Université Abdelhamid Ibn Badis Mostaganem

Faculté des Sciences Exactes et Informatique

Département d'informatique

Thèse de Doctorat

Présentée par :

Djamila BAROUDI

Sémantique formelle des observateurs pour la validation des transformations

Spécialité : Informatique

Option : Apprentissage Automatique et Web Intelligence

Soutenue le 27/06/2022 devant un jury composé de :

Président du jury	**Pr. Karim SEHABA**	(Université de Mostaganem)
Examinateur	**Dr. Bouchra KAID SLIMANE**	(Université de Mostaganem)
Examinateur	**Dr. Moulay Driss MECHAOUI**	(Université de Mostaganem)
Examinateur	