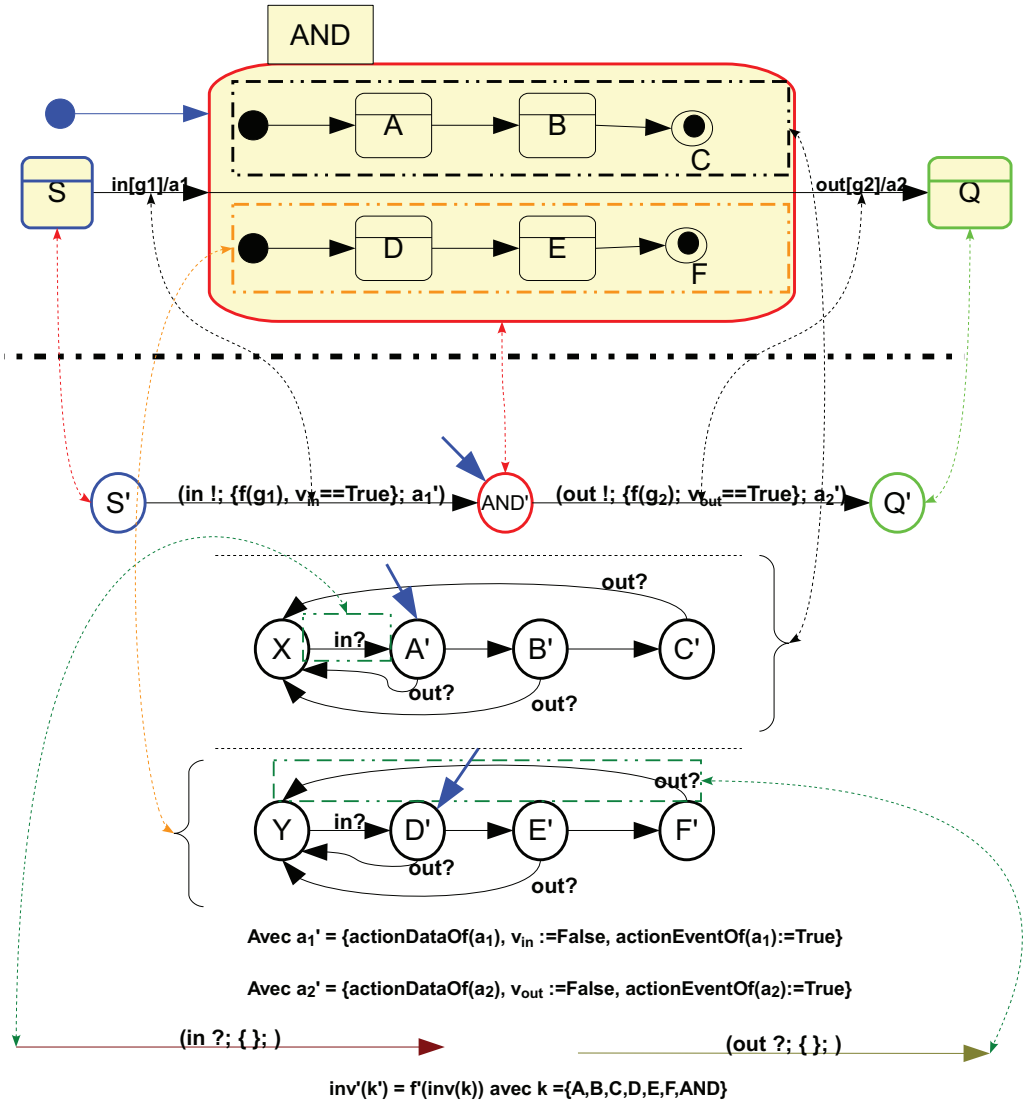

FIGURE 4.20: Nœud orthogonal en tant qu'état initial et ordinaire

nœud de type Final *self.oclIsTypeOf(UML::FinalState)* une localité *AT::Location* en AT obtenue en appliquant le motif de traduction de "nœud basique" présenté en section 4.5.3.1.

4.5.3.5 Transition

Par définition, une transition UML (*UML :: Transition*) est définie comme une relation (ou un lien) entre deux nœuds dans un diagramme SM. En raison de la notion de hiérarchie dans les SM, on peut distinguer deux cas d'utilisation possibles pour les transitions. Le premier cas est quand la transition traverse une ou plusieurs frontières d'états composites en allant de l'état source vers l'état cible (en anglais inter-level transition). En d'autres termes, la source et la destination de la transition appartiennent à deux régions différentes : *self.source.region* ≠ *self.target.region*. Le deuxième cas est quand la source et la destination de la transition appartiennent à la même région (*self.source.region* = *self.target.region*). Ce dernier cas est facile et a été traité implicitement ci-dessus, quand nous avons présenté les transitions entrantes et sortantes dans les motifs de traduction . Ci-dessous, nous traitons le premier cas, ce quand *self.source.region* ≠ *self.target.region*.

La figure 4.21 présente graphiquement le motif de traduction de 3 transitions dont la source

et la destination, pour chacune, ne sont pas dans la même région. En effet, comme le montre la figure 4.21, à toute transition "inter-level" (*UML :: Transition*) on associe plusieurs (n) transitions AT (*AT :: Transition*). Le nombre "n" dépend du type du nœud source, du type du nœud cible ainsi que de la région contenant le nœud source et celle du nœud destination.

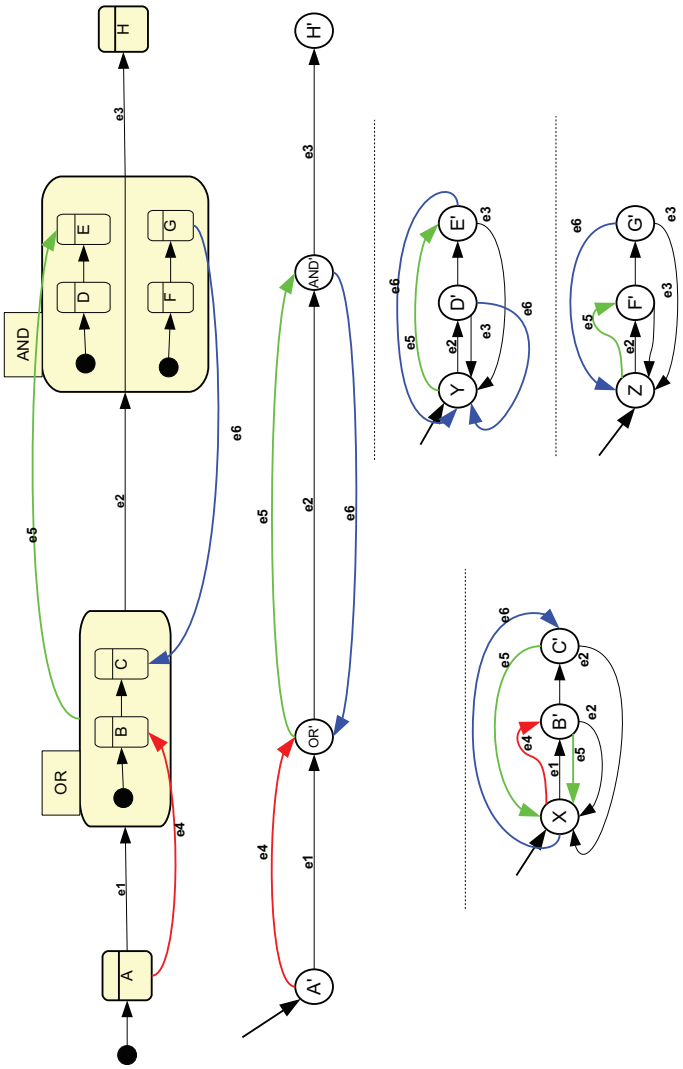

FIGURE 4.21: Traduction de la Transition "Inter-Level"

Une transition peut être composée par un ou plusieurs segments (fragments). Différents pseudo-nœuds (*UML::Pseudostate*) peuvent être utilisés pour structurer les transitions dans une machine à états. La spécification SM définit 10 types de pseudo-nœuds qui sont : l'état initial, l'état historique plat, l'état historique profond, la jonction, le choix, la terminaison, le point d'entrée, le point de sortie, le débranchement et la jointure. La version actuelle de l'algorithme de transformation que nous proposons ne traite que l'état initial (*self.kind = 'initial'*), la jonction (*self.kind = 'junction'*) et le choix (*self.kind = 'choice'*). Pat ailleurs, nous comptons intégrer les autres pseudo-nœuds (les états historiques, débranchement et jointure) dans notre transformation à court terme. La traduction du pseudo-nœud "état initial" a été montrée dans les motifs de traduction de trois types de nœuds présentés précédemment. En effet, le pseudo-nœud état initial est utilisé pour pointer l'état par défaut (ou initial) dans une région SM. Cependant, la transformation du pseudo-nœud "choix" et du pseudo-nœud "jonction" sera présentée graphiquement à travers les figures 4.22 et 4.23, respectivement.

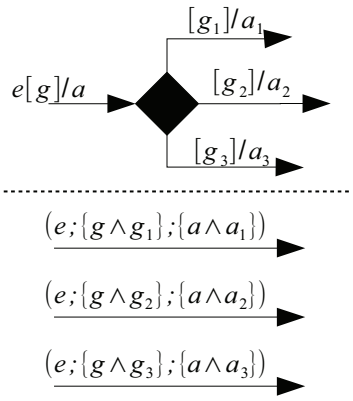

FIGURE 4.22: Traduction de pseudo-nœud "Choix"

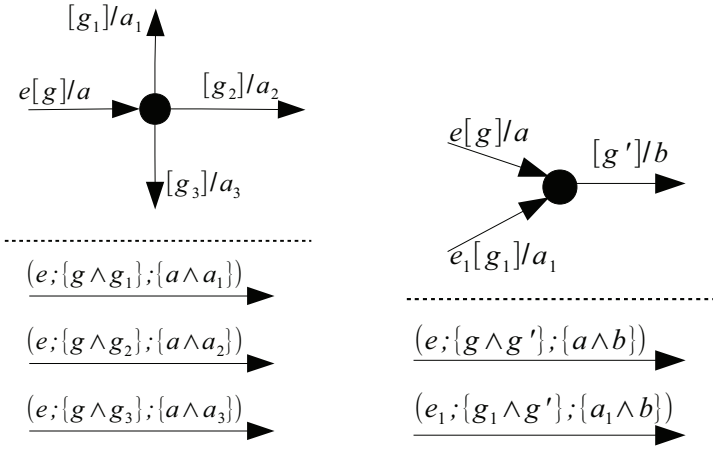

FIGURE 4.23: Traduction de pseudo-nœud "Jonction"

Remarque 4.11 *Soit n (respectivement m) le nombre de segments entrant à (resp. sortant de) un pseudo-nœud (choix ou jonction). Les transitions AT associées à ces pseudo-nœuds correspondent à l'ensemble de toutes les combinaisons des n et m segments. Ainsi, le nombre de transitions AT générées est égal à* $(n \times m)$.

Une transition peut être soit "simple", soit "composée" de plusieurs segments séparés par des pseudo-nœuds "choix" ou "jonction". En pratique, chaque fragment peut avoir une garde et une action alors qu'un seul segment (le premier segment d'une transition) peut être étiqueté par un événement déclencheur. La spécification des SM définit deux scénarios différents pour les pseudo-nœuds "jonction" et "choix". Pour la "jonction", la machine à états évalue les gardes de tous les chemins possibles avant de choisir un chemin à franchir. Cette évaluation est dite statique car elle se fait avant d'activer le premier segment. Pour le cas d'un pseudo-nœud "choix", l'évaluation du chemin est dite dynamique. En effet, dans ce cas, la machine à états évalue le premier segment. Une fois évalué, ce segment est activé sans tenir compte du segment suivant et la machine à états passe au pseudo-nœud choix. Ensuite dans ce dernier, est effectuée une évaluation dynamique des conditions des gardes des tous les segments sortants. Cette évaluation est dite dynamique parce qu'elle prend en compte, lors de l'évaluation, les changements provoqués par le premier segment (effets des actions associées à ce segment).

Remarque 4.12

1. *Nous ne prenons pas en compte l'aspect dynamique du pseudo-nœud "choix". En effet, afin de garder la nature dynamique du pseudo-nœud "choix" il fallait l'associer à une localité en AT. Ainsi, cette nouvelle localité n'est pas associée à un état réel dans la machine à états mais plutôt à un pseudo-état. Or dans les SM, un pseudo-état ne peut pas être considéré comme un état. Ainsi, cette association entre une localité AT et un pseudo-état SM rendra impossible la démonstration de la bisimulation forte entre les deux modèles (SM et AT).*

2. *Dans le cas où plusieurs transitions sont étiquetées par le même symbole, la politique des SM accordent aux transitions provenant d'un sous-état une priorité supérieure à celles provenant de l'un de ses états père. L'algorithme de transformation dans sa version actuelle ne prend pas en compte cette politique de priorité.*

4.6 State machine vers automate temporisé : préservation du comportement à travers la transformation

Dans le cadre de nos travaux nous avons retenu la bisimulation temporelle forte, entre les diagrammes SM et l'automate temporisé généré, pour l'étape de validation de la transformation. La présentation de cette preuve fera l'objet de cette section. Un aperçu de processus global de cette démonstration est donné en figure 4.24.

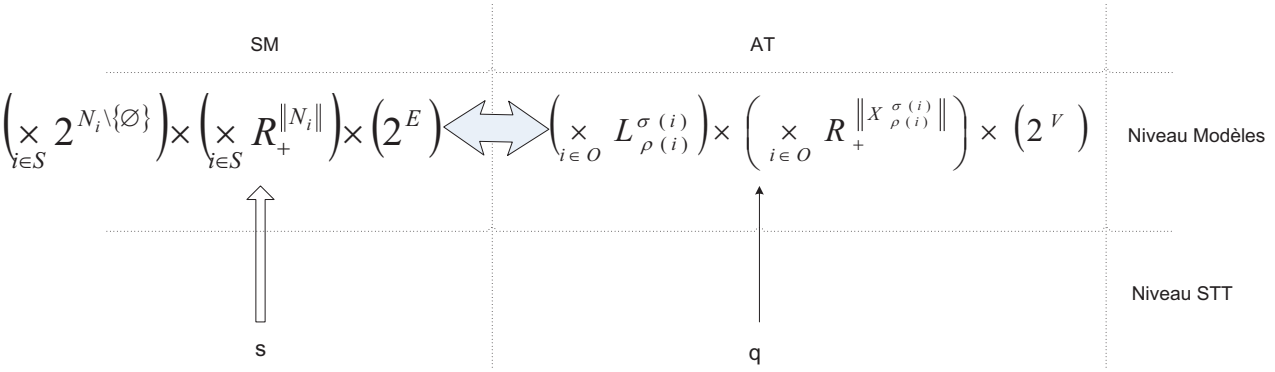

FIGURE 4.24: Schéma de démonstration

Soient un système de machines à états SMS $= \langle \mathbb{S}, \mathbb{E} \rangle$ et $SA = \langle A, \mathbb{C}, V \rangle$ un système d'automates où

- $\mathbb{S} = \{ s_i \mid s_i = (N_i, R_i, H_i, Tr_i, I_i, O_i, C_i, root_i, Default_i, Inv_i) \}$, avec $i \in [1, n]$, est l'ensemble des machines à états où $n = ||\mathbb{S}||$ et $\mathbb{E} = \bigcup_{i \in [1,n]} O_i \cup I_i$.
- $\bigcup_{i \in \mathbb{S}} R_i = \mathcal{O}$ est l'ensemble de toutes les régions du système de système SMS.
- $\sigma(i \in \mathcal{O})$ un fonction qui retourne le système SM associé à i.
- $\rho(i \in \mathcal{O})$ un fonction qui retourne la région du SM $\sigma(i)$ associé à i.
- A est un ensemble d'automates A_i, $i \in \mathcal{O}$, que nous écrivons plutôt $A^{\sigma(i)}_{\rho(i)} = (L^{\sigma(i)}_{\rho(i)}; \ell^{\sigma(i)}_{0_{\rho(i)}}; X^{\sigma(i)}_{\rho(i)}; T^{\sigma(i)}_{\rho(i)}; Inv^{\sigma(i)}_{\rho(i)}; G^{\sigma(i)}_{\rho(i)}; C^{\sigma(i)}_{\rho(i)}; \mathbb{R}^{\sigma(i)}_{\rho(i)}; \mathbb{T}^{\sigma(i)}_{\rho(i)}; \mathbb{F}^{\sigma(i)}_{\rho(i)}; Sync^{\sigma(i)}_{\rho(i)}; \Sigma^{\sigma(i)}_{\rho(i)})$ afin d'identifier à partir du quel SM et de quelle région du SM l'automate a été crée.
- Afin de simplifier la lecture de la preuve, nous utilisons les raccourcis sont :
  1. $Rof(n \in N) = r \Leftrightarrow r \to n$.
  2. $\omega(i, j) = x \Leftrightarrow (\sigma(x) = i \wedge \rho(x) = j)$.

Soient SSMS $= (S, s_0, \longrightarrow_{sm}, \Sigma_{sm})$ et SA $= (Q, q_0, \longrightarrow_{at}, \Sigma_{at})$ les sémantiques respectives de SMS et A.

Soit $R \subseteq S \times Q$, la relation reliant un état $s \in S$ à un état $q \in Q$ ssi s et q sont la représentation sous forme de TTS de deux configurations $\Uparrow(s)$ et $\uparrow(q)$ telles que $\uparrow(q)$ est relié à $\Uparrow(s)$ par la transformation. Cela s'écrit formellement comme suit :

$$(s,q) \in R \Leftrightarrow \begin{pmatrix} \Uparrow(s) = (\mathcal{C}_s, \gamma, \lambda) \wedge \uparrow(q) = (\mathcal{E}_q, \gamma', \lambda') & \wedge \\ \begin{pmatrix} \mathcal{C}_s \equiv \mathcal{E}_q & \wedge \\ \gamma \equiv \gamma' & \wedge \\ \lambda \equiv \lambda' & \end{pmatrix} & \end{pmatrix}$$

$$\text{avec } \mathcal{C}_s \equiv \mathcal{E}_q \Leftrightarrow \begin{pmatrix} (\forall i \in \mathbb{S})(\forall \alpha \in \widehat{\mathcal{C}}_s(i))(\exists x \in \omega(i, Rof(\alpha)))\widehat{\mathcal{E}_q}(x) = NtoL(\alpha) & \wedge \\ (\forall i \in \mathcal{O})(\exists \alpha \in \widehat{\mathcal{C}}_s(\sigma(i)))(\rho(i) \to \alpha \wedge (\widehat{\mathcal{E}_q}(i) = NtoL(\alpha) \vee \widehat{\mathcal{E}_q}(x) = RtoL(\rho(i)))) & \end{pmatrix}$$

$$\text{et } \gamma \equiv \gamma' \Leftrightarrow \left((\forall i \in \mathbb{S})(\forall j \in R^i)(\forall k \in N^i)(\widetilde{\widetilde{\gamma}(i)}(k) = \widetilde{\gamma'(\omega(i,j))}(NtoX(k))) \right)$$

La relation R découle directement de l'algorithme de transformation défini précédemment. Elle exprime le fait que si deux états $s \in SSMS$ et $q \in SA$ sont liés par R alors leurs configurations sont équivalentes :

- équivalence entre les configurations nodales : pour chaque SM i, les nœuds de i sont traduits en des localités AT, $\widetilde{\mathcal{E}_q}(x) = NtoL(\alpha)$. De la même manière, chaque localité AT est associée soit à un nœud soit à une région SM $\widetilde{\mathcal{E}_q}(i) = NtoL(\alpha) \vee \widetilde{\mathcal{E}_q}(x) = RtoL(\rho(i))$,
- équivalence entre les configurations des horloges : la valuation des horloges associées aux nœuds formant la configuration nodale $\mathcal{C}_s$ est la même que la valuation des horloges associées à la traduction de $\mathcal{C}_s$,
- égalité de la configuration événementielle, $\lambda \equiv \lambda'$.

Le schéma général de la démonstration prouvant que R est une bisimulation temporelle forte repose sur une démarche inductive :

- Montrer que les états initiaux sont liés par traduction et sont "localement" équivalents
- En supposant que la relation R est une bisimulation forte jusqu'à l'étape n, montrer que R est une bisimulation forte pour l'étape $n+1$

Dans la suite, nous commençons par démonter l'équivalence des états initiaux en section 4.6.1, ensuite nous démontrons l'étape d'induction en section 4.6.2.

Remarque 4.13 *1. L'ensemble des étiquettes utilisées dans les diagrammes SM, noté Σ_{sm}, est le même que l'ensemble des étiquettes utilisées dans les ATs, noté Σ_{at}. Dans la suite, nous notons Σ_{sm} et Σ_{sm} par le même symbole : Σ.*

2. Dans la suite nous utilisons la notation $\ell \xrightarrow{a}$ pour noter que l'on peut tirer une transition étiquetée par a à partir de la location ℓ.

4.6.1 Base d'induction

Soient $\Uparrow(s_0) = (\mathcal{C}_0, \gamma_0, \varnothing)$ et $\uparrow(q_0) = (\mathcal{E}_0, h_0, \varnothing)$.

▶ Comme on associe à chaque nœud d'un SM une horloge d'automate, on a $(\forall i \in \mathbb{S})(\forall j \in R^j)(\widetilde{\widetilde{\gamma_0}(i)}(j) = 0 = \widetilde{h_0}(\omega(i,j)))$. Plus précisément, toutes les horloges sont à zéro dans les deux sémantiques.

▶ Supposons que l'on puisse tirer une transition étiquetée par a : $s_0 \xrightarrow{a}_{sm}$, on a alors

$$(\exists i \mathbb{S})(\exists e = (n, in, c, out, n') \in T_{r_i}) \begin{pmatrix} n \in \widetilde{\mathcal{C}_0}(i) & \wedge \\ \mathcal{L}_i(e) = a & \wedge \\ in = \varnothing & \wedge \\ n' \in \widetilde{\mathcal{C}'_0}(i) & \end{pmatrix}.$$

Quelle que soit la forme de la garde c, puisque toutes les horloges sont à zéro, elles respectent nécessairement tout type de garde (aucune garde ne peut demander à une horloge d'avoir une valuation négative), donc aucune contrainte sur c n'est nécessaire pour le tir de e.

D'après l'algorithme de transformation, $(\exists t = (\ell, a, \ell') \in T^i_{Rof(n)})$

$$\left(\begin{array}{cc} \ell = NtoL(n) & \wedge \\ Sync^{i}_{Rof(n)}(t) = (lbl(e),!) & \wedge \\ C^{i}_{Rof(n)}(t) = \varnothing = in & \wedge \\ \ell' = NtoL(n') & \end{array}\right) \quad \textbf{(I)}$$

En effet, l'étiquette de synchronisation est apposée sur la transition donc la location source est la traduction du nœud source de la transition SM traduite. Or $src \in {}^{\bullet}\mathcal{I}^{tgt}_{src}$, donc l'existence d'une transition étiquetée par ! est assurée.

Par conséquent, **(I)** respectant les conditions de tir par les automates, on a $q_0 \xrightarrow{a}_{at}$.

▶ Maintenant, regardons ce qu'il en est pour une transition d'écoulement de temps :

$s_0 = (\mathcal{C}_0, \gamma_0, \varnothing) \xrightarrow{\theta\in\mathbb{R}^+}_{sm} \Rightarrow (\forall i \in \mathbb{S})(\forall n \in \widehat{\mathcal{C}_0}(i))(\widehat{\gamma_0}(i) + \theta \vDash Inv_i(n))$

D'après l'algorithme de transformation, tous les invariants SM sont transformés en invariants AT correspondants, donc $(\forall i \in \mathcal{O})(\widehat{h_0}(i) + \theta \vDash Inv^i(\widehat{\mathcal{E}_0}(i)))$. $\widehat{\mathcal{E}_0}(i)$ donne la location active de l'état initial pour l'automate i et $\widehat{h_0}(i) + \theta$ est le vecteur d'horloge $\widehat{h_0}(i)$ décalé de θ unités de temps vers le futur. On a donc $q_0 \xrightarrow{\theta\in\mathbb{R}^+}_{at}$.

Nous avons montré que $s_0 \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm} \Rightarrow q_0 \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at}$.

Il reste à monter que $q_0 \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at} \Rightarrow s_0 \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm}$.

▶ Soit $q_0 = (\mathcal{E}_0, h_0, \varnothing) \xrightarrow{a}_{at}$, par définition, on a $(\exists i \in \mathcal{O})(\exists t = \ell \xrightarrow{a}_{at} \ell' \in T^{\sigma(i)}_{\rho(i)})(\exists c \in$

$$\mathbb{C})\left(\begin{array}{cc} \ell = \widehat{\mathcal{E}_0}(i) & \wedge \\ Sync^{\sigma(i)}_{\rho(i)}(t) = (c,!) & \wedge \\ C^{\sigma(i)}_{\rho(i)} = \varnothing & \end{array}\right)$$

D'après la traduction, $(\exists e = (n, in, c, out, n') \in T_{r_{\sigma(i)}})\left(\begin{array}{cc} NtoL(n) = \ell & \wedge \\ \mathcal{L}_i(e) = a & \wedge \\ in = \varnothing & \end{array}\right)$

d'où $s_0 \xrightarrow{a}_{sm}$

▶ Soit $q_0 \xrightarrow{\theta\in\mathbb{R}^+}_{at} \Rightarrow (\forall i \in \mathcal{O})(\widehat{h_0}(i) + \theta \vDash \widehat{\mathcal{E}_0}(i))$. Or d'après la traduction toute location est issue d'une bijection $NtoL$, donc $(\forall i \in \mathcal{O})(\exists n \in N^{\sigma(i)})(NtoL(n) = \widehat{\mathcal{E}_0}(i) \wedge \widehat{\gamma_0}(\sigma(i)) + \theta \vDash Inv_i(n))$.

D'où $s_0 \xrightarrow{\theta\in\mathbb{R}^+}_{sm}$.

4.6.2 Étape d'induction

Hypothèse d'induction : on suppose que $sRq \wedge s \xrightarrow{a}_{sm} s' \wedge q \xrightarrow{a}_{at} q'$.

▶ $s \xrightarrow{a\in\Sigma}_{sm} s' \Rightarrow \Uparrow(s) = (\mathcal{C}_s, \gamma, \lambda) \wedge \Uparrow(s') = (\mathcal{C}'_s, \gamma, \lambda')$.

De la sémantique des SM, on a

$$(\exists i \in \mathbb{S})(\exists e = (n, in, c, out, n') \in T_{r_i})\left(\begin{array}{cc} n \in \widehat{\mathcal{C}_s}(i) & \wedge \\ n' \in \widehat{\mathcal{C}'_s}(i) & \wedge \\ \mathcal{L}_i(e) = a & \wedge \\ in \in \lambda & \wedge \\ \lambda' = \lambda \backslash in \cup out & \wedge \\ c = (s,m) \Rightarrow \widehat{\gamma}(NtoX(n)) \quad \tilde{s} \quad m & \end{array}\right). \text{ avec}$$

$\widetilde{at\_least} = \geqslant; \widetilde{at\_most} = \leqslant; \widetilde{at} = =; \widetilde{lower} = <; \widetilde{upper} = >$.

On a aussi, $(\forall i \in \mathbb{S})(\widetilde{\mathcal{C}_s}(i) = \widetilde{\mathcal{C}'_s}(i) \backslash {}^{\bullet}\mathcal{I}^{tgt}_{src} \cup \mathcal{I}^{\bullet tgt}_{src})$.

$q \xrightarrow{a\in\Sigma}_{at} q' \Rightarrow \uparrow(q) = (\mathcal{E}_s, h_0, v_0) \wedge \uparrow(s') = (\mathcal{E}'_s, h_0, v'_0)$.

${}^{\bullet}\mathcal{I}^{n'}_n$ est utilisé par la traduction pour calculer les états sources des transitions automates et $\mathcal{I}^{\bullet n'}_n$ est utilisé pour calculer les états destination.

$(\forall i \in \mathbb{S})(\forall r \in R_i)$, si ${}^{\bullet}\mathcal{I}^{n'}_n \cap r = \varnothing \wedge \mathcal{I}^{\bullet n'}_n = \varnothing$ alors une transition étiquetée par le label *?* est crée, si ${}^{\bullet}\mathcal{I}^{n'}_n \cap r = \{n\}$ la transition crée est étiquetée par *!*. Sinon, le canal utilisé est $lbl(e)$.

Cela signifie que toutes ces transitions seront tirées en même temps, sous l'impulsion de la transition étiquetée *!*. C'est-à-dire que dans un seul pas, toutes les locations correspondantes aux nœuds ${}^{\bullet}\mathcal{I}^{n'}_n$ sont désactivées au profit des locations correspondantes aux nœuds $\mathcal{I}^{\bullet n'}_n$.

Comme sRq, alors $\widetilde{\mathcal{C}_s} \equiv \widetilde{\mathcal{E}_q}$ et donc $\widetilde{\mathcal{C}'_s} \equiv \widetilde{\mathcal{E}'_q}$.

On a vu que $\lambda' = \lambda \backslash in \cup out$. Parmi l'ensemble des transitions franchies de manière synchrone comme décrit ci-dessus, il en est une seule qui porte l'information de garde et de condition sur les variables. C'est aussi elle qui met à jour les variables. C'est la transition étiquetée *!*. L'action de cette transition sur v est la même que celle de la transition SM sur λ. Or puisque sRq alors $v = \lambda$ et par conséquent $v \backslash in \cup out = v' = \lambda'$. Par conséquent $s'Rq'$.

Il reste à prouver que $s' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm} \Leftrightarrow q' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at}$.

▶ Supposons que $s' \xrightarrow{a}_{sm}$:

$$(\exists i \mathbb{S})(\exists e = (n, in, c, out, n') \in T_{r_i}) \left(\begin{array}{ccc} n \in \widetilde{\mathcal{C}'_s}(i) & & \wedge \\ \mathcal{L}_i(e) = a & & \wedge \\ c = (s,m) \Rightarrow \widetilde{\gamma}(i)(n) & \tilde{s} & m \quad \wedge \\ in \in \lambda' & & \end{array} \right)$$

D'après la traduction et $s'Rq'$, on a $(\exists i \in \mathcal{O})(\exists t = (\ell, a, \ell') \in T^{\sigma(i)}_{\rho(i)})$

$$\left(\begin{array}{ccc} \ell = NtoL(n) = \widetilde{\mathcal{E}'_q}(i) & & \wedge \\ Sync^{\sigma(i)}_{\rho(i)}(t) = (lbl(e), !) & & \wedge \\ c = (s,m) \Rightarrow G^{\sigma(i)}_{\rho(i)}(t) = (NtoX(n) \quad \tilde{s} \quad m) \wedge (\widetilde{h'_q}(i)(NtoX(n)) & \tilde{s} & m) \quad \wedge \\ in \subseteq v' & & \end{array} \right)$$

Cela implique que $q' \xrightarrow{a}_{at}$.

▶ Supposons que $s' \xrightarrow{\theta\in\mathbb{R}^+}_{sm}$.

$(\forall i \in \mathbb{S})(\forall n \in \widetilde{\mathcal{C}'_s}(i))(\widetilde{\gamma'}(i) + \theta \vDash Inv_i(n))$, comme on l'a vu au cas de base de l'induction, on a $(\forall i \mathcal{O})(\widetilde{h'_q}(i) + \theta \vDash Inv^i(\widetilde{\mathcal{E}'_q}(i)))$, et par conséquent $q' \xrightarrow{\theta}_{at}$.

On a donc $s' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm} \Rightarrow q' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at}$

▶ Les arguments de la preuve de $q' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at} \Rightarrow s' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm}$ sont aisément réversibles à partir du cas précédent, nous ne l'exposerons pas et donc $s' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{sm} \Leftrightarrow q' \xrightarrow{a\in\Sigma\cup\mathbb{R}^+}_{at}$, ce qui conclut la preuve ■.

4.7 Outil logiciel pour la transformation $SM \rightarrow AT$

Un outil logiciel a été développé afin de supporter la phase de transformation. Les modèles AT générés sont formatés automatiquement au format XML d'Uppaal. Ainsi, les fichiers obtenus par transformation sont directement exploitables dans l'outil de vérification Uppaal.

Pour illustrer cela, on considère un système contrôle-commande de protection automatique d'un passage à niveau dont le comportement est décrit comme suit : le système est composé de trois sous-systèmes : ST, SB et SC utilisés pour modéliser respectivement le comportement du système de détection des trains, de la barrière et finalement du contrôleur. Ces trois sous-systèmes opèrent en parallèle et se synchronisent sur les événements : Approach, Exit, GoDown, GoUp, Down et Open.

- Lorsqu'un train est en approche du croisement, *ST* envoie un signal *Approach* à *SC* et entre dans le croisement dans un délai entre 25 et 40 unités de temps. Quand un train quitte le croisement, *ST* envoie un signal *Exit* à *SC*. Le signal *Exit* est envoyé dans au moins 10 et au plus 15 unités de temps après l'entrée dans la zone de croisement.
- SC envoie un signal *GoDown* à *SB* exactement à la réception du signal *Approach* et envoie un signal *GoUp* à la réception du signal *Exit*.
- SB répond par *down* au signal *GoDown* dans 10 unités de temps, et répond par *open* au signal *GoUp* dans 10 unités de temps.

La figure 4.25 représente le modèle SM proposé pour décrire le comportement du système contrôle-commande décrit ci-dessus. Supposons que les trains sur la même direction sont suffisamment espacés. Ainsi, le contrôleur traite au plus deux trains en même temps, chacun dans une direction.

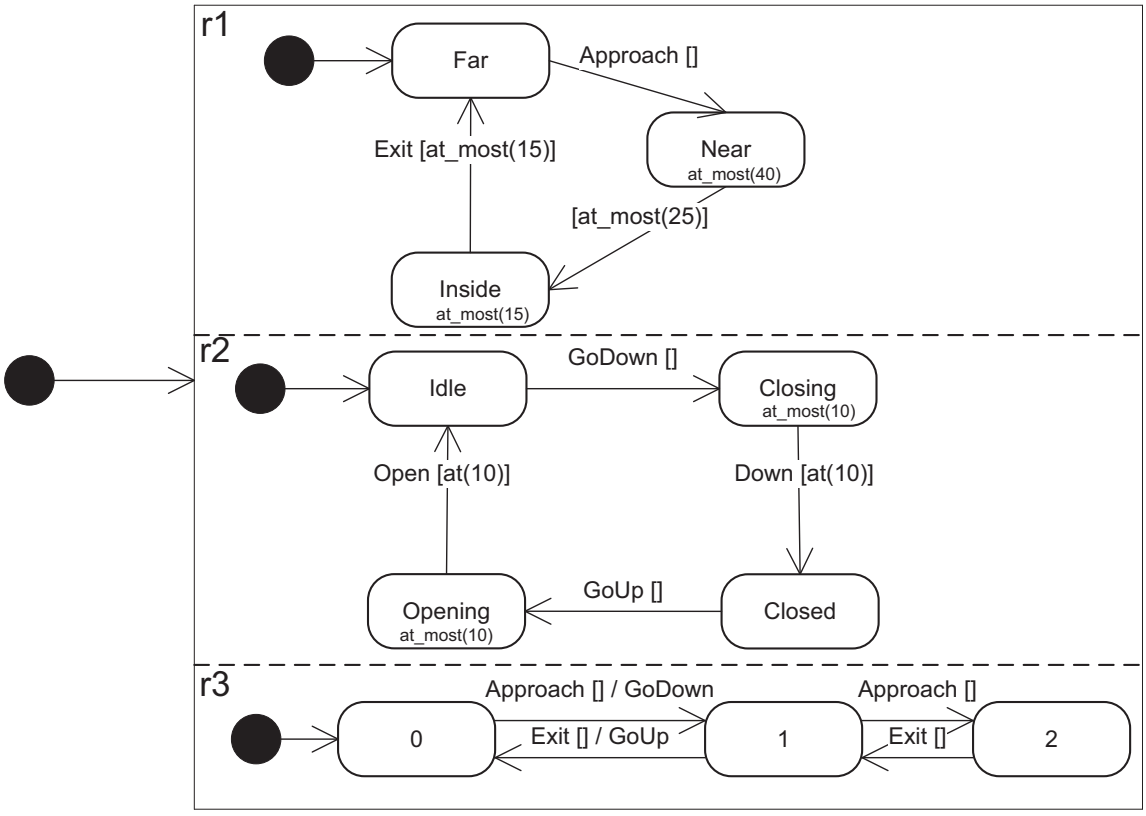

FIGURE 4.25: Modèle SM pour le système contrôle-commande du protection au passage à niveau

Sur ce modèle on applique l'algorithme de transformation que a été implémenté en *Scala*. Le résultat de transformation est formaté en fichier XML en respectant le format d'Uppaal. Le modèle graphique correspondant est donné ci-après en figures 4.26, 4.27 et 4.28.

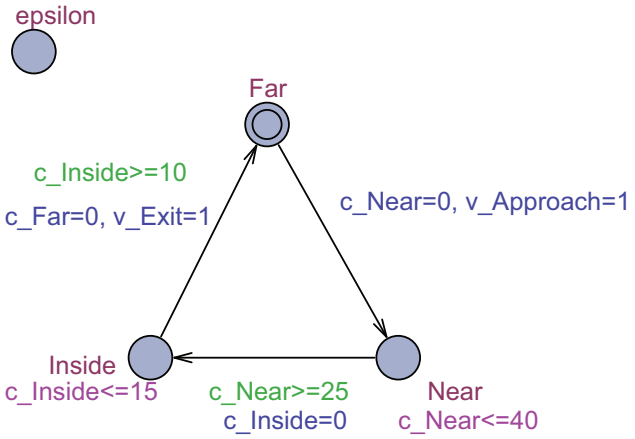

FIGURE 4.26: Modèle AT pour la région *r1*, le sous-système SD

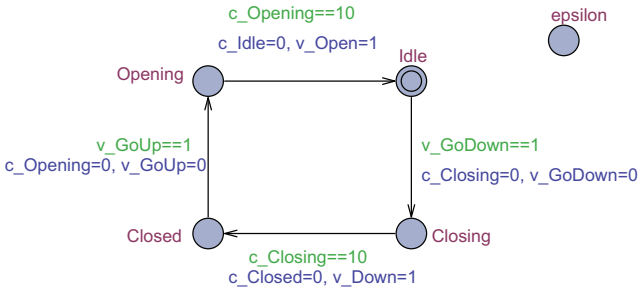

FIGURE 4.27: Modèle AT pour la région *r2*, le sous-système BP

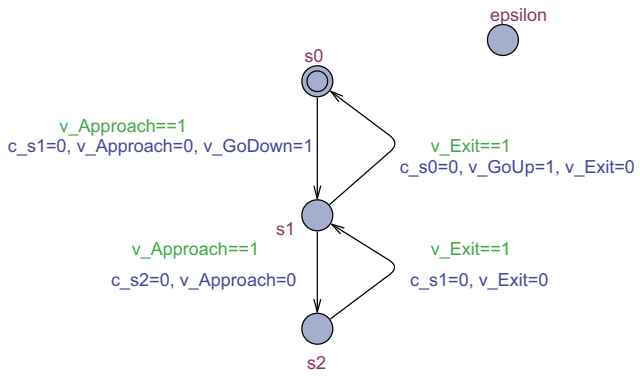

FIGURE 4.28: Modèle AT pour la région *r3*, le sous-système SC

4.8 Conclusions

Dans ce chapitre, nous nous sommes intéressés à l'étape de modélisation du comportement. Nous avons proposé une technique d'assistance afin de simplifier cette étape toute en cachant

les difficultés liées à l'utilisation des langages formels de modélisation. En effet, l'objectif est de permettre à l'utilisateur de manipuler une notation graphique assez-intuitive (dans notre cas des diagrammes SM avec des annotations temporelles) et d'en générer automatiquement des spécifications formelles en automates temporisés. Pour cela nous avons proposé un algorithme de transformation des SM vers ATs. Ensuite, nous avons proposé une démonstration de la préservation du comportement entre SM et AT à travers la transformation. En particulier, nous avons montré l'existence d'une bisimulation temporelle forte préservant la sémantique entre les systèmes en entrée et en sortie de la transformation. Cette démonstration d'équivalence comportementale est un point clé puisqu'elle permet de valider la traduction par la garantie la préservation de certaines propriétés comportementales.

La modélisation ainsi que la spécification forment les entrées pour l'étape de la vérification et validation. Cette dernière est abordée dans la troisième partie de ce manuscrit.

Troisième partie

Vérification et validation

Chapitre 5

Architecture pour la vérification et à la validation des exigences temporelles

Vérification par observateur

Résumé : *Dans cette troisième partie de ce manuscrit, nous allons nous intéresser à l'étape de vérification et validation (V&V). L'objectif de l'étape de V&V est de garantir la conformité du modèle d'implémentation par rapport à ses exigences. Des nombreuses techniques ont été proposées pour mener cette étape. Dans ce chapitre, nous commençons, tout d'abord, par faire un tour d'horizon de ces différentes techniques. Par la suite, nous détaillons notre contribution pour cette étape.*

Sommaire

5.1 Introduction

La phase de spécification permet de traduire les besoins utilisateur en spécifications exploitables. L'étape de modélisation permet de produire un modèle du système à vérifier. Les exigences ainsi que le modèle sont des entrées pour l'étape de vérification et validation. Cette étape consiste à vérifier le respect des exigences fixées par le modèle. Dans la suite de ce chapitre, nous commençons par introduire l'étape V&V en section 5.2.1. Comme nous le verrons dans la suite, différentes techniques de vérification peuvent être utilisées à différentes étapes dans le cycle de développement d'un système. Afin d'illustrer cela, nous présenterons le cycle de développement en "V" en section 5.2.2. Puis, nous passerons en revue certaines techniques de vérification : test (section 5.2.3), simulation (section 5.2.4), preuve (section 5.2.5) et model-checking (section 5.2.6). Pour chaque technique, nous donnons les concepts de base, les avantages, les inconvénients et nous la situerons dans le cycle de développement. Ensuite, nous consacrons les sections 5.3 et 5.4 à la présentation de nos contributions relatives à l'étape de vérification. Une brève conclusion est donnée en section 5.6.

5.2 Vérification et validation

5.2.1 Définitions et objectifs

L'objectif de la V&V est de révéler les fautes qui ont pu être introduites au cours des phases du cycle de développement. Pour des raisons de coût et d'efficacité, il est important de les révéler au plus tôt. Par conséquent, les techniques de V&V doivent être intégrées dès le début et tout au long du processus de développement.

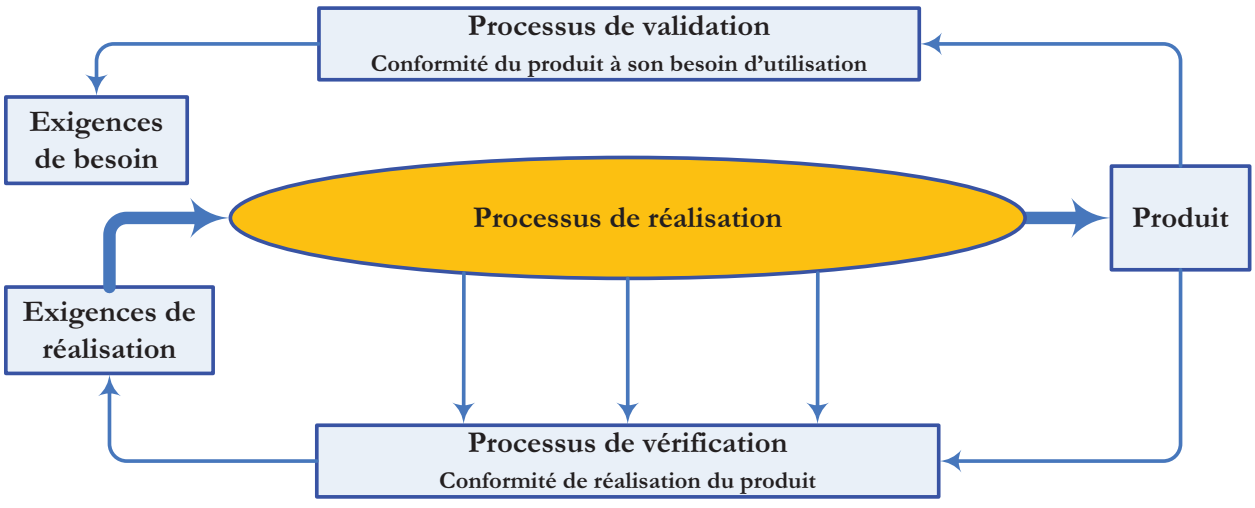

FIGURE 5.1: Vérification versus Validation (AFIS)

- La vérification a comme objectif de montrer que toutes les tâches ont été bien faites, qu'elles sont en conformité avec le plan de réalisation et qu'elles n'ont pas introduit des erreurs dans le produit (système/logiciel) en développement. La vérification est applicable à n'importe quelle étape du cycle de développement du produit. Dans la norme ISO 9000 la vérification est définie comme la ***«Confirmation par des preuves tangibles que les exigences spécifiées ont été satisfaites»***. En effet, vérifier c'est répondre à la question : *«Faisons-nous le produit correctement ?»*.

- La validation a comme objectif de montrer que le produit répond aux besoins/exigences pour lesquels il a été conçu. La norme ISO 9000 définit la validation comme une ***«Confirmation par des preuves tangibles que les exigences pour une utilisation spécifique ou une application prévue ont été satisfaites»***. En effet, valider c'est répondre à la question : *«Faisons-nous le bon produit ?»*.

En d'autres termes, la validation assure que les modèles utilisés tout au long du cycle de développement, à savoir les modèles fonctionnels, les modèles de conception et les modèles de réalisation/implémentation, sont conformes aux attentes du client ou de l'utilisateur final. Alors que la vérification assure que les modèles utilisés lors de l'étape de spécification et l'étape de conception (c'est-à-dire les modèles de spécification et les modèles de conception) sont conformes aux exigences définies. Le couplage des deux activités présente une garantie pour la non-violation des propriétés de sécurité et de bon fonctionnement du système.

Nombreuses sont les techniques de vérification/validation qu'on trouve dans la littérature et qui peuvent être répertoriées selon plusieurs critères à savoir

- Technique automatique vs technique semi-automatique (ou manuelle),
- Technique exhaustive vs technique non exhaustive,
- Technique *a Posteriori* vs technique *a Priori*.

Ces différentes classifications peuvent être facilement critiquées et mises en question. Par exemple, le test est généralement considéré comme une technique non exhaustive alors qu'on retrouve des approches exhaustives qui couplent la technique de test avec la technique de model-checking dans le but de générer des scénarios à dérouler en utilisant le model-checking. Dans la suite, nous donnons un aperçu général du processus de développement. Ensuite, nous présenterons certaines techniques utilisées pour l'étape de V&V.

5.2.2 Utilisation de techniques de vérification/validation dans le cycle de développement

Plusieurs cycles de développement ont été proposés dans la littérature comme le modèle en cascade (ou waterfall), le modèle en V, le modèle incrémental et le modèle en spirale. En pratique, les modèles de cycle de développement décrivent de manière abstraite les différentes organisations d'une chaîne de développement. Une organisation est composée généralement d'un ensemble ordonné d'étapes. Des conditions du passage d'une étape à une autre sont parfois ajoutées à l'organisation. Ces conditions peuvent être des critères de choix d'une étape, des conditions de démarrage ou de terminaison d'une étape. Les étapes présentent un découpage temporel des tâches. Parmi ces tâches on peut citer la spécification, la conception, l'implémentation ou réalisation et la V&V. L'objectif de cette section n'est pas de comparer les différents cycles de développement mais plutôt de choisir l'un d'entre eux pour illustrer l'intégration des techniques de V&V dans le cycle de développement. Pour cela, nous avons retenu le modèle en V. Ce choix trouve sa justification dans le fait que le modèle en V est incontestablement le cycle de vie le plus connu et certainement le plus utilisé [Audibert 2011].

Modèle en V : le cycle de développement en V (figure 5.2) peut être découpé en trois parties : la phase de spécification et de conception, la phase d'implémentation et finalement la phase de validation [Godary 2004] :

1. *Spécification et conception* : cette étape commence par une analyse des besoins du cahier des charges afin de proposer une spécification des fonctionnalités qui est utilisée pour la conception des architectures fonctionnelles et opérationnelles.

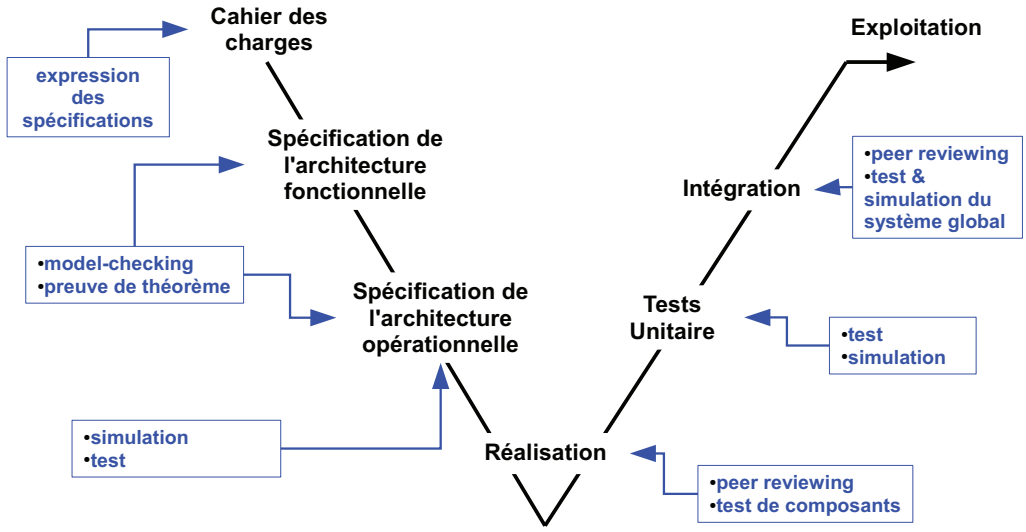

FIGURE 5.2: Vérification/Validation dans le cycle en V, inspiré de [Godary 2004]

2. *Implémentation* : en se basant sur l'architecture opérationnelle définie dans l'étape précédente, cette phase présente l'implémentation physique du produit. En effet, les différentes entités composant le futur produit sont conçues et implémentées individuellement. Ensuite, ces entités sont intégrées dans la conception du système global.
3. *Validation* : c'est la dernière étape du cycle et a comme objectif de garantir la conformité du futur produit aux exigences pré-définies. Elle est formée par deux sous-étapes : des tests sur les entités indépendantes et des tests sur le système global après intégration. L'étape de validation permet d'éliminer les erreurs de conception et/ou de réalisation ainsi que de valider l'architecture opérationnelle choisie pour le produit final.

Or, étant donné que le coût de correction d'erreurs (figure 5.3) détectées dans les phases finales de développement ou après la fin de cycle de développement est souvent bien plus important que le coût de correction à un stade primaire, c'est-à-dire au début du cycle de développement [Katoen 2008], il est judicieux de commencer les tâches de vérification le plus tôt possible dans le cycle de développement.

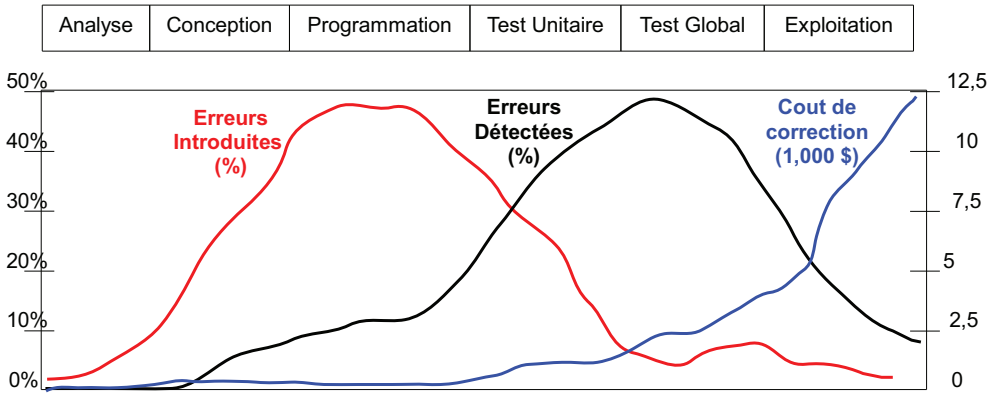

FIGURE 5.3: Erreurs dans le processus de développement, inspiré de [Katoen 2008]

Dans le cadre de nos travaux nous nous intéressons aux systèmes critiques notamment dans

le domaine du transport ferroviaire. Par conséquent, plusieurs contraintes sont associées à ce contexte de travail. En particulier, en plus de la fiabilité et de la sécurité, des contraintes comme la maîtrise du processus de développement ainsi que la réduction du coût peuvent être critiques pour de tels systèmes. Une étape indispensable pour garantir toutes ces contraintes est l'étape de V&V. Dans la suite de cette section, nous passons en revue les principales techniques de vérification de la littérature tout en pointant les avantages et les inconvénients de chacune d'entre elles.

5.2.3 Test

Le test [Beizer 1990, Beizer 1995] permet de vérifier la présence d'erreurs dans le système. Cette technique commence par définir un ensemble de vecteurs de test (en général non exhaustifs) à injecter en entrée du système à tester. À la fin de cette étape on obtient un ensemble qu'on nomme jeux de tests (ou jeux d'essais). Ensuite, pour chaque jeu de test, le produit est exécuté et les résultats obtenus sont comparés aux résultats attendus [Kissoum 2009]. Cela permet de détecter les erreurs de spécification et/ou de conception et/ou d'implémentation. Dans l'industrie, l'utilisation du test est très répandue pour la mise en place de nouveaux projets logiciels et cela représente de 30% à 50% du côut total [Katoen 2008, Kissoum 2010]. Néanmoins, cette technique ne garantit pas l'absence d'erreurs. On distingue trois catégories de construction des jeux de test à savoir le test aléatoire, le test fonctionnel ou test boîte-noire et le test structurel ou test boîte-blanche :

- **Le test aléatoire** : Dans cette catégorie le vecteur de test est sélectionné sur le domaine des entrées sans aucune stratégie de sélection (au hasard). Cette méthode est efficace dans le cas où l'utilisateur veut traiter les cas limites ou exceptionnels. Néanmoins, elle ne garantit pas une couverture de l'ensemble des entrées possible du produit.
- **Le test fonctionnel** (ou test boîte-noire) : Dans cette catégorie, on considère l'implémentation comme une boîte noire. En effet, on ne considère que la spécification des fonctionnalités du produit sans considérer sa structure interne [Beizer 1995]. Par conséquent, à l'aide de cette technique on peut vérifier toutes les fonctionnalités décrites dans la spécification du système. L'avantage majeur de cette technique est qu'on peut souvent l'appliquer très tôt dans le cycle de développement (c'est-à-dire dès qu'on définit la spécification). Néanmoins, la couverture de la totalité du domaine des entrées pour toutes les fonctionnalités entraîne l'explosion combinatoire du nombre de jeux de test.
- **Le test structurel** (ou test boîte-blanche) : Dans cette catégorie on tient compte de la structure interne du produit, pour générer les jeux de test, d'où le nom de test boîte-blanche [Beizer 1990].

Il est à noter qu'on rencontre parfois la notion de test boite grise où la structure interne du système est partiellement connue.

Beaucoup de travaux ont été proposés afin d'améliorer la mise en œuvre de cette technique. On peut citer par exemple la génération automatique de jeux de test et l'automatisation de l'oracle. Cependant, la technique de test ne permet pas d'affirmer avec certitude l'absence d'erreurs dans le produit, mais elle permet d'en trouver quelques-unes. De plus, une classe importante d'erreurs (erreurs de concurrence : l'inter-blocage ou l'exclusion mutuelle) peut-être provoquée par des enchaînements/processus très subtils et sont donc difficiles à reproduire. Cela rend le test de ces types d'erreurs extrêmement difficiles à implémenter. En plus de ces inconvénients, la technique de test présente une limite majeure qui est le coût. En effet, le test s'applique souvent après la phase de réalisation du système c'est-à-dire tard dans le processus

de développement. Par conséquent, la détection d'une erreur à ce stade entraine un retour en arrière parfois sur toutes les étapes déjà effectuées. Cela peut remettre en cause différents choix de conception et/ou d'implémentation et rend le processus de test long et coûteux. Un récapitulatif des avantages et des inconvénients est donné dans le tableau 5.1.

Avantages	Inconvénients
• outils performants, • détection des erreurs de conception et d'implémentation, • analyse dynamique.	• détection tardive des erreurs, • coût souvent élevé pour la correction des erreurs détectées, • technique non exhaustive.

Table 5.1: Avantages et Inconvénients du Test

5.2.4 Simulation

La vérification se fait par déroulement de scénarios d'exécution prédéfinis ou aléatoires qui visent à mettre en échec le modèle du système pour les propriétés souhaitées. Contrairement au test qui peut être utilisé pour des systèmes déjà opérationnels, la simulation ne peut être appliquée que sur des abstractions (modèles) du système à vérifier. Ainsi, une étape de modélisation du système en question est exigée pour l'utilisation de la simulation. Cette technique est l'une des plus utilisées dans l'industrie en raison de sa simplicité de mise en œuvre. Néanmoins, la simulation ne peut pas être exhaustive (elle ne couvre pas la totalité des comportements possibles) et ne détecte que les erreurs présentes dans les scénarios exécutés. Par exemple, pour un système comme le téléphone portable où les nombres d'exécutions possibles à chaque pas sont égaux à n. Pour un nombre m de pas, on obtient n^{m} scénarios possibles, par exemple, pour $n = 5$ et $m = 20$ on obtient $100,000,000,000,000$ scénarios possibles. Par conséquent, seule une partie des scénarios est déroulée en pratique. Cela ne garantit pas l'absence d'erreurs ou de défaillances car une grande partie des scénarios est ignorée. L'autre limite est que le modèle utilisé pour la simulation peut ne pas correspondre parfaitement au comportement du système, ce qui peut donner des résultats inappropriés. Un récapitulatif des avantages et des inconvénients est donné dans le tableau 5.2.

Avantages	Inconvénients
• outils performants, • détection des erreurs de conception, • faible coût.	• détection des erreurs au niveau modèle seulement (et non pas au niveau système), • les erreurs de réalisation ne sont pas prises en compte, • technique non exhaustive.

Table 5.2: Avantages et Inconvénients de la Simulation

5.2.5 Preuve

La preuve de théorème permet de prouver mathématiquement la correction d'un modèle. En effet, les propriétés sont représentées sous forme de théorèmes. La démonstration de ces théorèmes garantit la validation des propriétés sur le modèle à vérifier. En d'autres termes, la preuve est une démonstration mathématique (une approche syntaxique et axiomatique) qui permet de déduire certains faits à partir d'autres faits en se servant des propositions, des axiomes et des règles de déduction afin de prouver la validité des théorèmes (propriétés). La preuve peut être appliquée à toutes les étapes du processus de développement. Parmi les propriétés qu'on cherche généralement à prouver on peut citer :

- preuve de satisfaction : une spécification satisfait certaines propriétés ;
- preuve de raffinement : une abstraction (conception) est correcte c'est-à-dire qu'une spécification plus détaillée satisfait une abstraction de la spécification ;
- preuve de programme : un programme est conforme à une spécification ;
- preuve de terminaison : sous certaines conditions, un programme finira toujours par terminer.

On trouve deux classes de preuve : preuve formelle et preuve informelle :

1. Une preuve informelle est une démonstration en langage naturel. Des notations mathématiques peuvent être utilisées durant cette preuve. Ce type de preuve est utilisé souvent pour démontrer la validité de propriétés dans les domaines scientifiques (surtout en mathématiques). La validité d'une preuve informelle est donnée par consensus (lue et acceptée par les pairs).
2. Une preuve formelle est une démonstration automatisée dans laquelle une formalisation complète de la vérification est nécessaire. Afin que cette vérification soit automatique, l'algorithme de vérification, les hypothèses ainsi que les propriétés attendues doivent être exprimées dans un langage formel (une syntaxe et une sémantique formelles). Par ailleurs, les règles d'inférence indiquant comment produire des formules à partir de formules déjà produites, doivent être fournies aussi. En conséquence, une preuve formelle se ramène à une construction d'un arbre de formules dont les feuilles sont les axiomes et la racine est la formule à prouver.

Plusieurs assistants (ou prouveurs) ont été développés afin d'automatiser le processus de preuve, c'est-à-dire finaliser une preuve suggérée par l'utilisateur grâce aux méthodes de déduction automatique [Chapurlat 2007]. Parmi les assistants de preuve, on retrouve : PVS (Prototype Verification System) [Owre 1996], Coq [CDT 2002], HOL [Gordon 1993] et Isabelle [Paulson 1994].

Cependant, la mise en œuvre de la preuve provoque une contrainte d'utilisation. En pratique, la preuve nécessite la formalisation des spécifications ainsi que des propriétés dans un langage acceptable par l'assistant de preuve. Par ailleurs, la démonstration globale est difficilement obtenue. En plus, l'utilisateur est souvent obligé de guider la preuve en interagissant avec l'assistant [Cansell 2003]. Néanmoins, cette approche a l'avantage d'éviter l'explosion combinatoire [Rushby 2000, Volker 2002, Roussel 2002a, Roussel 2002b]. Un récapitulatif des avantages et des inconvénients est donné dans le tableau 5.3.

5.2.6 Model-checking

5.2.6.1 Principe et idée

Le model-checking est une approche automatique basée sur une modélisation formelle du système par un modèle souvent de type états-transitions (automate, réseaux de Petri, ...) et la

Avantages	Inconvénients
• détection des erreurs assez tôt, • possibilité de traiter des espaces d'états infinis, • faible coût.	• difficulté de formalisation, • expertise et connaissance des langages/outils de preuve, • notions mathématiques + manipulation des concepts abstraits, • erreurs de réalisation non détectées.

Table 5.3: Avantages et Inconvénients de la preuve

spécification des propriétés à vérifier par des formules logiques (logique temporelle). Le model-checking vérifie automatiquement les propriétés sur le modèle et détermine si elles sont vraies ou fausses [Missaoui 2010]. Concrètement, l'algorithme de model-checking combine le modèle et la formule pour calculer l'ensemble des états accessibles du système. Si la propriété n'est pas satisfaite, le model-checker fournit une trace conduisant à la violation de la propriété.

Cependant, étant donné que cette technique est basée sur l'exploration de tous les comportements du modèle (vérification exhaustive), le model-checking souffre d'une limite majeure liée à l'espace d'états qui est l'explosion du nombre d'états. En pratique, la taille du système à vérifier croit exponentiellement par rapport au nombre de processus/tâches qu'il modélise. Cela rend la construction de l'ensemble des états accessibles très difficile voire même impossible en raison de la taille de la mémoire disponible et du temps d'exécution nécessaire. Néanmoins, trois techniques ont été proposées afin de faire face à ce problème à savoir : les techniques de compression, les optimisations basées sur les réductions et les techniques d'abstraction [Merz 2006].

Une variante de cette technique de vérification est le model-checking aléatoire [Boyer 2004]. En effet, le model-checking aléatoire est une technique à mi-chemin entre le test et le model-checking. L'idée est de modéliser par une fonction aléatoire les stimuli et d'explorer toutes les réponses possibles. Cette fonction aléatoire ne permet de générer qu'un seul stimulus pour lequel toutes les réponses sont explorées. Cette technique ne permet pas toujours de s'assurer qu'une propriété est vérifiée puisqu'elle ne permet d'effectuer qu'une exploration partielle du système. En résumé cette technique est généralement plus efficace que le test, mais moins que le model-checking traditionnel qui repose sur une exploration exhaustive. Un récapitulatif des avantages et des inconvénients du model-checking est donné dans le tableau 5.4.

Avantages	Inconvénients
• assez efficace pour la détection des erreurs, • automatique, • exhaustif, • détection des erreurs relativement tôt, • bon rapport coût/bénéfice.	• espace d'état fini, • explosion combinatoire d'espace d'état, • erreurs de réalisation non détectées.

Table 5.4: Avantages et Inconvénients du model-checking

5.2.6.2 Outils

Les outils de model-checking peuvent être classés selon deux principales catégories : les outils qualitatifs et les outils quantitatifs. Un outil de model-checking qualitatif repose sur des représentations qualitatives des systèmes (sans quantités de temps), alors qu'un outil de model-checking quantitatif repose sur des représentations quantitatives (temporisées). Par conséquent, selon le type de model-checker, différentes propriétés peuvent être vérifiées. Les model-checkers qualitatifs permettent de vérifier des propriétés d'ordonnancement (ordre des tâches, des événements,...) ou de terminaison (fin d'un processus, d'une tâche, ...). Cependant, dû à leur nature, ces outils ne permettent pas de vérifier directement des propriétés temporisées (par exemple, *toute demande d'une ressource finira par être satisfaite en moins de 3 secondes*). Pour pouvoir vérifier ce type de propriété il faut faire appel à des model-checkers quantitatifs. Dans la suite, nous citons quelques outils de model-checking pour chaque catégorie.

5.2.6.2.1 Model-checkers qualitatifs

- **SMV** [1] [McMillan 1993] : c'est un model-checker qualitatif développé à l'université de Carnegie-Mellon aux États-Unis. SMV utilise la logique CTL pour l'expression des propriétés à vérifier. Cet outil fait recours à une approche symbolique pour la vérification et par conséquent offre la possibilité d'être appliqué à des systèmes à taille très élevée [Clarke 1994, Medjoudj 2006].
- **SPIN** [2] [Holzmann 1997] : c'est un model-checker qualitatif développé par Bell Labs aux États-Unis. SPIN repose sur un langage spécifique, Promela (a Process Meta Language) [Holzmann 1997], pour la description du système et sur la logique temporelle PLTL pour l'expression des propriétés. L'avantage de SPIN est qu'il a recours à des méthodes de réduction du nombre d'états : ordre partial et compression des états ce qui a permis de l'appliquer avec succès à des applications industrielles [Havelund 2001].
- **TINA** [3] (Time Petri net Analyzer) [Berthomieu 2003, Berthomieu 2010] : c'est un model-checker qualitatif. Tina utilise des réseaux de Petri T-temporels pour la représentation du système et LTL et CTL pour l'expression des propriétés.

5.2.6.2.2 Model-checkers quantitatifs

- **UPPAAL** [4] [Larsen 1997] : c'est un model-checker quantitatif développé par l'université d'UPPsala en Suède et l'université d'AALborg au Danemark. Cet outil utilise les automates temporisés pour la description du système et la logique temporelle TCTL pour l'expression des propriétés. Cependant, l'outil ne prend en compte que des formes de synchronisation binaire et ne vérifie qu'une classe restreinte de propriétés d'atteignabilité [Medjoudj 2006]. Néanmoins, Uppaal dispose d'un simulateur et d'une interface graphique bien développée. Cet outil a été utilisé dans plusieurs études de cas industriels et académiques [5].
- **KRONOS** [6] [Yovine 1997] : c'est un model-checker quantitatif développé par VERIMAG [7] à Grenoble. Kronos utilise les automates temporisés pour la description du sys-

1. http://www.cs.cmu.edu/~modelcheck/smv.html
2. http://www.spinroot.com
3. http://www.laas.fr/tina
4. http://www.uppaal.com et http://www.uppaal.org
5. Une liste non exhaustive de cas industriels et académiques est disponible via http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml
6. http://www-verimag.imag.fr/Kronos.html
7. http://www-verimag.imag.fr

tème et la logique temporelle TCTL pour l'expression des propriétés. Kronos permet la vérification d'une large classe de propriétés (atteignabilité, sûreté et vivacité) sur des modèles temporisés et combine plusieurs techniques d'analyses (*forward analysis*, *backward analysis* et *on-the-fly*). L'outil dispose également d'un algorithme de model-checking symbolique, ce qui permet de résoudre partiellement le problème d'explosion du nombre d'états. Kronos est cependant destiné aux utilisateurs spécialistes dans les énoncés formels et n'offre pas d'interface graphique ni de module de simulation.

Remarque 5.1 *Pour les nombreux avantages qu'il propose (automatique, exhaustif, utilisation industrielle et académique, etc.), le model-checking est la technique retenue et sur laquelle repose l'approche de vérification que nous proposons. Cette approche est détaillée dans la section suivante de ce chapitre.*

Rappelons à ce stade les volets sur lesquels porte notre contribution dans le cadre de ce travail de thèse. le premier consiste à développer des mécanismes d'assistance à la spécification des exigences temporelles, et qui a fait l'objet de la première partie. Le deuxième volet concerne l'activité de modélisation et consiste à transformer des modèles SM avec des annotations temporelles en modèles AT (2^{me} partie). Le troisième volet est axé sur la phase de vérification des exigences temporelles, et va faire l'objet de la suite de ce chapitre.

La section 5.3 est consacrée à la présentation du processus de vérification. En section 5.3.2, nous présenterons les fonctionalités qui supportent ce processus, dans l'outil logiciel développé. Finalement, une comparaison du processus de vérification que nous proposons avec des travaux similaires est donnée en section 5.5.

5.3 Processus de vérification proposé

Dans la section 5.3.1, nous passons en revue les différentes étapes par lesquelles passe l'approche que nous proposons. L'étape de vérification dans cette approche repose sur une base de patterns d'observation. Celle-ci est introduite en section 5.3.2.

5.3.1 Architecture globale

Sur la figure 5.4 illustrant le processus global allant de la spécification à la vérification, nous pointons les différentes contributions proposées.

Le point du départ de notre approche de vérification est l'activité relative à l'expression des exigences temporelles. Cette étape est cadrée par le biais d'une grammaire de spécification à base de motifs pré-établis en langage naturel. Ainsi, le processus d'identification des exigences temporelles exprimées est par conséquent rendu automatique à partir des assertions produites. Par ailleurs, nous avons développé un algorithme qui permet la détection de différents types d'incohérences pré-établies dans un ensemble d'exigences exprimées sur la base de notre grammaire. Pour l'étape de modélisation, l'idée est de proposer une transformation automatique vers des modèles AT sur lesquels la vérification des exigences sera déroulée.

L'idée que nous mettons en œuvre pour l'étape de vérification des exigences temporelles est la suivante : pour chaque classe de propriétés dans la classification proposée en chapitre 2, nous avons associé un *pattern d'observation* afin de jouer le rôle d'un *chien de garde* (*watchdog* en anglais) pour la surveiller. L'ensemble de ces patterns d'observation forment une base de patterns d'observation. Cette base de patterns sera présentée en section 5.3.2. En pratique, pour un système donné un ensemble $\mathbb{E}$ d'exigences temporelles a été défini sur la base de notre grammaire de spécification. Le type de chaque exigence est identifié, et le pattern d'observation

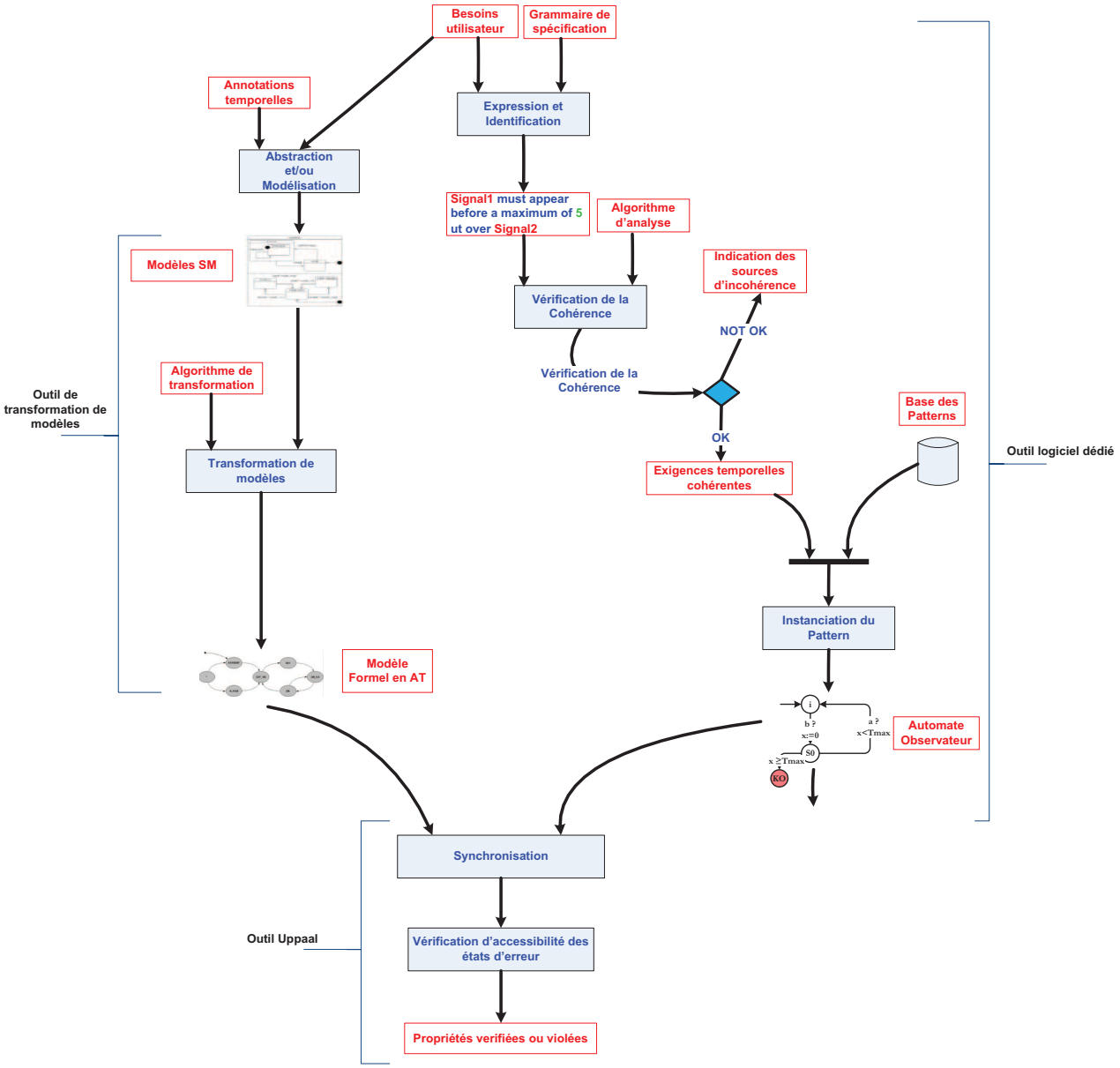

FIGURE 5.4: Approche Globale

associé à ce type est utilisé pour générer un AT observateur avec les paramètres appropriés. À la fin de cette étape un ensemble d'automates observateurs $\mathcal{O}$ est obtenu.

Par la suite, un produit de synchronisation est appliqué sur le modèle AT du système (obtenu par transformation de modèles) et les observateurs instanciés afin de générer un modèle global $\mathcal{M}$. En d'autres termes, $\mathcal{M}$ contient le modèle du système à vérifier ainsi que les observateurs associés aux propriétés à vérifier. La vérification par la suite consiste à analyser -par model-checking- si les états "KO" respectivement sur les différents automates observateurs sont accessibles. Cela est très intéressant de point de vue pratique vu que des outils de model-checking, comme Uppaal [Larsen 1997], se prêtent bien à la vérification des propriétés d'atteignabilité.

5.3.2 Base de patterns

La mise en œuvre du concept "observateur" pour la vérification est une approche largement utilisée dans la littérature [Halbwachs 1994, Aceto 1998, Roger 2006, Dhaussy 2007, Fontan 2008, Hedia 2008, Ghazel 2009b]. Dans cette approche, les exigences ou propriétés à vérifier sont associées à des observateurs passifs qui sont ensuite couplés au modèle du système à "*observer*" par un produit de synchronisation. Dans la suite, nous donnerons une définition de la notion d'observateur, ensuite nous présenterons la base de patterns d'observation que nous avons définie.

5.3.2.1 Observateur

Un observateur est un modèle construit de manière à encoder une propriété ou une exigence. Son rôle est d'*observer* ou surveiller les occurrences des évènements significatifs pour cette propriété survenant dans le modèle du système à surveiller. D'un point de vue pratique, notre étude repose sur des modèles d'automates (le modèle du système et les modèles observateurs). Ainsi, un observateur est un automate qui dispose d'un ensemble de nœuds dits d'erreur (KO). Ensuite, il est synchronisé avec le modèle à vérifier afin d'identifier s'il existe une exécution qui viole la propriété associée à celui-ci, c'est-à-dire une exécution qui mène à l'état d'erreur "KO".

L'utilisation d'automate observateur comporte plusieurs avantages [Godary 2004]. Un premier avantage est qu'un automate observateur est l'aspect générique des observateurs. En effet, un observateur est un modèle générique défini pour un type précis de propriétés, et est initialisé à chaque fois que ce type est rencontré. Le deuxième est l'indépendance de la définition des observateurs par rapport à la modélisation du système à observer. Cela permet de vérifier plusieurs propriétés en parallèle. Cependant, la limitation majeure des observateurs est qu'ils ne permettent pas de vérifier tout types de propriétés [Hedia 2008]. De plus, l'ajout d'observateurs augmente la taille du modèle à analyser.

5.3.2.2 Base des observateurs

Les "*Patterns*" sont des concepts du génie logiciel décrivant une solution à un problème récurrent dans un environnement donné, de telle manière que cette solution soit réutilisable à chaque fois qu'on rencontre le même problème [Gamma 1995]. Il s'agit d'un niveau d'abstraction supplémentaire qui vise à éviter la dispersion lors du développement d'applications complexes. Les *Patterns* proviennent à la base du domaine de l'architecture et ont été transposés au domaine du génie logiciel notamment avec l'apparition de l'orienté objet. De nos jours, nombreux sont les domaines d'utilisation de patterns : génie logiciel [Gamma 1995], ingénierie des exigences [Hedia 2008], diagnostic [Girault 2005], etc. On verra dans ce qui suit comment nous mettrons en œuvre ce concept dans le cadre de nos travaux sur la vérification des exigences temporelles.

Définition 5.1 *"Le mot anglais* pattern *est souvent utilisé pour désigner un modèle, une structure, un motif, un type, etc. Il s'agit souvent d'un phénomène que l'on peut observer de façon répétée lors de l'étude d'un de certains sujets auquel il peut conférer des propriétés caractéristiques . . . En informatique, les patterns sont des façons de programmer éprouvées et réputées pour apporter des propriétés comme la cohérence, la robustesse, la réutilisabilité, etc."*[8]

L'ensemble des patterns d'observation associés respectivement aux types d'exigences dans notre classification (cf. chapitre 2), va former notre base de patterns [Mekki 2010a]. Ainsi, en

8. http://dictionnaire.sensagent.com/pattern/fr-fr/

pratique le processus de vérification des exigences d'un système donné se fera par l'instanciation de patterns relatifs aux exigences exprimées pour obtenir des "observateurs". Ensuite, ces observateurs sont greffés au modèle du système. Le processus de vérification aura comme but la surveillance des violations des exigences tout en analysant l'accessibilité des états de défaillances (***KO***) dans le modèle de synchronisation. Par exemple, pour vérifier un système $\mathbb{S}$ avec un observateur $\mathbb{O}$, le processus est comme suit : l'observateur $\mathbb{O}$ est composé avec $\mathbb{S}$ au moyen d'une composition synchrone ($\mathbb{S} \parallel \mathbb{O}$). Une analyse d'accessibilité des états d'erreur est effectuée sur le produit de cette composition ($\mathcal{R} : \mathbb{S}\|\mathbb{O} \rightsquigarrow$ erreur). Si l'un de ces états (KO) est accessible, alors la propriété $\mathcal{R}$ n'est pas vérifiée. Si, au contraire, aucun de ces états n'est accessible, la propriété est considérée comme valide.

Dans la suite nous présenterons deux exemples d'observateurs. Tout d'abord nous donnons l'exigence concernée exprimée sur la base de la grammaire de spécification définie. Ensuite le type de l'exigence est donné, et l'observateur est généré à partir du pattern approprié. Les observateurs seront donnés sous forme graphique (modèle AT Uppaal), et sous forme de fichier XML généré par notre outil selon le format Uppaal.

5.3.2.3 Exemple 1

Soit une propriété $\mathcal{P}_1$ décrite comme suit : pour une tache de T, la chaine de production doit être arrêtée au plus tard 5 unités de temps après la troisième panne. Soient les signaux suivants :

- le signal **D** utilisé pour indiquer le début de la tâche T,
- le signal **F** pour indiquer la fin de la tâche T,
- le signal **P** pour indiquer la détection d'une panne,
- le signal **A** pour demander l'arrêt de la production.

Cette propriété peut être exprimée en utilisant la grammaire de spécification comme suit :

Between *D* and *F* : *A* **must occur before a maximum delay of** *5* **t.u. after** *3* **Th occurence of** *P*

La figure 5.5 représente l'automate observateur $\mathcal{O}_1$ associé à la propriété $\mathcal{P}_1$.

5.3.2.4 Exemple 2

Soit une propriété $\mathcal{P}_2$ décrite comme suit : après la détection d'une panne, la demande de réparation ne doit pas être lancée avant trois tentatives de redémarrage de la machine. Soient les signaux suivants :

- le signal **P** pour indiquer la détection d'une panne,
- le signal **R** pour indiquer la demande d'une réparation,
- le signal **D** pour redémarrage la machine.

Cette propriété peut être exprimée en utilisant la grammaire de spécification comme suit :

After *P* : *R* **cannot occur before** *3* **Th occurence of** *D*

La figure 5.6 représente l'automate observateur $\mathcal{O}_2$ associé à la propriété $\mathcal{P}_2$.

5.4 Fonctionnalités de l'outil logiciel relatives à la vérification par observateur

Des outils ont été implémentés afin d'automatiser les étapes de notre approche. Dans cette section, nous présenterons les fonctionnalités relatives à l'activité de vérification des exigences temporelles. À partir des assertions utilisées pour exprimer les exigences temporelles, l'outil

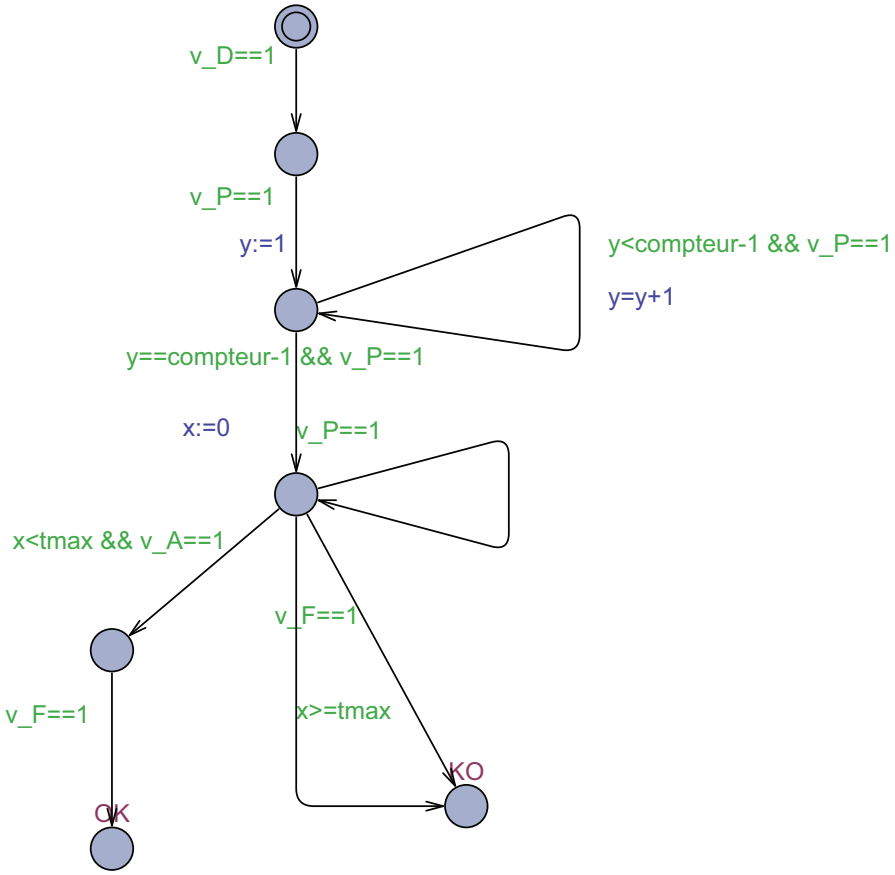

FIGURE 5.5: Automate Observateur $\mathcal{O}_1$ associé à la propriété $\mathcal{P}_1$

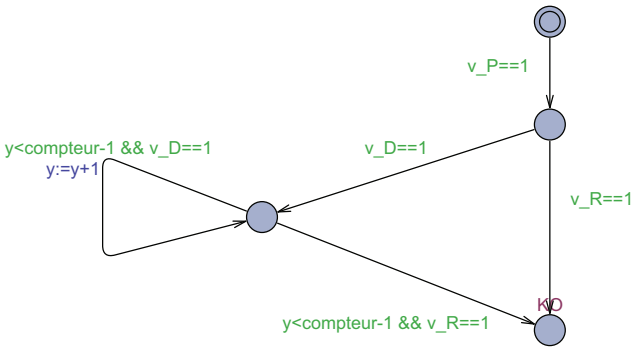

FIGURE 5.6: Automate Observateur $\mathcal{O}_2$ associé à la propriété $\mathcal{P}_2$

permet d'identifier le type des exigences exprimées, et instancie en conséquent les patterns observateurs associés à ces exigences. Chaque observateur est par la suite exporté par l'outil sous forme d'un fichier XML selon le format Uppaal.

Ainsi dans Uppaal, on synchronise le modèle AT obtenu par transformation du modèle SM du système à vérifier, avec les ATs observateurs générés par notre outil. Ensuite, une formule en logique temporelle CTL exprimant l'accessibilité de l'état *KO* relatif à l'observateur de $\mathcal{R}$ est définie, $\mathcal{R}$ étant une exigence qu'on cherche à vérifier sur le modèle du système. Finalement, la vérification se déroule en vérifiant la validité de la formule CTL exprimée. Deux cas de résultats sont possibles : la formule est valide ou non valide. Le premier résultat est interprété comme

l'existence de chemins menant à l'état *KO*. Cela exprime que l'exigence $\mathcal{R}$ n'est pas validée sur le modèle donné. Le deuxième résultat est interprété comme l'absence d'exécutions menant à l'état KO et par conséquent la validation de $\mathcal{R}$ sur le modèle à vérifier. Ici, concrètement, la formule CTL s'écrit comme suit : *EF(KO)*.

5.5 Comparaison avec les travaux existants

Cette section synthétise les principaux travaux qui présentent des approches similaires, c'est-à-dire des approches dont l'objectif est la vérification des propriétés temporelles d'une spécification en utilisant la notion d'observateur. En effet, nombreux sont les travaux autour de la mise en œuvre des observateurs pour la surveillance des propriétés. [Roger 2006, Hedia 2008] présentent un tour d'horizon de ces travaux qui sont répertoriés en deux classes : une première classe concerne les observateurs sous formes d'automates (automate de Büchi [Vardi 1986, Gerth 1996], automate de tests [Aceto 1998, Bouyer 2002, Aceto 2003]) et une deuxième catégorie qui regroupe les observateurs qui ne sont pas des automates (les observateurs Véda [Jard 1988], les observateurs pour systèmes auto-validés [Diaz 1994], les observateurs de machines synchrones [Halbwachs 1994]). Un bilan très intéressant de différents types d'observateurs est donné dans [Roger 2006, Hedia 2008]. Dans la suite, nous détaillerons certains travaux similaires.

5.5.1 Projet OMEGA

OMEGA [9] est un projet européen qui propose des modèles dédiés au développement des systèmes complexes, et dont l'objectif est de proposer une approche de validation de ces systèmes. Pour cela, un profil UML (dit profil OMEGA [10]) a été défini pour pouvoir exprimer les exigences temporelles. De plus, une boite à outils IFx a été développée afin d'exécuter l'étape de vérification. Ainsi, des transformations ont été implémentées pour supporter l'utilisation du profil OMEGA afin de générer des spécifications en langage IF [Ober 2006], langage spécifique à IFx. Ce dernier est utilisé, à travers une interface graphique, pour simuler et valider formellement les propriétés temporelles. L'avantage majeur du projet OMEGA est qu'il rend transparent l'utilisation de techniques formelles auxquelles il fait recours tout en simplifiant la vérification en utilisant des observateurs. Néanmoins, ce projet souffre de plusieurs limites données dans [Atitallah 2008] et [Hooman 2008]. D'après [Atitallah 2008] l'inconvénient d'OMEGA est qu'il ne supporte pas la description de l'architecture d'un système sur puce étant donné que l'utilisation de ce projet est restreinte à un domaine d'application spécifique. De plus, [Hooman 2008] dont les auteurs sont des membres d'OMEGA, présente un retour d'expérience sur l'utilisation de différentes techniques développées dans OMEGA. D'après ces auteurs, OMEGA et surtout la boite à outils IFx, souffre des limitations suivantes :

- Explosion combinatoire du nombre d'états,
- Difficulté d'apprentissage du langage IF (4 à 5 semaines pour apprendre le langage),
- Erreur signalée au niveau langage IF : OMEGA ne permet pas de remonter le message d'erreur jusqu'au niveau UML,
- Absence d'une base générique d'observateurs,
- Traduction manuelle des spécifications lors du passage d'une technique à une autre dans OMEGA,

9. Projet Européen OMEGA (IST-33522), `http://www-omega.imag.fr`
10. Ce profil est en fait un raffinement du profil UML standard Scheduling, Performance and Time (SPT)

- Validation du profil UML pas encore faite.

5.5.2 OBP (Observer-Based Prover)

[Roger 2006] propose une approche qui repose sur trois types d'automates à greffer au modèle du système à vérifier : les automates d'observation pour encoder les propriétés à surveiller, les automates de contexte afin de simuler l'environnement du modèle observé et finalement les automates de restriction qui limitent l'espace d'observation. Ce dernier type d'automate est l'équivalent du concept de **scope d'une propriété** proposé par Dwyer [Dwyer 1999] et repris dans notre approche. Dans cette approche, le modèle à vérifier ainsi que les propriétés à valider doivent être encodés en langage IF, alors que les automates de contexte sont décrits dans le langage CDL (Context Description Language). Ainsi, l'outil IFx est utilisé pour l'étape de vérification. De plus, un langage nommé CxUCC à été défini afin de faciliter l'expression des observateurs. Cependant, cette approche souffre des limites suivantes [Roger 2006] :

- Pas d'implémentation réelle pour l'automate de contexte, resté comme une proposition,
- Pas de validation pour la transformation CxUCC vers IF,
- Hérite des limites de la boite à outils IFx.

Remarque 5.2 *Malgré l'explosion combinatoire du nombre d'états, qu'on partage d'ailleurs avec la boite à outils IFx utilisée dans le projet OMEGA et dans l'outil OBP, notre approche présente plusieurs avantages par rapport aux approches présentées ci-dessus :*

1. *Contrairement à IFx qui utilise une notation spécifique qu'est IF, nous utilisons un langage standard qui est l'automate temporisé. Ce dernier est un langage beaucoup plus répandu contrairement à IF qui en plus nécessite de 4 à 5 semaines d'apprentissage,*
2. *Nous utilisons une formalisation bien définie des SM (formalisation détaillée au chapitre 4),*
3. *Nous proposons une transformation automatique et valide qui prend en entrée des modèles SM afin de générer des modèles en automates temporisés,*
4. *Nous simplifions la phase de construction des observateurs en utilisant une base générique pré-définie d'automates d'observation relatifs aux principaux types de propriétés temporelles,*
5. *Contrairement à IFx, de nombreux outils très performants utilisant les automates temporisés ont été développés et qui ont fait leur preuve dans le domaine académique ainsi que dans l'industrie.*

5.5.3 Autres travaux

D'autres travaux ont proposé l'utilisation de patterns pour l'expression des exigences temporelles. Parmi ces travaux on trouve : [Aceto 1998], [Dwyer 1999], [Aceto 2003], [Konrad 2005] et [Gruhn 2006]. Cependant, ces travaux s'arrêtent au niveau de la définition des patterns, c'est-à-dire qu'ils n'intègrent pas les patterns dans une chaine globale de vérification des exigences. Ainsi, leurs propositions s'arrêtent au niveau de la spécification. À cela se rajoutent d'autres limites comme la couverture de leur base de patterns ainsi que l'efficacité des outils qui pourraient être utilisés pour l'exploitation de leurs propositions. Néanmoins, certains travaux proposent d'utiliser les patterns de spécification dans le cadre d'une approche globale de vérification, comme [Toussaint 1997] et [Abid 2011]. [Toussaint 1997] propose une approche de vérification dont seulement quatre types d'exigences temporelles sont considérées. Nous nous

focalisons dans cette section sur les travaux de [Abid 2011]. Dans ces travaux, les auteurs proposent une base de patterns pour les exigences temporelles. Chaque pattern dans cette base est défini à travers des langages formels (MTL, FOTT et TGIL). Les observateurs associés aux patterns sont exprimés à l'aide du langage pivot Fiacre. Le point fort de ce travail est le nombre d'exigences qu'on peut exprimer. Néanmoins, cette approche souffre de certaines limites :

- Pas d'assistance à la phase d'expression et d'identification des exigences : bien que la base de patterns proposée permette d'exprimer une variété d'exigences temporelles, l'absence d'une assistance à l'identification des types d'exigences présente un obstacle non négligeable pour les non-experts.
- Pas d'assistance à la phase de modélisation du système : cette approche se focalise sur l'étape de spécification d'exigences et sur l'étape de vérification.

5.6 Conclusions

Dans ce chapitre, nous avons présenté l'étape de vérification des exigences temporelles. D'abord, nous avons donné un bilan (avantages et inconvénients) de certaines techniques de V&V les plus utilisées. Ensuite, nous avons introduit nos contributions pour cette l'étape. Nous avons commencé par donner l'architecture globale du processus de vérification que nous proposons. Ensuite, nous avons donné une synthèse de nos contributions par rapport aux principales approches similaires. Dans le chapitre suivant, nous illustrons les principales contributions de la thèse à travers la proposition d'une extension des SRS [11] d'ERTMS niveau 2.

11. SRS : System Requirements Specifications

Quatrième partie

Application

Chapitre 6

Proposition de l'intégration du passage à niveau aux SRS d'ERTMS Niveau 2

Résumé : *À travers ce chapitre, nous allons illustrer les principales contributions que nous avons développées sur un cas d'étude du domaine ferroviaire. Ce cas d'étude porte sur la proposition d'une spécification fonctionnelle des passages à niveau dans le cadre d'une exploitation en ERTMS niveau 2. La spécification que nous proposons vise notamment à éviter des situations qui ont été définies comme potentiellement dangereuses sur les passages à niveau.*

Sommaire

6.1 Introduction

Le cas d'étude est relatif au développement d'une architecture fonctionnelle d'un système de protection automatique d'un passage à niveau (PN), dans le cadre d'une exploitation en ERTMS niveau 2. Il s'agit concrètement d'un pas vers l'extension des SRS (System Requirement Specifications) de manière à prendre en compte les passages à niveau.

L'enjeu à travers ce chapitre est double : d'un coté illustrer nos contributions, et de l'autre côté proposer une base à l'intégration des PN, jusque-là considérés comme des points ouverts, dans les spécifications SRS d'ERTMS niveau 2.

Le chapitre est organisé comme suit : la section 6.2 est consacrée à la présentation d'ERTMS, ensuite nous développerons en section 6.3 notre proposition d'intégration des PN, avant de conclure en section 6.5.

6.2 ERTMS : European Rail Traffic Management System

6.2.1 Idées et objectifs

Nombreux sont les obstacles techniques qui se dressent au franchissement d'une frontière européenne [CE 2005, CE 2006]. L'obstacle le plus connu est celui de l'écartement des rails pour lequel la commission européenne recense au moins quatre types d'écartement différents en Europe. D'autres obstacles techniques moins visibles doivent également être levés pour favoriser l'interopérabilité parmi lesquels existent : les différents types de courant électrique, la hauteur des quais pour les voyageurs, les pentes maximales, ... Pour ce qui est de la signalisation ferroviaire et le contrôle de la vitesse de trains, les problèmes liés à leur diversité ont été résolus à l'échelle nationale, en d'autres termes, pour chaque réseau ferroviaire national et indépendamment des autres. Ce manque de standardisation explique la fragmentation caractérisant le réseau ferroviaire européen [CE 2006]. Cette fragmentation entraîne des surcoûts, des risques de panne et complique la tâche des conducteurs, ce qui présente un handicap pour le développement du transport ferroviaire à l'échelle européenne. Par exemple, le train *Thalys*, qui assure principalement la liaison entre Paris et Bruxelles, doit être équipé de sept systèmes de signalisation et de contrôle de vitesse différents pour pouvoir garantir ce trajet [CE 2005]. La mise en œuvre d'ERTMS permet de remédier à ces obstacles en standardisant le système de signalisation et de contrôle commande ferroviaire à travers l'Europe [Jabri 2010b]. Cette standardisation présentera nombreuses avantages [CE 2005, Jabri 2010a] :

- dynamiser le secteur ferroviaire en encourageant la compétitivité,
- favoriser l'intégration, à travers l'Europe, des services ferroviaires de marchandises et de voyageurs,
- stimuler le marché européen des équipements ferroviaires,
- réduire les coûts et accroitre la qualité du transport ferroviaire.

Il est à noter par ailleurs qu'il existe trois niveau d'ERTMS ; 1, 2 et 3.

6.2.2 Les composants d'ERTMS

ERTMS est constitué essentiellement de deux sous-systèmes de base [CE 2006, Barger 2009] :

1. l'ETCS (European Train Control System) est le système de contrôle du train, qui permet de transmettre au conducteur les informations relatives à la vitesse autorisée et les

autorisations de mouvement, ainsi que de contrôler le respect des indications. Un ordinateur embarqué compare la vitesse du train avec la vitesse maximale permise et freine automatiquement en cas de dépassement.

2. Le GSM-R (GSM for Railways) est le système de communication radio utilisé pour l'échange d'informations entre le sol et le bord du train. Ce système est fondé sur le standard GSM de téléphonie mobile mais utilise des fréquences différentes propres au ferroviaire, ainsi que certaines fonctions avancées relatives à l'application ferroviaire, comme la diffusion multicast par exemple. Il permet au conducteur de dialoguer/échanger avec les centres de régulation. Le GSM-R est utilisé en ETCS niveaux 2 et 3.

Remarque 6.1 *Deux autres composants (European Traffic Management Layer (ETML) et Interlocking Safety System (INESS)) sont en cours d'élaboration et seront prochainement intégrés à ERTMS.*

6.2.3 Architecture d'ERTMS/ETCS

L'élément clé sur lequel repose ERTMS est la standardisation de l'échange des données de gestion du trafic ferroviaire. L'idée est comparable à celle utilisée dans la gestion du trafic aérien. Pour ce dernier, la standardisation de la communication permet à n'importe quelle avion de décoller et d'atterrir de/dans n'importe quel aéroport. De la même façon, l'idée est de permettre à tout train circulant en Europe d'utiliser et de communiquer avec n'importe quelle infrastructure (système de contrôle, station, ...) dans les pays qu'il traverse. Cela n'implique pas que tous les trains doivent être équipés avec les mêmes équipements, mais plutôt une communication standard (un échange/format des données standard) entre le train et les équipements du sol ainsi qu'entre le train et le centre de contrôle et de gestion du trafic ferroviaire. C'est dans ce cadre là qu'ERTMS a été proposé. La spécification de celui-ci répond et propose des solutions à des besoins fonctionnels très variés selon trois niveaux [RFF 2006, CE 2006, Barger 2009]. La figure 6.1 [1] illustre graphiquement ces trois niveaux [2].

6.2.3.1 Niveau 1

Le train reçoit son autorisation de mouvement par le biais des balises installées au sol. L'autorisation de mouvement est calculée par le centre de contrôle. Concrètement, le niveau 1 repose sur le système de signalisation latérale tout en rajoutant un dispositif ponctuel standard pour la transmission sol-train. Ce standard concerne le support physique de communication ainsi que le contenu des messages échangés. Le système de transmission **Eurobalise** a été proposé par l'UNISIG [3] comme une réponse à ce standard et il est constitué des éléments suivants :

- Un codeur en interface entre les installations de signalisation latérale et la balise,
- Des groupes de balise sur la voie, une balise possède deux interfaces pour les échanges sol-bord et les échanges de la balise vers le codeur,
- Une antenne et un boîtier de réception embarqués, connus sous le nom de BTM (Balise Transmission Module).

1. Figure faite par Etienne Lemaire, chargé de recherche à IFSTTAR
2. http://www.ti.kviv.be/presentatiesignalisatie/public/Presentaties%2031%20maart%202011/Yves%20Carels%20ERTMS%20Factsheets%20ERTMS%20Levels.pdf
3. UNISIG, l'association des Industriels Européens de Signalisation Ferroviaire, est un des comités de l'association européenne de l'industrie du rail (UNIFE), http://www.unife.org/

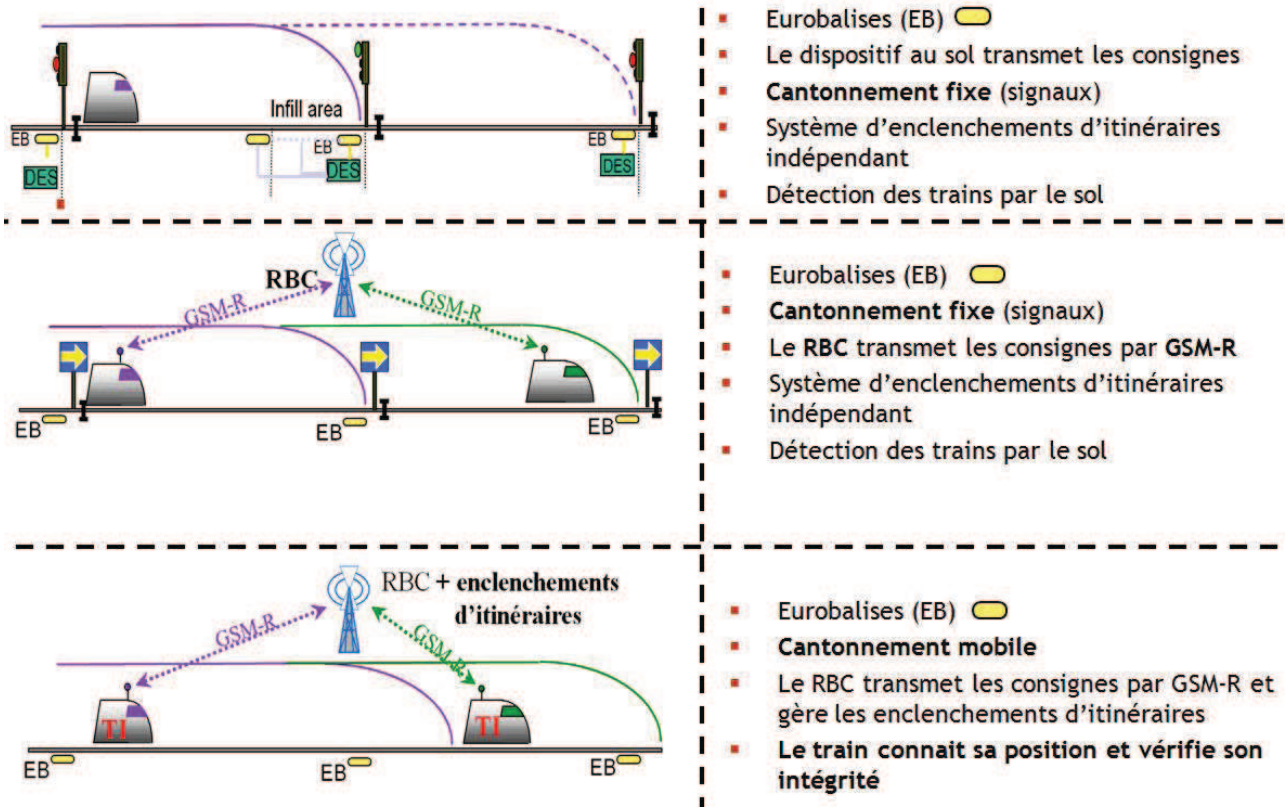

FIGURE 6.1: Les trois niveaux d'ERTMS/ECTS

Le principal module ERTMS\ETCS à bord du train est l'Eurocab (EVC, European Vital Computer). C'est un calculateur qui intègre l'antenne et le module de transmission Eurobalise. L'EVC est en interface avec le conducteur à travers le MMI (Man Machine Interface, parfois nommé DMI, Driver Machine Interface) et avec le train à travers le TIU (Train Interface Unit). De plus, il permet de contrôler la vitesse du train en utilisant les informations fournies par l'odomètre. Ce dernier détermine la position du train en utilisant les références géographiques obtenues par la lecture du dernier groupe d'Eurobalises rencontré et les informations de déplacement fournies par les pédales de mesure du chemin parcouru.

En résumé, au niveau 1, le sol transmet au train les informations à travers les Eurobalises placées le long de la voie, dont il a besoin pour tracer sa courbe de vitesse. Le niveau 1 est caractérisé par un cantonnement fixe avec une vérification de l'intégrité des trains faite par le système de contrôle au sol.

6.2.3.2 Niveau 2

L'architecture niveau 2 se caractérise par un transfert régulier d'informations en utilisant la liaison GSM-R. En effet, les informations obtenues en niveau 1 à travers les balises sont transmises en niveau 2 par le RBC (Radio Block Center) via la liaison GSM-R. Ainsi, les Eurobalises ne sont plus utilisées que pour le recalage de l'odomètre. Les RBC, installés en interface avec le système d'enclenchement, traduisent les données de signalisation en autorisations de mouvement et par la suite les transmettent au train via le GSM-R. Comme pour le niveau 1, le niveau 2 est caractérisé par le cantonnement fixe, et la vérification de l'intégrité des trains est faite par le sol. Cependant, contrairement au niveau 1, les communications sol-train se font par GSM-R et non pas à travers les Eurobalises. Cela explique le rajout d'un module GSM-R

à l'équipement à bord du train.

6.2.3.3 Niveau 3

Le niveau 3 vise une exploitation maximale du réseau ferroviaire. Ce niveau est destiné à être exclusivement à bord du train, en évitant tous les équipements le long de la voie. Ainsi, les circuits de voie ne sont plus utilisés pour la détection des trains. Par ailleurs, un dispositif de contrôle d'intégrité du train, qui garantit la stabilité du train avec le niveau de sécurité requis, est rajouté aux équipements à bord. De plus, en utilisant la localisation fournie par les trains eux-mêmes, les centres de traitement au sol (RBC), qui utilisent la transmission continue sol-train par radio (GSM-R), attribuent aux trains des cantons mobiles. Il est à noter cependant que le niveau 3 d'ERTMS reste pour le moment un concept théorique, puisqu'il n'est encore implémenté dans aucun réseau ferroviaire.

Nous rappelons que le cas d'étude traité ici vise à intégrer les PNs dans les SRS d'ERTMS niveau 2 dont l'architecture est décrite dans la section suivante.

6.2.4 Description détaillée de l'ERTMS niveau 2

L'ERTMS/ECTS niveau 2, comme illustré dans la figure 6.2 [4], est composé de deux parties, à savoir le sous-système à bord du train et le sous-système sol. La première mise en service commercial d'ERTMS niveau 2 a été, entre Rome et Naples, en janvier 2006 [RFF 2006]. Ainsi, il a fallu quinze ans pour que l'initiative de l'union internationale des chemins de fer (UIC), lancée en 1991, commence à se concrétiser en Europe. "Une Europe du rail où les spécificités nationales ont donné naissance à 23 systèmes de signalisation différents" [RFF 2006].

6.2.4.1 Le sous-système bord

En pratique, le train doit recevoir une autorisation de mouvement contenant la distance de voie réservée pour pouvoir rouler. Cette autorisation est mise à jour au fur et à mesure que le train avance. À partir de cette autorisation, le calculateur à bord, l'EVC, dessine la courbe de vitesse maximale autorisée, la courbe d'alerte, la courbe d'intervention du freinage de service et la courbe d'intervention de freinage d'urgence. En tenant compte de la position du train, de sa vitesse instantanée et de sa capacité de freinage, ces différentes courbes sont utilisés par la suite afin de contrôler la vitesse du train. Par exemple, dans le cas où le conducteur ne tient pas compte de la courbe de vitesse maximale autorisée, l'EVC intervient pour l'alerter et lui demander un freinage de service. On note ici que le calculateur EVC est en interface avec le conducteur à travers le MMI (ou DMI), avec le BTM (Balise Transmission Module) utilisé pour récupérer les informations à partir de balises placées sur la voie et, finalement avec l'antenne du GSM-R utilisée pour l'échange de messages avec le RBC (Radio Block Center).

Remarque 6.2 *Le cœur du système bord est le EVC (figure 6.3 [5]). Ce dernier est interfacé avec les équipements suivants :*

- *Le MMI (ou DMI, Men/Driver Machine Interface) afin de dialoguer avec le conducteur.*
- *Le BTM (Balise Transmission Module) afin de lire les messages contenus dans les balises au sol.*
- *Le LTM (Loop Transmission Module) afin de lire les données contenues dans les boucles au sol.*

4. http://www.x-rail.com/static-docs/ERTMS_UK.pdf
5. http://www.belrail.be/F/infrastructure/signalisation/index.php?page=ertms

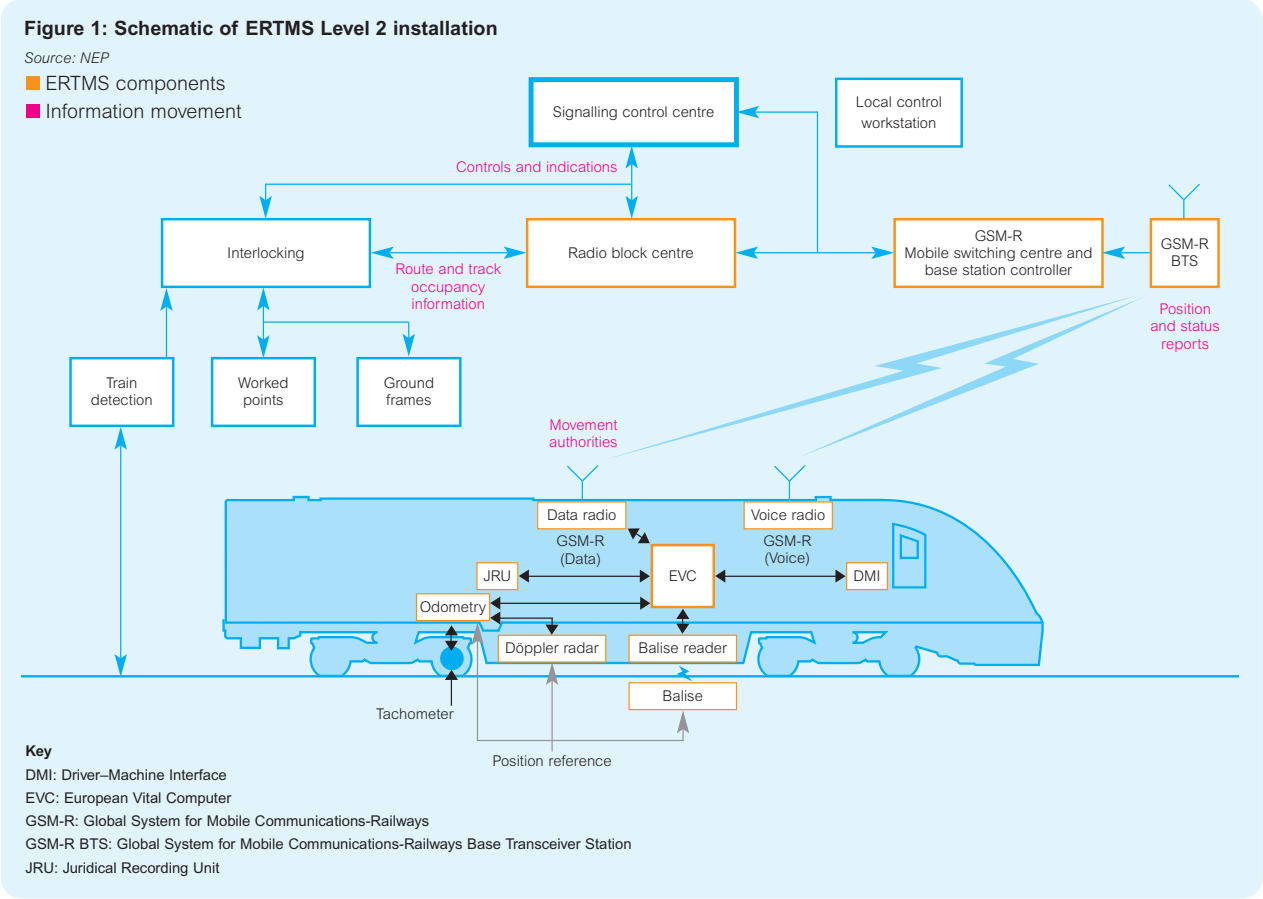

FIGURE 6.2: ERTMS/ECTS niveau 2 : architecture

- *Les capteurs odométriques afin de déterminer la position du train. Celle-ci est utilisée pour le calcul des courbes de vitesse.*
- *Le RTM (Radio Transmission Module) afin de communiquer avec le centre de contrôle via l'antenne GSM-R.*
- *Le TIU (Train Interface Unit) afin de contrôler les composants du train (pantographes, freins, portes, etc).*
- *Les STM (Specific Transmission Modules) afin de pouvoir rouler sur des voies non équipées d'ERTMS.*

6.2.4.2 Le sous-système sol

Le sous-système sol est composé du RBC (Radio Block Center) et des groupes de balises. Ces derniers sont utilisés comme des repères géographiques tandis que le RBC gère les autorisations de mouvement des trains. Ce gestionnaire est en interface avec le système d'enclenchement et l'Euroradio en transmettant à ce dernier les données provenant de la voie, comme la position des trains. L'équipement Euroradio transmet à son tour les autorisations de mouvement à tous les trains qui se trouvent sur la voie.

Remarque 6.3 *Le système sol (figure 6.4*[6]*) est constitué des entités suivantes :*

- *Les euro-balises placées sur la voie afin de transmettre ponctuellement aux trains les informations venant de l'interlocking*

6. http://www.belrail.be/F/infrastructure/signalisation/index.php?page=ertms

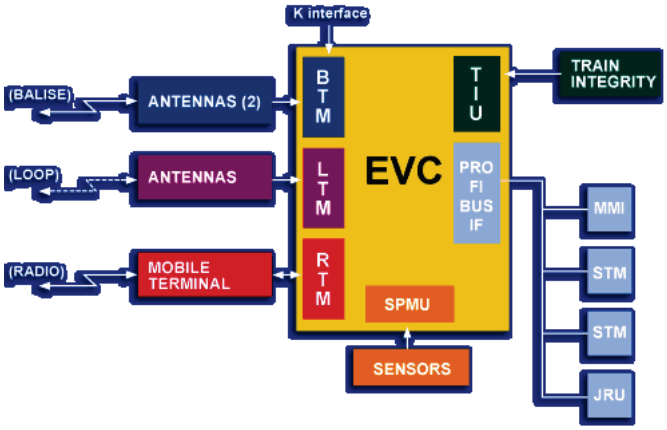

FIGURE 6.3: ERTMS/ECTS niveau 2 : système à bord

- *Les euro-boucles assurent une transmission continue des informations aux trains.*
- *L'interlocking (les enclenchements) permet de prévenir toute situation d'actions incompatibles.*
- *Le centre de contrôle dont le rôle est de centraliser les itinéraires des trains.*
- *Le RBC (Radio Block Center,) afin de communiquer en continu avec les trains via le GSM-R.*

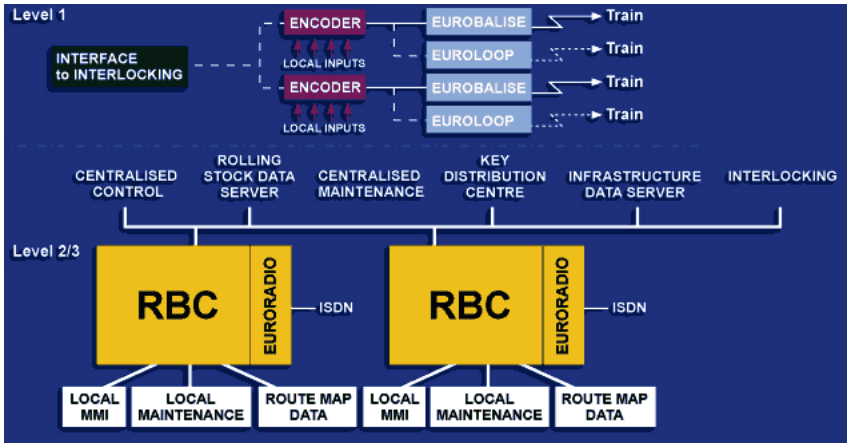

FIGURE 6.4: ERTMS/ECTS niveau 2 : système au sol

Remarque 6.4 *Étant donné son mode de transmission continu, l'ERTMS niveau 2 permet d'augmenter la capacité des lignes et de diminuer les temps de parcours. En chiffres, cela devrait permettre d'augmenter la capacité des lignes les plus fréquentées de 10 à 15 % [RFF 2006].*

Après avoir présenté le système ERTMS en mettant l'accent sur le niveau 2, nous détaillons dans la suite l'architecture du PN que nous proposons dans ce contexte d'exploitation. Cette architecture est une esquisse d'une proposition dont l'objectif est l'intégration du PN à la spécification d'ERTMS niveau 2, jusqu'à présent omis dans les SRS.

6.3 Intégration du passage à niveau dans les spécifications d'ERTMS niveau 2

Dans cette section, nous abordons les mêmes objectifs traités dans [Mekki 2012b] avec objectif l'intégration du PN aux spécifications d'ERTMS niveau 2. Nous illustrons au fur et à mesure les contributions présentées dans les chapitres antérieurs de la thèse.

6.3.1 Passage à niveau - contexte

Définition 6.1 *Un passage à niveau (PN)*[7] *est une intersection entre une ou plusieurs voies ferroviaires avec une voie routière ou piétonnière. En france, ce type de croisement est caractérisé par la priorité absolue donnée aux trains par rapport aux usagers de la route.*

Les statistiques des accidents dans le secteur ferroviaire pointent le PN comme le maillon faible de la sécurité ferroviaire avec plus de 300 morts par an en Europe [Ghazel 2009a] suite à des accidents survenant aux PN. Ce chiffre représente 29 % des décès occasionnés par les accidents ferroviaires. Ces accidents génèrent des dégâts considérables étant donné le coût des réparations ainsi que la perturbation du trafic souvent engendrée. De plus, ces accidents sont largement couverts médiatiquement, ce qui altère la réputation du transport ferroviaire considérablement. Nous proposons ici une liste non exhaustive de certains accidents majeurs[8] aux PNs en France :

- 8 Avril 1993 : 4 morts dans une collision entre un train et un bus scolaire près d'Aix en Provence ;
- 22 Septembre 1995 : 5 morts dans une collision entre un train et une voiture à Agde ;
- 8 Septembre 1997 : 13 morts dans une collision entre un autorail et un camion de fioul à Porte Sainte Foy ;
- 2 Juin 2008 : 7 morts et 31 blessés dans une collision entre un TER et un bus scolaire au passage à niveau de Mésinges, Haute-Savoie ;
- 12 Octobre 2011 : 2 morts et 48 blessés dans une collision entre un train et un camion au passage à niveau de Saint-Médard, Rennes.

6.3.2 Architecture du passage à niveau

L'architecture typique caractérisant la plupart des PNs automatiques (figure 6.5) est constituée de :

1. Une ou plusieurs voies ferroviaires,
2. Une voie routière,
3. Deux demi-barrières pour empêcher les usagers de la route de traverser en présence d'un train,
4. Des pédales d'annonce à l'entrée et à la sortie du PN pour détecter l'arrivé et le départ des trains,
5. Alarmes sonores et de feux de circulation routière pour alerter les usagers de la route de l'approche d'un train.

7. En France, on recense 15000 passages à niveau au 1er janvier 2011, source : `http://www.rff.fr/reseau/projets/modernisation/pan-805`
8. En 2010, 110 collisions aux passages à niveau ont eu lieu en France avec un bilan de 25 morts.

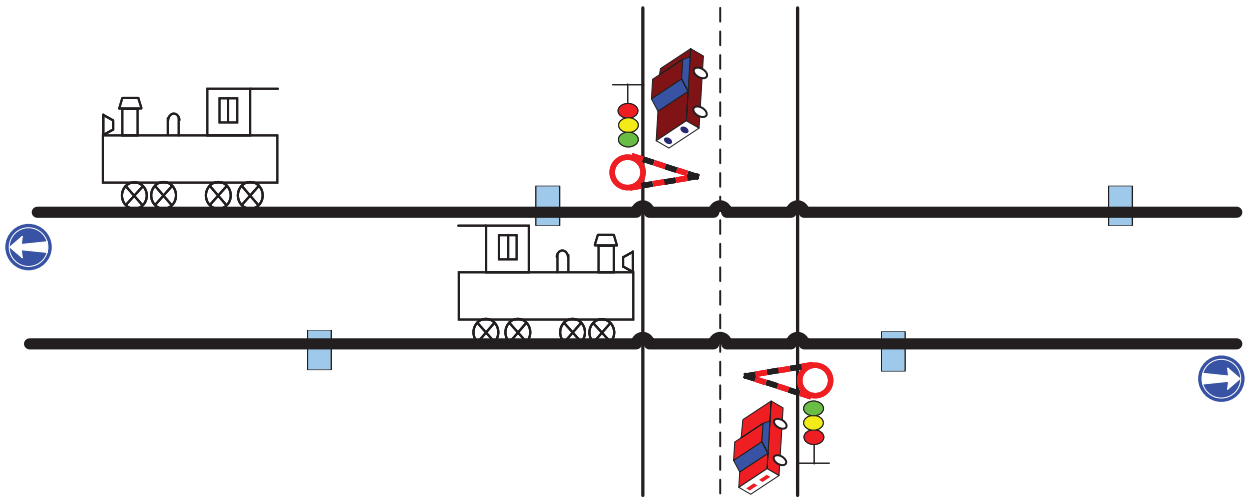

FIGURE 6.5: Architecture actuellement utilisée pour une grande partie du PN

Le fonctionnement d'un PN est décrit comme suit : quand la pédale d'entrée au PN détecte le passage d'un train, les alarmes sont déclenchées, les feux de circulation passent au rouge et les barrières sont baissées. Le PN est ouvert à la circulation routière dès que le train active la pédale de sortie et si aucun autre train n'arrive entre temps. Ainsi, les alarmes sont arrêtées, les feux de circulation sont éteints et les barrières sont levées.

6.3.3 Deux scénarios critiques au passage à niveau

Bien que la majorité des accidents aux passages à niveau soient dus au non respect du code de la route, une partie non négligeable de ces accidents est due au fonctionnement ainsi qu'à l'architecture du système de protection utilisée. En effet, à travers cette architecture, nous abordons deux scénarios de fonctionnement qui sont à l'origine d'un nombre important d'accidents : la durée d'ouverture trop courte (DOC) ainsi que la durée de fermeture trop longue (DFL) du PN.

6.3.3.1 Durée d'ouverture trop courte

La DOC est considérée comme l'une des situations dangereuses au PN avec deux voies ferroviaires. En effet, même si les trains circulant dans la même direction sont suffisamment espacés, les trains circulant dans des directions opposées peuvent arriver au PN indépendamment les uns des autres. Ainsi, quand un train quitte le PN, la barrière est ouverte à la circulation routière, puis fermée dès qu'un autre train est détecté. Par conséquent, dans le cas où un autre train arrive très rapidement par la direction opposée, le cycle de fermeture est re-enclenché aussi tôt. Ainsi, certains usagers de la route traversant le PN, peuvent paniquer et prendre de mauvaises décisions.

Les causes directes des accidents sont souvent données comme les causes réelles, alors que dans certains cas la cause réelle réside ailleurs. Par exemple, parmi les causes on trouve "un véhicule arrêté sur la zone de croisement". Bien évidement, c'est la cause directe de certains accidents. Néanmoins, plusieurs scénarios peuvent mener à cette situation, comme la fréquence du trafic routier sur le PN, les délais utilisés pour la signalisation, etc. De plus, étant donné que le facteur humain est la principale cause des accidents/incidents aux PN (plus de 95% à travers

l'Europe [Slovak 2007]), certains aspects techniques, comme la durée d'ouverture, sont souvent ignorés dans les rapports des accidents. En fait, même si la DOC n'est pas la principale situation dangereuse sur le PN, ce scénario a été clairement pointé par un certain nombre d'experts ferroviaires et routiers. Parmi les références, on trouve un article publié dans le magazine "Evening Gazette" en date de février 2009 [9]. L'article est un témoignage des usagers de la route victimes d'un incident à Billingham en Royaume-Uni, dans lequel la DOC a été donnée comme la cause de cet incident. De plus, un rapport technique intitulé "Adamstown Level Crossing - Operational Review" [10] souligne la DOC comme une cause potentielle d'accidents au PN. Dans ce rapport, les auteurs expliquent comment la DOC peut génèrer des files d'attente du trafic sur le PN, et que parfois, les voitures n'auraient pas assez de temps pour libérer la zone de croisement en toute sécurité.

6.3.3.2 Durée de fermeture trop longue

Comme la DOC, la DFL est l'une des situations dangereuses au PN. En fait, les architectures actuellement utilisées sont déployées en utilisant la vitesse maximale autorisée sur le PN afin de garantir des délais minimum d'annonce. Par conséquent, la barrière pourra être maintenue fermée, inutilement, pour une durée trop longue quand un train lent traverse le PN. Par exemple, supposons que la vitesse maximale autorisée est $160km/h$ avec un délai d'annonce de 25 secondes. Dans le cas où un train de marchandises circulant à $40km/h$ traverse le PN, ce dernier sera maintenu fermé pendant 100 secondes. Ainsi, les usagers de la route impatients pourraient prendre le risque de contourner (passer en chicane) les (demi) barrières. Nombreuses sont les études qui pointent l'imprudence ainsi que l'impatience des usagers de la route comme principaux facteurs d'accidents/incidents sur le PN [Taylor 2008]. Un sondage mesurant l'impatience des usagers de la route mentionne que, parmi plus de 4400 usagers de la route qui ont participé, à l'enquête présentée dans [Taylor 2008] 24% d'entre eux ont déclaré ne pas avoir respecté les règles de circulation sur un passage à niveau une ou plusieurs fois.

Néanmoins, nous ne sommes pas les premiers à tenter de proposer des solutions pour la problématique de DFL. Des recherches ont été menées et certaines technologies telles que HXP-3 et Westex GCP 3000 [11], ont été développées afin d'eviter les scénarios menant à la DFL.

Après avoir introduit les deux scénarios sur lesquels nous nous focalisons, nous détaillons dans la suite l'architecture du PN que nous proposons.

6.3.4 Proposition

6.3.4.1 Description

Nous proposons une architecture pour un système de protection automatique (SPA) capable d'éviter la DOC et DFL dans le cadre d'une exploitation en ERTMS niveau 2. Concrètement, le nouveau SPA repose sur l'architecture donnée en section 6.3.2 et utilise certaines informations provenant du système ERTMS, à savoir la position du train ainsi que sa vitesse. Ces deux informations permettent d'ajuster les délais d'ouverture et de fermeture de manière à éviter les deux scénarios en question. Ces deux derniers peuvent être exprimés en langage naturel comme suit :

9. http://ts20.gazettelive.co.uk/local-news/billingham-mum-tells-of-terror-on-level-crossing/
10. http://www.transport.nsw.gov.au/sites/default/file/news/Adamstown-Level-Crossing-Review.pdf
11. "TCA Risk Model Trial : Level crossing predictors train detection performance", http://www.rssb.co.uk/sitecollectiondocuments/pdf/reports/research/T579_lcp_final.pdf

- DOC : une fois ouvert à la circulation routière, le PN doit rester ouvert au moins T_{min} unités de temps, avec T_{min} étant la durée minimale d'ouverture que nous fixons.
- DFL : une fois fermé et quand il n'y a pas de train qui approche, le PN doit être maintenu fermé au moins T_{inf}, et au plus T_{sup} unités de temps, avec T_{inf} et T_{sup} sont les bornes que nous fixons pour le délai d'annonce.

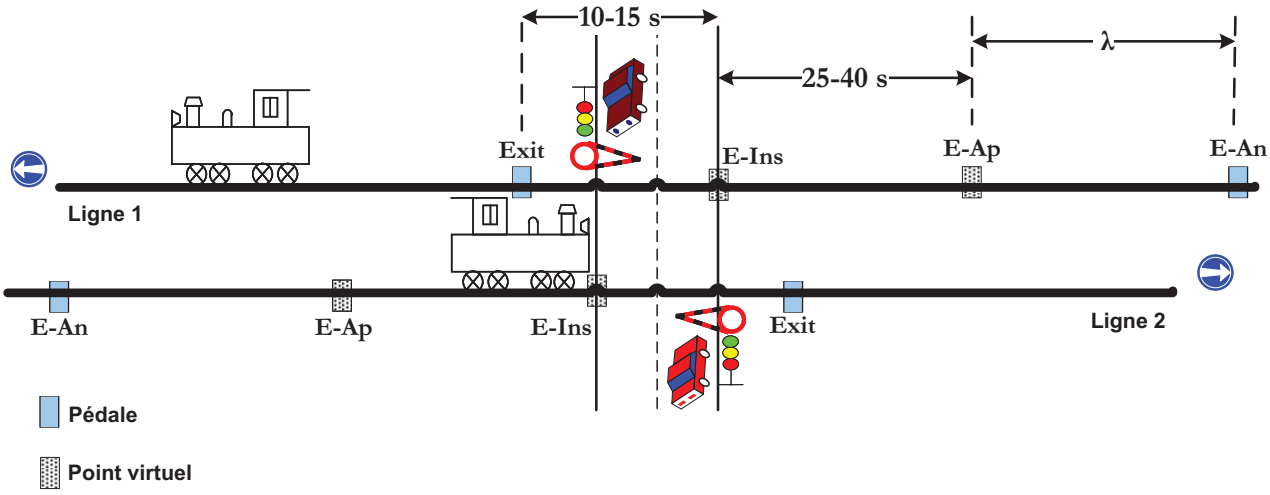

FIGURE 6.6: Pédales physiques et pédales virtuelles

L'idée est de placer la pédale de détection du train (E-An$_i$) avant sa position habituelle (E-Ap$_i$) pour chaque direction, comme le montre la figure 6.6 (avec $i \in \{1, 2\}$). Par conséquent, quand un train **T1** quitte la zone de croisement, la pédale de détection (E-An$_i$) nous permet de détecter si un autre train **T2** arrive très vite dans la direction opposée et ainsi éviter les scénarios qui pourraient mener à la DOC. Par ailleurs, la E-Spd$_i$ correspond à l'instant où le RBC retourne la vitesse du train approchant. Cette vitesse permettra d'ajuster les délais de fermeture de manière à éviter le scénario DFL. En pratique, cette vitesse est obtenue à l'aide de l'odomètre et la lecture du dernier groupe de balises.

Supposons que le nouveau SPA suggère que le déclenchement du cycle de fermeture du PN doit être déclenché entre 25 et 40 secondes avant l'arrivée du train sur la zone de croisement (figure 6.6). Néanmoins, étant donné que les emplacements [12] physiques des pédales sont fixes, le cycle de fermeture déclenché par des trains lents doit être retardé par une certaine durée après l'activation de E-An. Cela permet d'éviter la durée de fermeture excessivement longue du PN due aux trains lents (la DFL). De plus, la vitesse du train permet également de prévoir le délai d'arrivée des trains sur la section de croisement. Par conséquent, dans le cas où un autre train arrive très rapidement par la direction opposée, la connaissance de la vitesse de celui-ci permet de décider soit d'ouvrir les barrières de protection soit de les garder fermées. Cela permet d'éviter le scénario menant à la DOC.

6.3.5 Composition du passage à niveau

Le système de protection automatique au PN est composé de trois modules : le module détection des trains, le module de contrôle des barrières ainsi que le module coordination entre la détection des trains et le contrôleur des barrières. Ainsi, le système de contrôle-commande

12. en utilisant la vitesse maximale autorisée

du PN est composé de trois sous-systèmes : **SD** [13], **BP** [14] et **S2C** [15], qui opèrent en parallèle et se synchronisent sur les événements : *arrive*, *approach*, *exit*, *GoDown* et *GoUp*.

6.3.5.1 Système de détection

Lorsqu'un train s'approche du PN, il active la première pédale E-An1 (E-An2 pour la direction opposée). Cela provoque l'envoi du signal *arrive* par **SD** à **S2C**. De la même manière, quand le train arrive au point virtuel E-Ap1 (resp. E-Ap2), le signal *approach* est envoyé par **SD** à **S2C**. Finalement, lorsque le train quitte la zone de croisement, il active la pédale de sortie Exit1 (resp. Exit2) et le signal *exit* est envoyé par **SD** à **S2C**.

Remarque 6.5 *Dans la figure 6.6,E-Ap et E-Ins sont des points virtuels (ou pédales virtuelles). Concrètement, la pédale E-Ins correspond à l'instant où le train rentre dans la section critique. Finalement, la pédale E-Ap correspond à l'instant de déclenchement du cycle de fermeture du PN.*

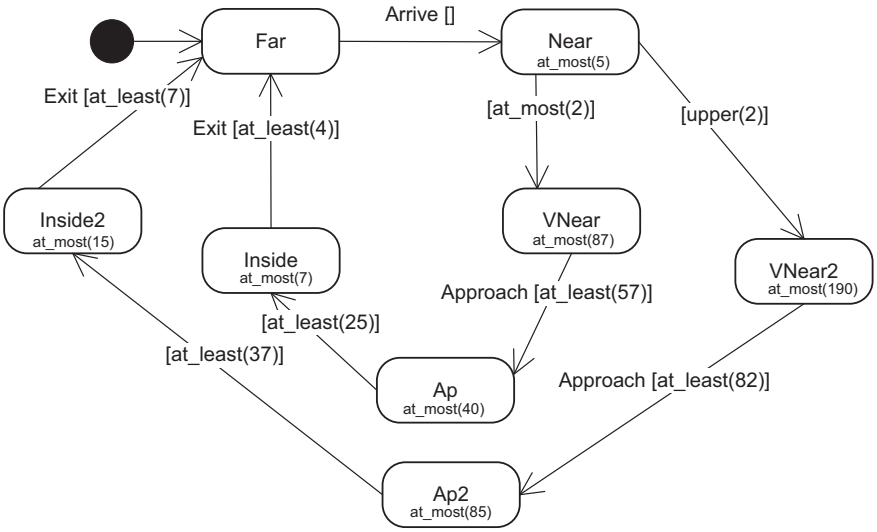

FIGURE 6.7: Modèle de sous-système "système de détection" pour une décomposition de la vitesse en deux intervalles

Plusieurs catégories de trains peuvent circuler sur la section considérée (trains passagers, trains marchandises, ...). Ainsi, selon sa catégorie, un train met λ secondes dans la première section (entre E-An$_i$ et E-Ap$_i$), "α'' secondes (avec "$\alpha'' \in [25, 40]$) dans la deuxième section (entre E-Ap$_i$ et la zone de croisement), et "β'' (avec "$\beta'' \in [10, 15]$) secondes dans la troisième section (entre la zone de croisement et la pédale Exit$_i$). Les pédales sont installées en utilisant la vitesse maximale autorisée sur la ligne. Nous supposons que la vitesse des trains circulant sur le PN varie de $50km/h$ et jusqu'au $160km/h$. Le module SD utilise certaines informations fournies par l'EVC qui se charge du calcul de la vitesse des trains.

13. Système de Détection
14. Barrières de Protection
15. Système de Coordination et de Contrôle

6.3.5.2 Système de coordination et de contrôle

Le **S2C** gère le PN de manière à avoir pour tous les trains un déclenchement du cycle de fermeture entre 25 et 40 secondes avant son arrivée sur la zone de croisement. Par conséquent, étant données les différentes catégories de trains traversant le PN, avec différentes vitesses, le déclenchement du cycle de fermeture peut être retardé avec un délai inversement proportionnel à la vitesse du train.

Afin de simplifier la présentation du comportement du sous-système **S2C** (figure 6.8), nous considérons que la vitesse du train appartient à l'intervalle $[50, 160]$ km/h. Par ailleurs, le retard de lancement du cycle de fermeture est inversement proportionnel à la vitesse du train. Cependant, afin de considérer les différentes valeurs de vitesse, nous divisons l'intervalle de vitesse en "n" intervalles avec une marge de sécurité de $5km/h$ pour chaque intervalle. En pratique, cette marge de sécurité est rajoutée afin de prendre en compte une possible accélération après avoir calculé sa vitesse mais, dans la limite de $BorneMax+5km/h$. Par exemple, pour une décomposition en trois intervalles $[50, 80[$, $[80, 110[$ et $[110, 160]$ km/h. Si la vitesse du train, retournée par par l'EVC, est de $90km/h$ ($[80, 110[$), alors celui-ci peut rouler à au maximum $115km/h$ tant qu'il n'a pas quitté le PN. Une marge peut être calibrée de façon adéquate lors d'une implémentation concrète.

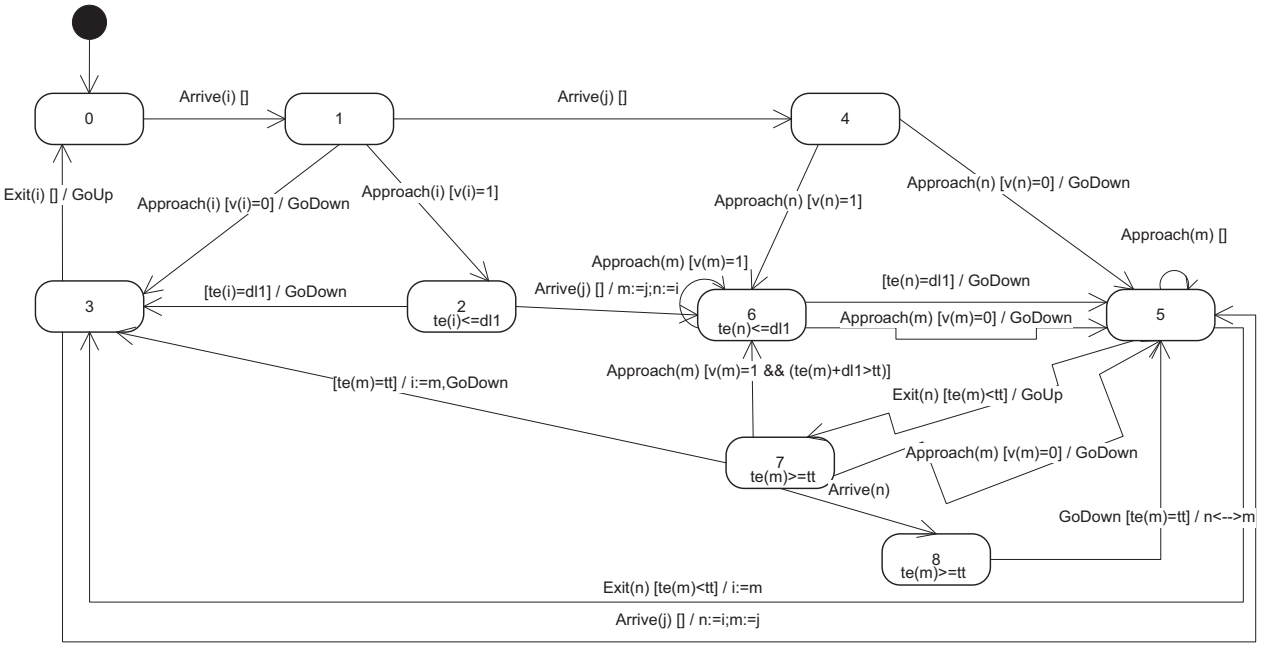

FIGURE 6.8: Modèle de sous-système "système de coordination et de contrôle" ($i, j \in \{0, 1\}$, $n, m \in \{i, j\}$ et $m = 1 - n$)

Le modèle SM donné dans la figure 6.8, est une version simplifiée du modèle comportemental de **S2C**. Dans ce modèle, deux intervalles de vitesses sont utilisés, $[50, 110[km/h$ et $[110, 160]km/h$. Concrètement, le **S2C** fait appel à des variables globales pour enregistrer la vitesse ($v(i)$) du train et le temps écoulé ($te(i)$) depuis son arrive. Ensuite, nous avons utilisé une troisième variable globale qui est le temps nécessaire par celui-ci au point virtuel de démarrage de cycle de fermeture. Le calcul de cette dernière variable dépend de la vitesse du train. L'ensemble de ces trois variables sont mis à jour par le RBC dont le **S2C** y accède pour lire la valeur. Afin de donner un aperçu du modèle, un scénario simple est explicité ici : quelques secondes avant qu'un train A quitte le PN, un deuxième train B est détecté. La vitesse de B,

$v(B)$, va conditionner l'ouverture ou non du PN à la circulation routière. Concrètement, si le train B nécessite un temps tt supérieur à $(te(B)+Tmin)$ pour atteindre la zone de croisement, le **S2C** décide d'ouvrir le PN au trafic routier. Sinon dans le cas contraire, il décide de le maintenir fermé. Bien sûr tout en respectant l'exigence DOC ainsi que le délai de fermeture qui doit être de 25 à 40 secondes avant l'arrivée du train sur la zone critique. T_{min} étant la durée minimale entre les cycles de fermeture successifs du PN.

Le **S2C** envoie un signal *GoDown* au **BP** dl secondes après la réception du signal *Arrive*, et il envoie un signal *GoUp* immédiatement lorsque il reçoit le signal *Exit*, à condition bien sûr, qu'aucun autre train ne soit détecté par la direction opposée. Le délai dl est calculé manuellement et dépend de la vitesse du train. Il est important de rappeler que l'idée est que **S2C** gère le PN de telle manière que chaque train met au moins 25 secondes et au plus 40 secondes entre le déclenchement du cycle de fermeture des barrières et son arrivée sur la zone de croisement. Ainsi, pour les trains rapides (dans notre cas, les trains dont la vitesse mesurée $\in [110, 160]km/h$), le cycle de fermeture est lancé dès que le train arrive au point virtuel E-Ap (et *"Approach"* est envoyé) et par conséquent $dl(i) = 0$ secondes. Cependant, pour les trains lents (les trains dont la vitesse mesurée $\in [50, 110[km/h$), le cycle de fermeture est lancé après l'activation de E-Ap. $dl = tt(\Sigma$ [16], $115km/h)$ - 25 s = 9 secondes. Ainsi, le cycle de fermeture est déclenché avec un retard de 9 secondes pour les trains circulant avec une vitesse $\in [50, 110[km/h$.

Le délai dl est utilisé pour éviter les scénarios DFL ainsi que les scénarios DOC. Par exemple, quand un train (T1) quitte la zone de croisement et un autre train (T2) se rapproche par l'autre direction, le retard dl couplé avec la vitesse de T2 permet de déterminer le moment de déclenchement du cycle de fermeture des barrières, dans le cas où elles ont été ouvertes après le passage de T1. De cette façon, le PN est réouvert suite au départ de T1 si et seulement si nous nous assurons que les barrières seront maintenues ouvertes pendant au moins T_{min}. Concrètement, pour une décomposition en deux intervalles, dl peut avoir deux valeurs : dl_0 = 0 secondes et dl_1 = 9 secondes. Ces délais sont associés aux train roulant avec une vitesse appartenant respectivement à l'intervalle $[110, 160]km/h$ et à l'intervalle $[50, 110[km/h$.

Remarque 6.6 *Dans le modèle SM du **S2C**, la condition $v(i) = 1$ exprime que le train i roule avec une vitesse appartenant à l'intervalle $[50, 110[km/h$. De la même manière, la condition $v(i) = 0$ exprime que le train i roule avec une vitesse appartenant à l'intervalle $[110, 160]km/h$.*

6.3.5.3 Barrière de protection

Le **BP** répond à *GoDown* en lançant la fermeture de barrières qui nécessite 6 secondes pour les fermer complètement. Avant de lancer le cycle de fermeture, la **BP** utilise un délai fixe de "4″ secondes pour alerter les usagers de la route avec l'alarme sonore et les feux de circulation. De la même façon, il répond à *GoUp* en déclenchant l'ouverture des barrières qui nécessitent aussi 6 secondes pour les ouvrir complètement (figure 6.9). Par ailleurs, nous supposons qu'il n'y a pas de chevauchement entre les trains circulant dans la même direction, ce qui signifie que le **S2C** gère au plus deux trains simultanément : au maximum un train dans chaque direction.

6.4 Démarche de validation

Les emplacements physiques des pédales d'annonce sont déterminés en se basant sur la vitesse maximale autorisée sur la zone PN. Pour une vitesse maximale de $160km/h$, la distance

16. Σ est la distance entre E-An et la zone de croisement.

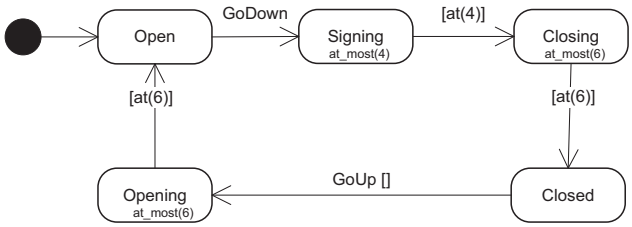

FIGURE 6.9: Modèle de sous-système "Barrière de protection"

entre le point virtuel E-Ap et la zone de croisement est égal à 1200 m (afin de garantir un délai minimum de 25 secondes comme mentionné précédemment), et la distance entre la zone de croisement et la pédale Exit est égale à 200 m. Enfin, la distance (λ) entre la pédale E-An et le point virtuel E-Ap, choisi en utilisant la vitesse maximale, est une variable dont la valeur est inconnue et qu'on cherche à déterminer. Pour commencer, nous avons choisi aléatoirement une distance de $500m$.

Remarque 6.7 *Comme on s'intéresse ici à un mode d'exploitation en d'ERTMS niveau 2, la communication entre les différents modules est effectuée via GSM-R. Ainsi, une intégration concrète du PN aux SRS d'ERTMS nécessite une étape complémentaire d'étude dont l'objectif est d'établir des formats standards d'échange d'informations entre les différents modules.*

6.4.1 Durée d'ouverture trop courte

Suite au passage d'un train, une fois le PN reouvert à la circulation routière, il doit rester ouvert au moins T_{min} unités de temps. Sur la base des modèles donnés ci-dessus, cette exigence peut être exprimée comme suit : *Globaly* "*GoDown*" **cannot occur Before a delay of** "T_{min}" **after** "*GoUp*". Cette exigence est identifiée comme de type "Interdiction Réponse Délai Max" et le pattern d'observation associé à ce type d'exigences sera utilisé. Concrètement, le pattern d'observation en question est instancié avec les paramètres appropriés : "GoUp" comme l'événement de référence et "GoDown" comme l'événement surveillé pour générer l'automate observateur donné en figure 6.10.

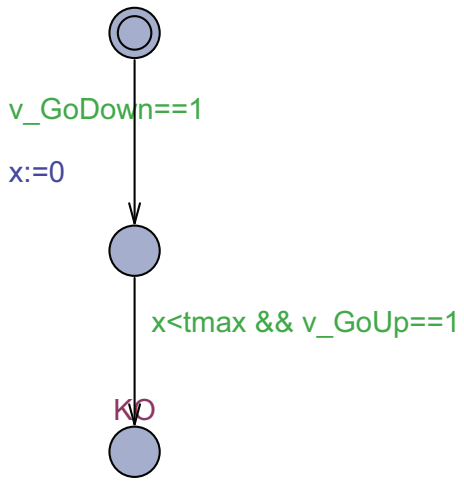

FIGURE 6.10: Pattern d'observation associé à l'exigence DOC

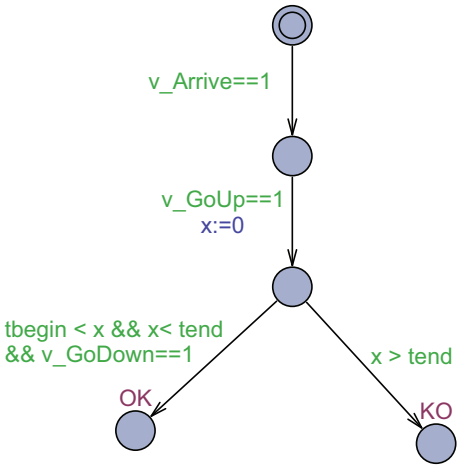

FIGURE 6.11: Pattern d'observation associé à l'exigence DFL

6.4.2 Durée de fermeture trop longue

Le second scénario que nous traitons à travers cette nouvelle architecture est la DFL. En fait, comme expliqué précédemment, la nouvelle architecture prend en compte la vitesse des trains afin d'interdire un tel scénario. L'exigence associée à ce scénario peut être définie comme suit : "une fois fermé et quand il n'y a pas de train qui approche sur la direction opposée, le PN doit être maintenu fermé au moins $29s$, et au plus $55s$". Basé sur le modèle formel donné dans la figure 6.7 et la figure 6.9, cette exigence peut être exprimée comme suit : ***After Arrive** "GoUp" **must occur between a delay of 29 seconds and a delay of 55 secondes over** "GoDown"*. Par conséquent, le pattern d'observation associé à "*Réponse Entre*" est utilisé avec le *scope after* pour surveiller la propriété. Concrètement, ce pattern est instancié avec les paramètres appropriés : "Arrive", "GoUp" et "GoDown" afin d'obtenir un automate observateur donné en figure 6.11.

Considérons le cas où la valeur désirée de T_{min} est fixée à 55 secondes (le délai min séparant les cycles de fermeture successifs de PN). Les observateurs obtenus sont synchronisés avec le modèle comportemental du système généré à travers l'algorithme de transformation. Ensuite, nous appliquons la procédure détaillée dans le chapitre 5 : le fichier XML obtenu par la traduction est importé dans l'outil de vérification Uppaal pour générer un modèle qui sera synchronisé avec les observateurs générés pour surveiller les exigences DOC et DFL. Par la suite, une formule CTL qui exprime l'accessibilité des états d'erreur relatifs aux deux observateurs est définie. Finalement, Uppaal effectue la vérification de la validité de cette formula CTL.

Ici, les deux états KO relatifs aux patterns DOC et DFL sont accessibles. Dans la suite de cette section, nous présenterons la procédure à suivre pour valider ces deux exigences.

6.4.3 Calcul numérique

6.4.3.1 Calcul de λ

L'objectif est de déterminer la distance *"λ"* séparant la nouvelle position de la pédale d'annonce du train (E-An) par rapport à sa position habituelle (E-Ap), de telle manière à satisfaire l'exigence liée à la DOC puisque la distance *"λ"* est proportionnelle au retard T_{min} que nous fixons. Or pour des raisons de sécurité, λ doit être calculée en utilisant la vitesse maximale autorisée sur cette section du PN. Néanmoins, plus la pédale E-An est éloignée de sa position habituelle E-Ap, plus nous assurons que la DOC est évitée. Cependant, λ ne doit pas être surdimensionnée parce que cela pourrait dégrader le niveau de sécurité au PN en générant des longues durées de fermeture (DFL). En outre, nous cherchons à trouver la valeur minimale de λ satisfaisant DOC. Concrètement, la distance λ est proportionnelle à T_{min}. Ainsi, plus la valeur de λ est grande, plus on a d'assurance pour la satisfaction de la DOC. Par conséquent, un point de départ pour la recherche dichotomique pourrait être une grande valeur pour λ satisfaisant l'exigence liée à la DOC (le pseudo-code de l'algorithme de recherche, donné dans la suite, explique comment cette valeur de départ pourrait être définie). Ensuite, l'algorithme de recherche est déroulé afin de trouver la valeur minimale de λ qui vérifie la DOC. L'avantage de cette technique est qu'elle permet de localiser rapidement la valeur optimale.

Pour une valeur de T_{min} est fixée à 55 secondes, l'algorithme de recherche est déroulé comme suit : d'abord doubler itérativement la valeur de λ jusqu'à trouver une valeur de λ qui satisfait DOC. La première valeur satisfaisant DOC est de $64s$. Basée sur cette valeur, la recherche est effectuée sur l'intervalle $[32, 64]$. À la fin du ce processus, la valeur retournée par l'algorithme est $57 secondes$.

Algorithme 1 Calcul de λ

N: *integer*
λ: *float*
$\lambda \leftarrow \varepsilon$ //ε est une variable de type entier dont la valeur est strictement positive mais, en même temps, infiniment petite
tant que $(\lambda \nvDash DOC)$ **faire**
  $\lambda \leftarrow \lambda \times 2$
$N \leftarrow$ **Floor**$(\lambda/2)$ //**Floor** est la partie entière de la division
tant que $((\lambda - 1) > N)$ **faire**
  $halfway \leftarrow$ **Floor**$((\lambda + N)/2)$
  si $(halfway \vDash R1)$ **alors**
    $\lambda \leftarrow halfway$
  sinon
    $N \leftarrow halfway$
si $((\lambda - 1) \vDash R1)$ **alors**
  retour $(\lambda - 1)$
sinon
  retour λ

Étant donné que λ est déterminée en utilisant la vitesse maximale autorisée de la ligne, ici $160km/h$, alors la distance optimale qui sépare la nouvelle position de la pédale d'annonce par rapport à sa position habituelle, dans la direction d'arrivée, est $160km/h \times 57$ s $= 2533,33m$. Ainsi, cette pédale sera placée à une distance de $160km/h \times (57{+}25)$ s $= 160km/h \times (82)$ s $= 3644,44m$ par rapport à la zone de croisement. En pratique, une marge de sécurité pourra ajoutée aux différentes valeurs obtenues.

6.4.3.2 Décomposition de la vitesse

Le résultat de l'étape de vérification de cette exigence permet de définir la décomposition en intervalles de la vitesse. En pratique, comme expliqué dans la section 6.3.5.2, la première itération est faite avec deux intervalles de vitesse du train, à savoir $[50, 110[$ et $[110, 160]$ km/h. Ensuite, nous vérifions la satisfaction de l'exigence de DFL. Dans le cas où cette première itération valide l'exigence DFL, la décomposition est retenue comme la solution. Dans le cas inverse, nous divisons la vitesse en trois intervalles, et nous vérifions l'exigence liée à la DFL de nouveau. Ce processus est répété jusqu'à ce qu'une décomposition de la vitesse qui valide la DFL soit trouvée. Évidemment, plus nous décomposons la vitesse en intervalles, plus nous aurons une garantie de la validation de DFL. Les différentes étapes que nous avons utilisées afin de définir le comportement de **S2C** sont énumérés ci-après (algorithme "étapes de la recherche de décomposition de la vitesse").

À la fin, la décomposition en intervalles obtenue et qui valide la DOC et la DFL est : $[50, 55[$, $[55, 60[$, $[60, 65[$, $[65, 70[$, $[70, 80[$, $[80, 110[$ et $[110, 160]$. Basée sur cette décomposition de vitesse, le retard qui devrait être ajouté au déclenchement de cycle de fermeture du PN (voir la section 6.3.5.2) est donné dans le tableau 6.1.

6.5 Conclusions

Dans ce chapitre, nous avons proposé une nouvelle architecture d'un système de protection automatique aux PNs. L'objectif de cette architecture est d'intégrer les PNs aux spécifications

Algorithme 2 Étapes de la recherche de décomposition de la vitesse

Étape 0: "$n \leftarrow 2''$,
Étape 1: Décomposer la vitesse autorisée du train sur le PN en "n'' intervalles,
Étape 2: Calculer la vitesse considérée pour chaque intervalle (rajouter une marge de sécurité de $5km/h$ à la borne max de chaque intervalle),
Étape 3: Pour chaque valeur de vitesse considérée, calculer le délai "m'' utilisé pour retarder le cycle de fermeture du PN,
Étape 4: Donner des nouveaux modèles pour **SD** et **S2C**,
Étape 5: Vérifier la DFL,
Étape 6:
si DFL est validée **alors**
  Stop
sinon
  $n \leftarrow n + 1$
  Aller à l'étape 1

Vitesse du Train (Km/h) $\in$	Vitesse Considérée (Km/h)	Délai "m" (secondes)
$[50, 55[$	60	$47 + \lambda = 104$
$[55, 60[$	65	$41 + \lambda = 98$
$[60, 65[$	70	$36 + \lambda = 93$
$[65, 70[$	75	$32 + \lambda = 89$
$[70, 80[$	85	$21 + \lambda = 78$
$[80, 110[$	115	$9 + \lambda = 66$
$[110, 160]$	165	$0 + \lambda = 57$

Table 6.1: Décomposition en intervalles de vitesse

SRS d'ERTMS niveau 2. L'avantage de l'architecture proposée est qu'elle permet d'éviter deux scénarios ont été identifiés comme potentiellement dangereux. À travers ce cas d'étude, nous avons pu illustrer les différents mécanismes de notre contribution.

Chapitre 7

Conclusions : retour sur les contributions

Résumé : *Bien que des progrès incontestables aient été réalisés pour la mise en œuvre de l'ingénierie système, cette étape doit toujours faire face à certains défis inhérents aux types d'activités entreprises dans ce domaine. Ainsi, le développement de moyens à même d'aider à surmonter les difficultés rencontrées dans ces activités, demeure indispensable notamment au regard de la complexité croissante des systèmes.*

Dans ce dernier chapitre, nous rappelons les principales contributions de ce travail de thèse relatives aux étapes de spécification des exigences, de la modélisation et de la vérification. Ensuite, nous dressons les principales perspectives qui se dégagent de notre travail.

Sommaire

7.1 Conclusions

La maîtrise des systèmes complexes exigeants en qualité et en sûreté de fonctionnement est l'une des préoccupations actuelles en ingénierie de systèmes vu le coût matériel et parfois aussi humain engendré par le dysfonctionnement de ces systèmes. Le comportement de tels systèmes est généralement caractérisé par des contraintes de temps. Ces contraintes temporelles peuvent refléter des propriétés de sûreté de fonctionnement comme, par exemple, la sécurité et la disponibilité. Le respect des ces contraintes est ainsi indispensable lors de la conception de ces systèmes. Néanmoins, la phase de spécification ainsi que la phase de vérification et validation de ces exigences restent très souvent ardues, d'une part par le niveau d'expertise nécessaire pour l'expression et la vérification de ces exigences, et d'autre part par le manque de moyens d'assistance.

Afin de cadrer et assister l'utilisateur au long du cycle de développement de systèmes à contraintes de temps, une démarche pour la spécification et la vérification des propriétés temporelles a été proposée dans le cadre de notre travail. Dans cette démarche, nous avons commencé par étudier les différents types d'exigences temporelles. Suite à cette étape nous avons proposé une nouvelle classification d'exigences temporelles dont les principaux avantages sont la grande diversité des exigences qu'elle couvre ainsi que la prise en compte à la fois des exigences portant sur des événements et celles portant sur des états d'une manière homogène. Ensuite, en se basant sur cette classification, nous avons développé une base de patterns d'observation. Concrètement, pour chaque classe d'exigence, nous avons associé un *automate observateur* afin de jouer le rôle d'un *chien de garde* (*watchdog* en anglais) pour la surveiller. Chaque observateur dispose d'un ensemble de nœuds dits d'erreur (***KO***) relatifs à la violation de l'exigence associée. Ces deux étapes, à savoir la classification des exigences temporelles et la base de patterns d'observation, sont développées une fois pour toutes et elles présentent l'avantage d'être génériques (*context-free* en anglais) et extensibles i.e. peuvent être étendues pour prendre en compte de nouvelles classes d'exigences temporelles.

En pratique, la démarche que nous développons propose de cadrer l'utilisateur durant le processus d'expression des besoins à travers une grammaire d'expression des propriétés temporelles. Ces dernières sont exprimées par des assertions obtenues à partir de motifs pré-établis en langage naturel. Les assertions ainsi composées sont à la fois simples et précises. Le processus d'identification des exigences temporelles exprimées est par conséquent rendu automatique à partir des assertions produites. Par ailleurs, nous avons développé un algorithme qui permet la détection de différents types d'incohérences pré-établies dans un ensemble d'exigences exprimées sur la base de notre grammaire. Cet algorithme se base sur une représentation des exigences sous forme d'un graphe orienté.

Sur le volet *modélisation du comportement* notre contribution porte sur l'idée de donner la possibilité à l'utilisateur d'utiliser des notations semi-formelles considérées comme plus intuitive, ici des state-machine UML avec des annotations temporelles, et de proposer une transformation automatique vers des modèles AT sur lesquels la vérification des exigences sera déroulée. L'étape de transformation de modèles, assurée par un algorithme composée d'un ensemble de règles, a été validée au regard de la préservation de la sémantique. Ici, nous avons démontré une relation de bisimulation entre le modèle source SM et le modèle cible généré AT.

Dans la pratique, les exigences à vérifier sur un système donné sont exprimées en utilisant la grammaire de spécification que nous avons définie. À la fin de cette étape, un ensemble d'exigences est obtenu sur lequel une analyse de cohérence est appliquée. L'étape d'expression des exigences ainsi que l'activité de vérification de leur cohérence sont supportées par un outil logiciel adhoc que nous avons développé. Ensuite, pour chaque assertion le type de l'exigence

temporelle exprimée est (automatiquement) identifié et le pattern d'observation correspondant est instancié pour générer un observateur à l'exigence avec les paramètres appropriés. Les observateurs ainsi obtenus sont synchronisés avec le modèle AT du système. Ce modèle AT est lui-même obtenu par une étape de transformation à partir d'un modèle de comportement en SM avec des annotations temporelles. La vérification à proprement parlée consiste ainsi à analyser -par model-checking- si les états "KO" respectivement sur les différents observateurs sont accessibles.

L'illustration des nos différents contributions a fait l'objet du chapitre 6, à travers une proposition d'intégration du PN dans les spécifications du système de contrôle-commande et signalisation ERTMS, pour le niveau 2. La proposition en question décrit une architecture fonctionnelle d'un système de protection automatique aux passages à niveau. L'avantage de cette nouvelle architecture est qu'elle propose des solutions à certains scénarios potentiellement dangereux ignorés dans la plupart des systèmes de protection automatique actuels des PNs.

7.2 Perspectives

Dans cette section, nous proposons un bilan de perspectives dégagées de nos travaux ainsi qu'une revue des points qui peuvent être améliorés. Tout d'abord, il est à rappeler que l'étape de vérification dans notre approche se base sur la technique de *model-checking*. Or, la mise en œuvre de cette technique présente deux limitations majeures : (1) l'espace d'états du modèle à valider doit être fini et (2) l'explosion combinatoire du nombre d'états. Par ailleurs, notre approche repose sur une modélisation du système à vérifier. Cela empêche l'application directe de cette méthode pour les systèmes existants et pour lesquels nous ne disposons pas de modèle de comportement.

Dans la suite de cette section, nous listons les principales perspectives relatives aux différentes contributions.

7.2.1 Analyse de la cohérence des exigences

Dans sa version actuelle, l'analyse de la cohérence traite des exigences temporelles dont la référence est la première apparition d'un événement donné. Néanmoins, cette analyse doit être reformulée afin de prendre en compte les exigences dont la référence est la n^{ime} apparition d'un événement donné, c'est-à-dire, prendre en compte des événements répétitifs.

7.2.2 Algorithme de transformation

Comme nous l'avons montré dans ce manuscrit, l'utilité de la transformation de modèles est incontestable d'un point de vue pratique. Néanmoins, l'algorithme de transformation proposé peut être amélioré. En effet, un nombre considérable d'horloges est généré lors de la transformation. En fait, conformément à la spécification de la sémantique des SM, nous avons associé à chaque localité AT une horloge. Ainsi, une minimisation du nombre de timers permettra une réduction considérable du nombre d'états et ainsi simplifier le model-checking.

7.2.3 Base de patterns

Dans la pratique, nous avons remarqué que certaines propriétés peuvent être exprimées (1) comme une composition d'exigences ou (2) en faisant référence non pas à des événements élémentaires mais à d'autres exigences. Ces deux cas peuvent être facilement intégrés. En effet, la base de patterns d'observation est composée par un ensemble de patterns élémentaires

faisant référence aux événements. Cela facilite énormément l'extension de la base de patterns afin de prendre en compte ces deux variantes. Pour le premier, l'idée est de définir des *templates* génériques afin de modéliser certains types de composition qu'on peut avoir (comme **OR** et **XOR**) entre les patterns élémentaires. Pour le deuxième point, l'idée est de revoir la grammaire de spécification afin d'intégrer ce type de composition et de modifier les observateurs afin de rajouter des événements signalant le résultat de la surveillance d'exigences associées. Par exemple, rajouter deux événements (*OKevent*, *NOKevent*) pour chaque automate observateur afin de signaler la validation ou la violation de l'exigence associée.

7.2.4 Nouvelle architecture du système de protection automatique du passage à niveau

Rappelons ici que nous avons concentré notre contribution sur les aspects fonctionnels du système de protection du PN. En vue d'une implémentation pratique de l'architecture fonctionnelle proposée, les aspects matériels doivent être pris en compte notamment pour évaluer les caractéristiques FMDS. En effet, le système de protection automatique de PN est un système critique qui doit répondre à des exigences sévères de sûreté de fonctionnement (SIL4).

Bibliographie

[14th IFAC Symposium on Information Control problems in Manufacturing 2012] Bucharest Romania 14th IFAC Symposium on Information Control problems in Manufacturing INCOM'2012, editeur. Revisiting the interoperation relationships between systems engineering collaborative processes, 2012. (Cité en page 37.)

[Abid 2011] Nouha Abid, Silvano Dal Zilio et Didier Le Botlan. *A Real-Time Specification Patterns Language*. Rapport technique, 2011. (Cité en pages 133 et 134.)

[Abrial 1996] Jean-Raymond Abrial. The b book - assigning programs to meanings. Cambridge University Press, 1996. (Cité en pages 43 et 44.)

[Aceto 1998] Luca Aceto, Augusto Burgueno et Kim Larsen. *Model checking via reachability testing for timed automata*. In Bernhard Steffen, editeur, Tools and Algorithms for the Construction and Analysis of Systems, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer Berlin / Heidelberg, 1998. (Cité en pages 129, 132 et 133.)

[Aceto 2003] Luca Aceto, Patricia Bouyer, Augusto Burgueño et Kim G. Larsen. *The power of reachability testing for timed automata*. Theoretical Computer Science, vol. 300, pages 411–475, May 2003. (Cité en pages 132 et 133.)

[Allen 1984] James F. Allen. *Towards a general theory of action and time*. Artifcial Intelligence, vol. 1984, no. 23, pages 123–154, 1984. (Cité en pages 39 et 45.)

[Alur 1990] Rajeev Alur, Costas Courcoubetis et David L. Dill. *Model-Checking for Real-Time Systems*. In Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA, pages 414–425. IEEE Computer Society, 1990. (Cité en page 42.)

[Alur 1992] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford, CA, USA, 1992. (Cité en page 40.)

[Alur 1993a] Rajeev Alur, Costas Courcoubetis et David L. Dill. *Model-Checking in Dense Real-time*. Information and Computation, vol. 104, no. 1, pages 2–34, 1993. (Cité en page 42.)

[Alur 1993b] Rajeev Alur et Thomas A. Henzinger. *Real-time logics : complexity and expressiveness*. Information and Computation, vol. 104, pages 390–401, 1993. (Cité en page 40.)

[Alur 1994] Rajeev Alur et David L. Dill. *A theory of Timed Automata*. Theoretical Computer Science, vol. 126, pages 183–235, 1994. (Cité en pages 43 et 92.)

[Atitallah 2008] Rabie Ben Atitallah. Modèles et simulation des systèmes sur puce multiprocesseurs : estimation des performances et de la consommation d'énergie. Thèse de Doctorat de l'Université Lille 1, 2008. (Cité en page 132.)

[Atkinson 2003] Colin Atkinson et Thomas Kuhne. *Model-driven development : a metamodeling foundation*. IEEE Software, vol. 20, no. 5, pages 36–41, 2003. (Cité en page 65.)

[Audibert 2011] Laurent Audibert. *UML 2.0*. 2011. (Cité en page 120.)

[Barger 2009] Pavol Barger, Walter Schön et Mohamed Bouali. *A study of railway ERTMS safety with Colored Petri Nets*. In Guedes Soares & Martorell (eds) Bris, editeur, Reliability, Risk and Safety : theory and application The European Safety and Reliability Conference (ESREL'09), volume 2, pages 1303–1309, Prague Tchèque, République, 09 2009. Taylor & Francis Group. (Cité en pages 139 et 140.)

[Beizer 1990] Boris Beizer. Software testing techniques. New York : Van Nostrand Reinhold, 1990. (Cité en page 122.)

[Beizer 1995] Boris Beizer. Black-box testinge. John Wiley and Sons, 1995. (Cité en page 122.)

[Belmonte 2010] Fabien Belmonte, Jean-Louis Boulanger, Walter Schön et Karim Berkani. *Role of supervision systems in railway safety.* In G. Sciutto, editeur, Safety and Security in Railway Engineering, pages 59–68. WITPress, 2010. (Cité en page 23.)

[Berrada 2005] Ismail Berrada, Richard Castanet et Patrick Félix. *TGSE : Un outil générique pour le test.* In Colloque Francophone sur l'ingénierie des Protocoles (CFIP'05), pages 67–84, Bordeaux, France, 2005. Hermes. (Cité en page 92.)

[Berthomieu 2003] B. Berthomieu, P.-O. Ribet et F. Vernadat. *L'outil TINA – Construction d'espaces d'états abstraits pour les réseaux de Petri et réseaux Temporels.* Modélisation des Systèmes Réactifs, MSR'2003, 2003. (Cité en page 126.)

[Berthomieu 2010] Bernard Berthomieu, Florent Peres et François Vernadat. *Time Petri Nets, Analysis Methods and Verification with TINA.* In Modeling and Verification of Real-Time Systems, pages 19–49. ISTE, 2010. (Cité en page 126.)

[Beyer 1998] Dirk Beyer et Heinrich Rust. *Modeling a Production Cell as a Distributed Real-Time System with Cottbus Timed Automata.* In H. König et P. Langendörfer, editeurs, Tagungsband Formale Beschreibungstechniken für verteilte Systeme (FBT 1998, Cottbus, June 4-5), pages 148–159. Shaker Verlag, Aachen, 1998. (Cité en page 80.)

[Beyer 2001] Dirk Beyer et Heinrich Rust. *Cottbus Timed Automata : Formal Definition and Semantics.* In Proceedings of the 2nd IEEE/IFIP Joint Workshop on Formal Specifications of Computer-Based Systems (FSCBS 2001, pages 75–87, 2001. (Cité en page 80.)

[Beyer 2003] Dirk Beyer, Claus Lewerentz et Andreas Noack. *Rabbit : A Tool for BDD-Based Verification of Real-Time Systems.* In In : Computer-Aided Verification (CAV 2003), Volume 2725 of Lecture Notes in Computer Science, Springer-Verlag, pages 122–125. Springer-Verlag, 2003. (Cité en page 80.)

[Blanc 2005] Xavier Blanc. Mda en action ingénierie logicielle guidée par les modèles. Eyrolles, 2005. (Cité en pages 13, 65, 66, 68 et 69.)

[Boehm 1988] Barry W. Boehm. *A Spiral Model of Software Development and Enhancement.* Computer, vol. 21, no. 5, pages 61–72, Mai 1988. (Cité en page 22.)

[Bouali 2012] Mohamed Bouali, Pavol Barger et Walter Schön. *Colored Petri nets inversion for backward reachability analysis.* Reliability Engineering and System Safety, vol. 99, pages 1–14, march 2012. (Cité en page 42.)

[Boufenara 2010] Sabine Boufenara, Faiza Belala et Kamel Barkaoui. *Mapping UML 2.0 Activities to Zero-Safe Nets.* Journal of Software Engineering and Applications, vol. 3, no. 5, pages 426–435, 2010. (Cité en page 79.)

[Bouyer 2002] Patricia Bouyer. Modèles et algorithmes pour la vérification des systèmes temporisés. Thèse de Doctorat de Laboratoire Spécification et Vérification, ENS Cachan, 2002. (Cité en page 132.)

[Bouyer 2006] Patricia Bouyer et François Laroussinie. *Vérification par automates temporisés.* Systèmes temps réel 1 : techniques de description et de vérification, pages 121–150, 2006. (Cité en page 43.)

[Bowen 1995] Jonathan P. Bowen et Michael G. Hinchey. *Seven More Myths of Formal Methods.* IEEE Software, vol. 12, pages 34–41, 1995. (Cité en pages 25 et 44.)

[Boyer 2004] Marc Boyer et Jean-Christophe Pince. *Model-Checking aléatoire : une approche entre test et vérification*. In 11ème Jounrnée de Formalisation des Activités Concurrentes, 2004. (Cité en page 125.)

[Bézivin 2001] Jean Bézivin et Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework*. In ASE'01, Automated Software Engineering. IEEE Computer Society Press, 2001. (Cité en page 65.)

[Bézivin 2003] Jean Bézivin. *La transformation de modèles*. École d'Été d'Informatique CEA EDF INRIA 2003, vol. cours # 6, 2003. (Cité en pages 13, 66 et 67.)

[Bézivin 2004a] Jean Bézivin. *In Search of a Basic Principle for Model Driven Engineering*. CEPIS, UPGRADE, The European Journal for the Informatics Professional, vol. V, no. 2, pages 21–24, 2004. (Cité en page 65.)

[Bézivin 2004b] Jean Bézivin. *Sur les principes de base de l'ingénierie des modèles*. RSTI-L'Objet, vol. 10, no. 4, pages 145–157, 2004. (Cité en pages 65 et 68.)

[Bézivin 2005] Jean Bézivin. *On the Unification Power of Models*. Software and System Modeling (SoSym), vol. 4, no. 2, pages 171–188, 2005. (Cité en page 65.)

[Cansell 2003] Dominique Cansell. Assistance au développement incrémental et à sa preuve. Habilitation à Diriger les Recherches de l'Université Henri Poincaré, 2003. (Cité en page 124.)

[Cassez 2003] Franck Cassez. *Vérification qualitative – Model-Checking et Logiques Temporelles*. 2003. Toulouse. (Cité en pages 71 et 72.)

[CDT 2002] Coq-Development-Team CDT. *The Coq Proof Assistant Reference Manual Version 7, INRIA*, 2002. (Cité en page 124.)

[CE 2005] Commission Européenne CE. *Sur le déploiement du système européen de signalisation ferroviaire ERTMS/ETCS, COM(2005) 298 final*. Rapport technique, Commission des Communautés Européennes, 2005. (Cité en page 139.)

[CE 2006] Commission Européenne CE. *ERTMS - pour un trafic ferroviaire fluide et sûr : un grand projet industriel européen*. Rapport technique, 2006. (Cité en pages 139 et 140.)

[Chapurlat 2007] Vincent Chapurlat. Vérification et validation de modèles de systèmes complexes : application à la modélisation d'entreprise. Habilitation à Diriger les Recherches de l'Université de Montpellier II, 2007. (Cité en pages 37, 40 et 124.)

[Cheikh 2009] Fahima Cheikh. Composition de services : Algorithmes et complexités. Thèse de Doctorat de l'Université de Toulouse, Paul Sabatier, 2009. (Cité en page 74.)

[Cherif 2012] Ismail Cherif. Outil logiciel pour la spécification et la vérification des exigences temporelles dans les systèmes critiques complexes. Rapport de stage ingénieur, INSAT Tunisie, 2012. (Cité en page 54.)

[Clarke 1986] Edmund M. Clarke, E. Allen Emerson et A. Prasad Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems, vol. 8, no. 2, pages 244–263, 1986. (Cité en pages 39 et 42.)

[Clarke 1994] Edmund M. Clarke, Orna Grumberg et David E. Long. *Model checking and abstraction*. ACM Transactions on Programming Languages and Systems, vol. 16, pages 1512–1542, September 1994. (Cité en page 126.)

[Combemale 2008] Benoît Combemale. Ingénierie Dirigée par les Modèles (IDM) – état de l'art. 2008. (Cité en page 65.)

[Crane 2005] Michelle L. Crane et Juergen Dingel. *On the Semantics of UML State Machines : Categorization and Comparision.* In In Technical Report 2005-501, School of Computing, Queen's, 2005. (Cité en page 88.)

[Dasarathy 1985] Balakrishnan Dasarathy. *Timing constraints of real-time systems : construct for expressing them, methods of validating them.* IEEE Transaction Software Engineering, vol. 11, pages 80–86, 1985. (Cité en pages 39 et 45.)

[David 2001] Alexandre David et M. Oliver Möller. From huppaal to uppaal : A translation from hierarchical timed automata to flat timed automata. Technical Report RS-01-11, BRICS, 2001. (Cité en page 80.)

[David 2003] Alexandre David. Hierarchical modeling and analysis of timed systems. Thèse de Doctorat de l'Uppsala University, 2003. (Cité en page 80.)

[Delfieu 1995] David Delfieu. Expression et validation de contraintes temporelles pour la spécification des systèmes réactifs. Thèse de Doctorat de l'Université Paul Sabatier, 1995. (Cité en pages 39 et 45.)

[Demri 2009] Amel Demri. Contribution à l'évaluation de la fiabilité d'un système mécatronique par modélisation fonctionnelle et dysfonctionnelle. Thèse de Doctorat de l'Université d'Angers, 2009. (Cité en page 41.)

[Dhaussy 2007] Philippe Dhaussy et Frédéric Boniol. *Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués.* Ingénierie des Systèmes d'Information, vol. 12, no. 5, pages 133–157, 2007. (Cité en page 129.)

[Diaz 1994] Michel Diaz, Guy Juanole et Jean-Pierre Courtiat. *Observer-A Concept for Formal On-Line Validation of Distributed Systems.* IEEE Transactions on Software Engineering, vol. 20, pages 900–913, December 1994. (Cité en page 132.)

[Dobre 2010] Dragos Dobre. Contribution à la modélisation d'un système interactif d'aide à la conduite d'un procédé industriel. Thèse de Doctorat de l'Université Henri Poincaré, Nancy 1, 2010. (Cité en page 21.)

[Drissi 2000] Jawad Drissi. Vers la construction automatique d'un module inconnu dans un système composé. Thèse de Doctorat de l'Université de Montréal, 2000. (Cité en page 73.)

[Dwyer 1999] Matthew B. Dwyer, George S. Avrunin et James C. Corbett. *Patterns in Property Specifications for Finite-state Verification.* In Proceedings of the 21st international conference on Software Engineering, pages 411–420, 1999. (Cité en pages 39, 40, 45, 47 et 133.)

[Ehrig 2009] Karsten Ehrig, Jochen Malte Küster et Gabriele Taentzer. *Generating instance models from meta models.* Software and System Modeling, vol. 8, no. 4, pages 479–500, 2009. (Cité en page 71.)

[EIA 1998] Electronic Industries Alliance EIA. ANSI/EIA-632-1999 : Processes for engineering a system. EIA, Arlington, VA, 1998. (Cité en page 21.)

[Emerson 1989] E. Allen Emerson et Jai Srinivasan. *Branching time temporal logic.* In Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, pages 123–172, London, UK, 1989. Springer-Verlag. (Cité en page 42.)

[Eshuis 2000] Rik Eshuis et Roel Wieringa. *Requirements Level Semantics for UML Statecharts.* In S.F. Smith et C.L. Talcott, editeurs, 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), volume 177

of *IFIP Conference Proceedings*, pages 121–140, Norwell, MA, USA, September 2000. Kluwer Academic Publishers. (Cité en pages 88, 90 et 97.)

[Fecher 2005] Harald Fecher, Jens Schönborn, Marcel Kyas et Willem paul De. *W.P. : 29 New unclarities in the semantics of UML 2.0 state machines*. In Proceedings of the 7th international conference on formal engineering methods (ICFEM'05), LNCS, pages 52–65, 2005. (Cité en page 88.)

[Fleurey 2006] Franck Fleurey. Langage et méthode pour une ingénierie des modèles fiable. Thèse de Doctorat de l'Université de Rennes 1, 2006. (Cité en page 70.)

[Fleurey 2009] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller et Yves Le Traon. *Towards Dependable Model Transformations : Qualifying Input Test Data*. Journal of Software and Systems Modeling (SoSyM), vol. 8, no. 2, pages 185–203, 2009. (Cité en page 71.)

[Fontan 2008] Bejamin Fontan. Méthodologie de conception de systèmes temps réel et distribués en contexte uml/sysml. Thèse de Doctorat de l'Université Toulouse Paul Sabatier, 2008. (Cité en pages 39, 45 et 129.)

[Force 1994] United States Air Force. MIL-STD-499b : Systems engineering. US Department of Defense, 1994. (Cité en page 21.)

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Design patterns : Elements of reusable object-oriented software. Addison-Wesley, 1995. (Cité en pages 30, 65 et 129.)

[Gerth 1996] Rob Gerth, Doron Peled, Moshe Y. Vardi et Pierre Wolper. *Simple on-the-fly automatic verification of linear temporal logic*. In Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd. (Cité en page 132.)

[Gervais 2004] Frédéric Gervais. Eb4 : Vers une méthode combinée de spécification formelle des systèmes d'information. Rapport d'examen de spécialité : Université de Sherbrooke & Conservatoire National des Arts et Métiers, 2004. (Cité en page 24.)

[Ghazel 2009a] Mohamed Ghazel. *Using Stochastic Petri Nets for Level-Crossing Collision Risk Assessment*. IEEE Transactions on Intelligent Transportation Systems, vol. 10, no. 4, pages 668–677, 2009. (Cité en page 145.)

[Ghazel 2009b] Mohamed Ghazel, Armand Toguyéni et Pascal Yim. *State Observer for DES Under Partial Observation with Time Petri Nets*. Discrete Event Dynamic Systems, vol. 19, pages 137–165, 2009. (Cité en page 129.)

[Ghazel 2010] Mohamed Ghazel et Ahmed Mekki. *Assisting Specification and Consistency-Check of Temporal Requirements for Critical Systems*. In Proceedings of the Software Engineering Research and Practice (SERP10), Las Vegas, Nevada, USA, 2010. (Cité en page 27.)

[Giese 2006] Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer et Robert Wagner. *Towards Verified Model Transformations*. In Proceedings of MODEVA workshop associated to MODELS'06, pages 78–93, 2006. (Cité en page 71.)

[Girault 2005] Alain Girault et Eric Rutten. *Modeling Fault-tolerant Distributed Systems for Discrete Controller Synthesis*. Electronic Notes in Theoretical Computer Science, vol. 133, pages 81–100, 2005. (Cité en page 129.)

[Godary 2004] Karen Godary. Validation temporelle de réseaux embarqués critiques et fiables pour l'automobile. Thèse de Doctorat de l'Institut National des Sciences Appliquées de Lyon, 2004. (Cité en pages 14, 120, 121 et 129.)

[Gordon 1993] Michael J. C. Gordon et Tom F. Melham. *Introduction to HOL : A theorem proving Environment for higher-order logic*. 1993. (Cité en page 124.)

[Gruhn 2006] Volker Gruhn et Ralf Laue. *Patterns for Timed Property Specifications*. Electronic Notes in Theoretical Computer Science, vol. 153, no. 2, pages 117–133, 2006. (Cité en page 133.)

[Guillerm 2011] Romaric Guillerm. Intégration de la sûreté de fonctionnement dans les processus d'ingénierie système. Thèse de Doctorat de l'Université Paul Sabatier - Toulouse, 2011. (Cité en page 38.)

[Halbwachs 1994] Nicolas Halbwachs, Fabienne Lagnier et Pascal Raymond. *Synchronous Observers and the Verification of Reactive Systems*. In Proceedings of the Third International Conference on Methodology and Software Technology : Algebraic Methodology and Software Technology, pages 83–96. Springer-Verlag, 1994. (Cité en pages 129 et 132.)

[Hall 1990] Anthony Hall. *Seven Myths of Formal Methods*. IEEE Softw., vol. 7, pages 11–19, 1990. (Cité en pages 25 et 44.)

[Harel 1987] David Harel. *Statecharts : A Visual Formalism for Complex Systems*. Science of Computer Programming, vol. 8, no. 3, pages 231–274, Juin 1987. (Cité en page 81.)

[Havelund 2001] Klaus Havelund, Mike Lowry et John Penix. *Formal Analysis of a Space-Craft Controller Using SPIN*. IEEE Transactions on Software Engineering, vol. 27, pages 749–765, 2001. (Cité en page 126.)

[Hedia 2008] Belgacem Ben Hedia. Analyse temporelle des systèmes d'acquisition de données : une approche à base d'automates temporisés communicants et d'observateurs. Thèse de Doctorat de l'Institut National des Sciences appliquées de Lyon (INSA-Lyon), 2008. (Cité en pages 129 et 132.)

[Henzinger 1994] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis et Sergio Yovine. *Symbolic Model Checking for Real-time Systems*. Information and Computation, vol. 111, pages 193–244, 1994. (Cité en page 92.)

[Holzmann 1997] Gerard J. Holzmann. *The Model Checker SPIN*. Transactions on Software Engineering - Special issue on formal methods in software practice, vol. 23, pages 279–295, May 1997. (Cité en page 126.)

[Hooman 2008] Jozef Hooman, Hillel Kugler, Iulian Ober, Anjelika Votintseva et Yuri Yushtein. *Supporting UML-based development of embedded systems by formal techniques*. Software and Systems Modeling, vol. 7, pages 131–155, 2008. (Cité en page 132.)

[Hopcroft 2006] John E. Hopcroft, Rajeev Motwani et Jeffrey D. Ullman. Introduction to automata theory, languages, and computation (3rd edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. (Cité en page 43.)

[Hull 2005] Elizabeth Hull, Kenneth Jackson et Jeremy Dick. *Requirement Engineering (Second Edition)*. Springer, 2005. (Cité en page 38.)

[IEEE 1990] Institute of Electrical and Electronics Engineers IEEE. IEEE 610.12- 1990 : Standard glossary of software engineering terminology. IEEE, New York, NY, 1990. (Cité en page 23.)

[IEEE 2005] Institute of Electrical and Electronics Engineers IEEE. IEEE Std 1220TM- 2005 : IEEE standard for application and management of the systems engineering process. IEEE, New York, NY, 2005. (Cité en pages 13, 21 et 22.)

[ISO 2008] International Organization for Standardization ISO. ISO/IEC 15288-2008 : Systems engineering - system life-cycle processes. ISO, 2008. (Cité en page 21.)

[Jabri 2010a] Sana Jabri. Génération de scénarios de tests pour la vérification de systèmes répartis : application au système européen de signalisation ferroviaire (ertms). Thèse de Doctorat de l'école centrale de Lille, 2010. (Cité en page 139.)

[Jabri 2010b] Sana Jabri, El Miloudi El Koursi, Thomas Bourdeaud'huy et Etienne Lemaire. *European railway traffic management system validation using UML/Petri nets modelling strategy.* European Transport Research Review, vol. 2, no. 2, pages 113–128, 2010. (Cité en page 139.)

[Jard 1988] C. Jard, J. F. Monin et R. Groz. *Development of Veda, a Prototyping Tool for Distributed Algorithms.* IEEE Transactions on Software Engineering, vol. 14, pages 339–352, March 1988. (Cité en page 132.)

[Jones 1990] Cliff-B Jones. Systematic software development using vdm. Prentice Hall, 1990. (Cité en page 44.)

[Jouault 2005] Frédéric Jouault et Ivan Kurtev. *Transforming Models with ATL.* In Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, 2005. (Cité en page 69.)

[Junior 1997] José Celso Freire Junior. Ingénierie des systèmes d'information : Une approche de multi-modélisation et de méta-modélisation. Thèse de Doctorat de l'Université Joseph Fourrier - Grenoble 1, 1997. (Cité en page 43.)

[Katoen 2008] Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008. (Cité en pages 14, 121 et 122.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina-Videira Lopes, Jean-Marc Loingtier et John Irwin. *Aspect-oriented programming.* In Mehmet Akşit et Satoshi Matsuoka, editeurs, ECOOP'97 Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, chapitre 10, pages 220–242. Springer Verlag, 1997. (Cité en page 65.)

[Kissoum 2009] Yacine Kissoum, Zaidi Sahnoun et Kamel Barkaoui. *An Approach for Testing Mobile Agents Using the Nets within Nets Paradigm.* In RCIS'09, 3rd IEEE International Conference on Research Challenges in Information Science, Fez, Morocco, April 22-24, IEEE, pages 207–216, April 2009. (Cité en page 122.)

[Kissoum 2010] Yacine Kissoum, Zaidi Sahnoun et Kamel Barkaoui. *Model-based testing approach for mobile agents using the paradigm of reference net.* Multiagent and Grid Systems, vol. 6, pages 271–292, December 2010. (Cité en page 122.)

[Konaté 2009] Jacqueline Konaté. Approche système pour la conception d'une méthodologie pour l'elicitation collaborative des exigences. Thèse de Doctorat de l'Université de Toulouse, 2009. (Cité en pages 13, 23 et 24.)

[Konrad 2005] Sascha Konrad et Betty H. C. Cheng. *Real-time Specification Patterns.* In Proceedings of the 27th International Conference on Software Engineering (ICSE05), 2005. (Cité en pages 39, 40, 45, 47 et 133.)

[Kotonya 1998] Gerald Kotonya et Ian Sommerville. Requirements engineering - processes and techniques. John Wiley & Sons, 1998. (Cité en page 24.)

[Kurtev 2008] Ivan Kurtev. *State of the Art of QVT : A Model Transformation Language Standard.* Data Engineering, vol. 5088, no. ii, pages 377–393, 2008. (Cité en page 68.)

[Küster 2003] Jochen Malte Küster, Reiko Heckel et Gregor Engels. *Defining and validating transformations of UML models.* In IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), pages 145–152. IEEE Computer Society, 2003. (Cité en page 71.)

[Küster 2004] Jochen Küster. *Systematic Validation of Model Transformations*. In Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004), Lisbon (Portugal), 2004. (Cité en page 71.)

[Küster 2006a] Jochen Küster. *Definition and validation of model transformations*. Software and Systems Modeling, vol. 5, pages 233–259, 2006. (Cité en page 71.)

[Küster 2006b] Jochen Küster et Mohamed AbdelRazik. *Validation of Model Transformations – First Experiences using a White Box Approach*. In In Proceedings Of MoDeVa'06 (3rd International Workshop on Model Development, Validation and Verification), associated to MODELS'06 (9th International Conference on Model Driven Engineering Languages And Systems), pages 62–77, 2006. (Cité en page 70.)

[Lanotte 2000] Ruggero Lanotte, Andrea Maggiolo-Schettini et Adriano Peron. *Timed cooperating automata*. Fundamenta Informaticae - Special issue on Concurrency specification and programming, vol. 43, pages 153–173, 2000. (Cité en page 80.)

[Lanotte 2001] Ruggero Lanotte, Andrea Maggiolo-Schettini, Simone Tini et Adriano Peron. *Transformations of Timed Cooperating Automata*. Fundamenta Informaticae - Concurrency Specification and Programming, vol. 47, pages 271–282, October 2001. (Cité en page 80.)

[Lanotte 2006] Ruggero Lanotte, Andrea Maggioloschettini, Paolo Milazzo et Angelo Troina. *Modeling Longrunning Transactions with Communicating Hierarchical Timed Automata*. In Proc. of Formal Methods for Open Object-Based Distributed Systems (FMOODS'06), LNCS 4037. Springer, 2006. (Cité en page 80.)

[Lanotte 2008] Ruggero Lanotte, Andrea Maggiolo-Schettini, Paolo Milazzo et Angelo Troina. *Design and verification of long-running transactions in a timed framework*. Science of Computer Programming, vol. 73, pages 76–94, 2008. (Cité en page 80.)

[Larsen 1997] Kim G. Larsen, Paul Pettersson et Wang Yi. UPPAAL *in a Nutshell*. Int. Journal on Software Tools for Technology Transfer, vol. 1, no. 1–2, pages 134–152, Octobre 1997. (Cité en pages 92, 126 et 128.)

[Lepors 2010] Eric Lepors. Interprétation sémantique des exigences pour l'enrichissement de la traçabilité et pour l'amélioration des architectures de systèmes complexes. Thèse de Doctorat de l'Université de Bretagne Sud, 2010. (Cité en page 23.)

[Manna 1992] Zohar Manna et Amir Pnueli. The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc., New York, NY, USA, 1992. (Cité en page 39.)

[Marangé 2009] Pascale Marangé, David Gouyon, Jean-François Pétin et Bernard Riera. *Verification of functional constraints for safe product driven control*. In 2nd IFAC Workshop on Dependable Control of Discrete System, DCDS'09, pages 315–320, 2009. (Cité en page 38.)

[Marangé 2010] Pascale Marangé, David Gouyon, Jean-François Pétin et François Gellot. *Formalisation et vérification de contraintes fonctionnelles dans le cadre d'un contrôle par le produit*. In Sixième Conférence Internationale Francophone d'Automatique, CIFA 2010, 2010. (Cité en page 38.)

[McCulloch 1943] Warren McCulloch et Walter Pitts. *A logical calculus of the ideas immanent in nervous activity*. Bulletin of Mathematical Biology, vol. 5, no. 4, pages 115–133, Décembre 1943. (Cité en page 43.)

[McMillan 1993] Kenneth L. McMillan. Symbolic model checking. Kluwer Academic Publishers, 1993. (Cité en page 126.)

[Medjoudj 2006] Malika Medjoudj. Contribution à l'analyse des systèmes pilotés par calculateurs : Extraction de scénarios redoutés et vérification de contraintes temporelles. Thèse de Doctorat de l'Université Paul Sabatier de Toulouse, 2006. (Cité en page 126.)

[Mekki 2009a] Ahmed Mekki. *Évaluation des exigences temporelles des systèmes complexes. Aperçu de l'existant et introduction d'une nouvelle approche générique.* In Journée des doctorants (JDD), Actes INRETS, ISBN 978 – 2 – 85782 – 677 – 4, ISSN 0769 – 0266, IFSTTAR (ex INRETS), pages 133–148, 2009. (Cité en page 29.)

[Mekki 2009b] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Validating time-constrained systems using UML Statecharts Patterns and Timed Automata Observers.* In Proceedings of the 3rd International Workshop on Verification and Evaluation of Computer and Communication Systems, Rabat, Morroco, 2009. (Cité en pages 27 et 29.)

[Mekki 2010a] Ahmed Mekki. *Patterns de spécification des exigences temporelles.* In Journée des doctorants (JDD), Actes INRETS, ISBN 978 – 2 – 85782 – 690 – 3, ISSN 0769 – 0266, IFSTTAR (ex INRETS), 2010. (Cité en page 129.)

[Mekki 2010b] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Patterns for Temporal Requirements Engineering ; A level crossing case study.* In Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics,Funchal, Madeira, Portugal, 2010. (Cité en page 27.)

[Mekki 2010c] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Time-constrained systems validation using MDA model transformation. A railway case study.* In Proceedings of the 8th ENIM IFAC International Conference of Modeling and Simulation MOSIM2010, Hammamet, Tunisia, 2010. (Cité en pages 28 et 79.)

[Mekki 2011a] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Patterns-Based Assistance for Temporal Requirement Specification.* In Proceedings of the Software Engineering Research and Practice (SERP11), Las Vegas, Nevada, USA, 2011. (Cité en page 27.)

[Mekki 2011b] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Timed Specification Patterns for System Validation : A Railway Case Study.* In Juan Andrade Cetto, Jean-Louis Ferrier et Joaquim Filipe, editeurs, Informatics in Control, Automation and Robotics, volume 89 of *Lecture Notes in Electrical Engineering*, pages 121–134. Springer Berlin Heidelberg, 2011. (Cité en page 29.)

[Mekki 2012a] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Assisting Temporal Requirement Specification.* Computer Technology and Application, vol. 3, pages 47–55, 2012. (Cité en page 29.)

[Mekki 2012b] Ahmed Mekki, Mohamed Ghazel et Armand Toguyéni. *Validation of a New Functional Design of Automatic Protection Systems at Level-Crossings with Model-Checking Techniques.* IEEE Transactions on Intelligent Transportation Systems, vol. 13, no. 2, pages 714–723, 2012. (Cité en page 145.)

[Merz 2006] Stephan Merz. *Model checking : Éléments de base.* Systèmes temps réel : techniques de description et de vérification, pages 89–120, 2006. (Cité en pages 43 et 125.)

[Meyer 2001] Eric Meyer. Développement formels par objets : utilisation conjointe de b et uml. Thèse de Doctorat de l'Université Nancy 2, 2001. (Cité en page 44.)

[Mikk 1997] Erich Mikk, Yassine Lakhnech et Michael Siegel. *Hierarchical Automata as Model for Statecharts.* In Proceedings of the Third Asian Computing Science Conference

on Advances in Computing Science, ASIAN '97, pages 181–196, London, UK, 1997. Springer-Verlag. (Cité en page 80.)

[Minsky 1965] Marvin Minsky. *Matter, Mind and Models*. In Proceedings of IFIP Congress 65, pages 45–49, 1965. (Cité en page 65.)

[Missaoui 2010] Abdallah Missaoui, Zohra Sbai et Kamel Barkaoui. *Model Checking Verification of Web Services Composition*. In ACT4SOC'10, 4th International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, Athens, Greece, 23 July, July 2010. (Cité en page 125.)

[Moigne 1999] Jean-Louis Le Moigne. La modélisation des systèmes complexes. Dunod, 1999. (Cité en page 22.)

[Morin 1994] Edgar Morin. Introduction à la pensée complexe. Communication et complexité. ESF, 1994. (Cité en page 22.)

[Moser 1997] L.E. Moser, Y. S. Ramakrishna, G. Kutty, P.M. MELLIAR-SMITH et L. K. Dillon. *A Graphical Environment for Design of Concurrent Real-Time Systems*. ACM Transactions on Software Engineering and Methodology, vol. 6, pages 31–79, 1997. (Cité en page 40.)

[Mottu 2010] Jean-Marie Mottu, Benoit Baudry et Yves Le TrRaon. *Construction de tests qualifiés de transformations de modèles*. Technique Et Science Informatiques, vol. 29, pages 537–569, 2010. (Cité en page 71.)

[Murata 1989] Tadao Murata. *Petri nets : Properties, analysis and applications*. Proceedings of the IEEE, vol. 77, no. 4, pages 541–580, 1989. (Cité en page 42.)

[Ober 2006] Iulian Ober, Susanne Graf et Ileana Ober. *Validating timed UML models by simulation and verification*. International Journal on Software Tools for Technology Transfer (STTT), Special Section on Specification and Validation of Models of Real Time and Embedded Systems with UM, vol. 8, pages 128–145, 2006. (Cité en page 132.)

[Olender 1990] K.M. Olender et L.J. Osterweil. *Cecil : a sequencing constraint language for automatic static analysis generation*. Software Engineering, IEEE Transactions on, vol. 16, no. 3, pages 268 –280, mar 1990. (Cité en page 39.)

[OMG 2004] OMG. *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*, 2004. (Cité en pages 65 et 67.)

[OMG 2005] OMG. *MOF QVT Final Adopted Specification*, 2005. OMG doc. ptc/05-11-01. (Cité en pages 68 et 69.)

[OMG 2011a] OMG. *Unified Modeling Language Specification 2.4 : Infrastructure*, 2011. OMG doc. ptc/2010-11-16. (Cité en page 66.)

[OMG 2011b] OMG. *Unified Modeling Language Specification 2.4 : Superstructure*, 2011. OMG doc. ptc/2010-11-14. (Cité en pages 13, 66, 81 et 96.)

[OMG 2011c] OMG. *Semantics of a Foundational Subset for Executable UML Models (FUML)*. Février 2011. (Cité en page 41.)

[Ott 2009] François Ott. Complexité de l'accompagnement en formations professionnelles des conceptions en tensions : sujet(s), projet(s), organisations(s). Thèse de Doctorat de l'Université Lille 1, 2009. (Cité en page 22.)

[Owre 1996] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar et Mandayam K. Srivas. *PVS : combining specification, proof checking and model checking*. pages 411–414, 1996. (Cité en page 124.)

[Paulson 1994] Lawrence C. Paulson. *Isabelle : A Generic Theorem Prover*. vol. 828, 1994. (Cité en page 124.)

[Peres 2011] Florent Peres, Bernard Berthomieu et François Vernadat. *On the composition of time Petri nets*. Discrete Event Dynamic Systems, vol. 21, no. 3, pages 395–424, 2011. (Cité en page 71.)

[Petri 1962] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn : Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition :, New York : Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages : Suppl. 1, English translation. (Cité en page 42.)

[Pnueli 1977] Amir Pnueli. *The Temporal Logic of Programs*. In 18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA, pages 46–57, 1977. (Cité en pages 41 et 42.)

[Pnueli 1981] Amir Pnueli. *The Temporal Semantics of Concurrent Programs*. Theoretical Computer Science, vol. 13, pages 45–60, 1981. (Cité en page 42.)

[Poernomo 2008] Iman Poernomo. *Proofs-as-Model-Transformations*. In Theory and Practice of Model Transformations, First International Conference, ICMT 2008, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008. (Cité en page 71.)

[Ramakrishna 1996] Y.S. Ramakrishna, P.M. Melliar-Smith, Louise E. Moser, Laura K. Dillon et G. Kutty. *Interval Logics and Their Decision Procedures - Part I + II : A Real-Time Interval Logic*. Theoretical Computer Science, vol. 170, no. 1-2, pages 1–46, 1996. (Cité en page 39.)

[RFF 2006] RFF. *Partout en Europe, un même langage ferroviaire*. Le journal de la ligne, vol. 4, pages 8–9, 2006. (Cité en pages 140, 142 et 144.)

[Roger 2006] Jean Charles Roger. Exploitation de contextes et d'observateurs pour la validation formelle de modèles. Thèse de Doctorat de l'Université Rennes 1, 2006. (Cité en pages 129, 132 et 133.)

[Roussel 2002a] Jean-Marc Roussel et Bruno Denis. *Safety properties verification of ladder diagram programs*. Journal Européen des Systèmes Automatisés, vol. 36, no. 7, pages 905–917, 2002. (Cité en page 124.)

[Roussel 2002b] Jean-Marc Roussel et Jean-Marc Faure. *An algebraic approach for PLC programs verification*. In Proceedings of 6th International Workshop on Discrete Event Systems (WODES'02) 6th International Workshop on Discrete Event Systems (WODES'02), pages 303–308, 2002. (Cité en page 124.)

[Rumbaugh 2004] James Rumbaugh, Ivar Jacobson et Grady Booch. Uml 2.0 guide de référence. CampusPress, 2004. (Cité en page 81.)

[Rushby 2000] John Rushby. *Theorem proving for verification*, 2000. (Cité en page 124.)

[Rut] (Cité en page 45.)

[Rutten 2009] Eric Rutten. *Supervisory control of adaptive and reconfigurable computing systems*. In Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing, INCOM'09, June, 2009, 2009. (Cité en page 45.)

[Sadani 2007] Tarek Sadani. Vers l'utilisation des réseaux de petri temporels étendus pour la vérification de systèmes temps-réel décrits en rt-lotos. Thèse de Doctorat de l'Institut National Polytechnique de Toulouse, 2007. (Cité en pages 39, 40 et 45.)

[Salmon 1992] Salmon. Backus-naur forms. Ed. Lavoisier, July 1992. (Cité en page 48.)

[Schnoebelen 1999] Ph. Schnoebelen. Vérification de logiciels : Techniques et outils du model checking. Vuibert, 1999. (Cité en page 38.)

[Schön 2008] Walter Schön. *Sécurité des systèmes critiques et cybercriminalité : vers une sécurité globale ?* Cahiers de la Sécurité, vol. La criminalité numérique, no. 6, pages 146–154, 2008. (Cité en pages 22 et 23.)

[Seidewitz 2003] Ed Seidewitz. *What Models Mean.* IEEE Software, vol. 20, pages 26–32, 2003. (Cité en page 65.)

[Seidewitz 2011] Ed Seidewitz. *TUTORIAL : Programming in UML, An Introduction to fUML and Alf.* In Executable UML Information Day, March 22, 2011, 2011. (Cité en page 79.)

[Seidner 2009] Charlotte Seidner. Vérification des effbds : Model checking en ingénierie système. Thèse de Doctorat de l'Université de Nantes, 2009. (Cité en pages 13, 21, 22, 37, 71, 72 et 73.)

[Sen 2007] Sagar Sen, Benoit Baudry et Doina Precup. *Partial Model Completion in Model Driven Engineering using Constraint Logic Programming.* In International Conference on the Applications of Declarative Programming, 2007. (Cité en page 71.)

[Sen 2008] Sagar Sen, Benoit Baudry et Jean-Marie Mottu. *On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing.* In International Conference on Software Testing, Verification, and Validation, ICST'08., Lillehammer, Norway, 2008. (Cité en page 71.)

[Slovak 2007] Roman Slovak et Stefan Wegele. *Level crossing accidents statistics, normalisation and comparison.* In Second SELCAT Workshop, Morroco, pages 83–100, 2007. (Cité en page 147.)

[Sommerville 1997] Ian Sommerville et Pete Sawyer. Requirements Engineering : A Good Practice Guide. John Wiley & Sons, New York, NY, USA, 1997. (Cité en page 21.)

[Spivey 1992] J.Michael Spivey. The z notation : a reference manual. Prentice- Hall, 1992. (Cité en pages 43 et 44.)

[Staff 2008] Topcased Staff. *TOPCASED-WP5, Guide méthodologique pour les transformations de modèles.* Rapport de recherche n 8, Novembre 2008, 2008. (Cité en pages 13 et 69.)

[Steel 2004] Jim Steel et Michael Lawley. *Model-Based Test Driven Development of the Tefkat Model-Transformation Engine.* In Proceedings of the 15th International Symposium on Software Reliability Engineering, pages 151–160, Washington, DC, USA, 2004. IEEE Computer Society. (Cité en page 70.)

[Taylor 2008] Kevin Taylor. *Addressing road user behavioural changes at railway level crossings.* In ACRS-Travelsafe National Conference, Brisbane, pages 368–375, 2008. (Cité en page 147.)

[Toussaint 1997] Joel Toussaint, Françoise Simonot-Lion et Jean-Pierre Thomesse. *Time constraints verification method based on time Petri nets.* In 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, pages 262–267, 1997. (Cité en page 133.)

[Vardi 1986] Moshe Y. Vardi et Pierre Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* In Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86), pages 332–344. IEEE Comp. Soc. Press, Juin 1986. (Cité en page 132.)

[Varró 2003] Dániel Varró et András Pataricza. *Automated Formal Verification of Model Transformations*. In Jan Jürjens, Bernhard Rumpe, Robert France et Eduardo B. Fernandez, editeurs, CSDUML 2003 : Critical Systems Development in UML ; Proceedings of the UML'03 Workshop, numéro TUM-I0323 de Technical Report, page 63–78. Technische Universität München, Technische Universität München, September 2003. (Cité en page 71.)

[Volker 2002] Norbert Volker et Bernd J. Kramer. *Automated verification of function block-based industrial control systems*. Science of Computer Programming, vol. 42, no. 1, pages 101–113, 2002. (Cité en page 124.)

[Waez 2011] Md Tawhid Bin Waez, Juergen Dingel et Karen Rudie. Timed automata for the development of real-time systems. Technical Report 2011-579, 2011. (Cité en page 80.)

[Wahl 1994] Thomas Wahl et Kurt Rothermel. *Representing Time in Multimedia Systems*. In Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS'94), pages 538–543, Boston, USA, 1994. (Cité en page 39.)

[Yovine 1997] Sergio Yovine. *KRONOS : A Verification Tool for Real-Time Systems*. Springer International Journal of Software Tools for Technology Transfer, vol. 1, no. 1–2, pages 123–133, 1997. (Cité en page 126.)

Contribution à la Spécification et à la Vérification des Exigences Temporelles. Proposition d'une extension des SRS d'ERTMS niveau 2

Résumé : De par la complexité croissante des systèmes industriels de nos jours et notamment dans le secteur du transport ferroviaire, l'ingénierie des exigences est devenue un enjeu majeur dans le cycle de développement. Les travaux développés dans le cadre de cette thèse visent à assister le processus d'ingénierie des exigences temporelles pour les systèmes complexes à contraintes de temps. Nos contributions portent sur trois volets, à savoir : la phase de spécification des exigences, la phase de modélisation du comportement et la phase de vérification.

Pour le volet spécification, une nouvelle classification des exigences temporelles les plus communément utilisées a été proposée. Par la suite, afin d'assister et cadrer l'utilisateur lors de la phase d'expression des exigences, nous avons développé une grammaire de spécification à base de motifs pré-définis en langage naturel. Chaque assertion générée à partir de cette grammaire identifie une classe unique dans la classification des exigences (et vice-versa). Les exigences générées sont syntaxiquement précises et correctes quand elles sont prises individuellement, néanmoins cela ne garantit pas la cohérence de l'ensemble des exigences exprimées. Ainsi, afin de remédier à cette problématique de cohérence, nous avons développé des mécanismes capables de détecter certains types d'incohérences entre les exigences temporelles. Ces mécanismes sont basés sur une reformulation des exigences sous forme d'un graphe orienté. Pour le deuxième volet, la modélisation du comportement, nous avons proposé un algorithme de transformation des state-machines (SM) avec des annotations temporelles en des automates temporisés (AT). L'idée étant de donner la possibilité à l'utilisateur de manipuler une notation assez intuitive (diagramme SM d'UML) lors de la phase de modélisation et d'en générer automatiquement des modèles formels (AT) qui se prêtent à la vérification. Les règles de transformation sont définies au niveau des méta-modèles respectifs des SM et des AT, que nous avons proposés. Par ailleurs, une sémantique formelle a été définie pour chacun de ces modèles. Enfin, notre algorithme de transformation a été validé par la démonstration d'une relation de bisimulation temporelle forte entre le modèle source et le modèle cible. Finalement, pour la phase de vérification, nous avons adopté une technique de vérification à base d'observateurs et qui repose sur le *model-checking*. Concrètement, en se basant sur la classification des exigences temporelles que nous avons développée, nous avons élaboré une base de patterns d'observation ; chacun des patterns développés est relatif à un type d'exigence temporelle. Concrètement, pour chaque classe d'exigence dans notre classification, le pattern observateur associé correspond à un modèle automate temporisé paramétrable dont les instances jouent le rôle de "chien de garde" (ou *watchdog* en anglais) pour surveiller les exigences de cette classe. Ainsi, la vérification est réduite à une analyse d'accessibilité des états d'erreur, états correspondant à la violation de l'exigence associée. Il est à noter par ailleurs que des prototypes logiciels ont été développés pour supporter les différents mécanismes proposés.

La dernière partie de nos travaux concerne une proposition d'extension des spécifications du système de contrôle-commande et de signalisation ferroviaire, ERTMS, afin d'y intégrer les passages à niveau. Ce cas d'étude nous a servi à l'illustration des différentes contributions proposées dans cette thèse.

Mots clés : Exigences temporelles, spécification, transformation de modèles, vérification et validation, observateur, model-checking, contrôle/commande ferroviaire, ERTMS.

Contribution for the Specification and the Verification of Temporal Requirements.
Proposal of an extension for the ERTMS-Level 2 specifications

Abstract : Due to the increasing complexity of today's industrial systems in the railway domain, requirements' engineering has become a major issue in the development cycle of such systems. The work developed in this thesis aims at assisting the engineering process of temporal requirements for time-constrained complex systems. Our contributions concern three phases : the specification, the behavioural modelling and the verification.

For the specification of temporal requirements, a new temporal property typology taking into account all the common requirements one may meet when dealing with requirements specification, is introduced. Then, to facilitate the expression and the identification, we have proposed a structured English grammar (SEG). Concretely, each assertion generated through our grammar describes one temporal property and serves as a handler that aids expressing and understanding the requirement. Nevertheless, even if each requirement taken individually is correct, we have no guarantee that a set of temporal properties one may express is consistent. Here we have proposed an algorithm based on graph theory techniques to check the consistency of temporal requirements sets. For the behaviour modelling, we have proposed an algorithm for transforming UML State Machine with time annotations into Timed Automata (TA). The idea is to allow the user manipulating a quite intuitive notation (UML SM diagrams) during the modelling phase and thereby, automatically generate formal models (TA) that could be used directly by the verification process. The transformation rules have been defined at the SM and TA meta-models level. Furthemore, a formal semantics has been defined for each of these models (SM and TA). Finally, our transformation algorithm was validated while prooving a strong temporal bisimulation relationship between the source model and the target model. Finally, for the verification phase, we have adopted an observer-based technique. Actually, we have developped a repository of observation patterns where each pattern is relative to a particular temporal requirement class in our classification. In practice, in order to check the temporal requirements of a given model, the observation patterns relative to the investigated properties are instantiated to make appropriate TA observers. Then using our transformation algorithm, the system model (denoted in the shape of a UML SM model with time annotations) is translated into TA models. The TA system model is next synchronized with the TA observers. Thereby, the verification process is reduced to a reachability analysis of the observers' KO states relative to the requirements' violation. The whole of the developed mechanisms have been implemented in prototype software tools.

The last part of our work is a proposal for the integration of level crossings (LC) into the specifications of the European railway control and signalling system, ERTMS. This case study has been used to illustrate the various contributions developed in this thesis.

Keywords : Temporal Requirements, System Specification, Model Transformation, Verification & Validation, Observation Patterns, Model-checking, Railway control systems, ERTMS.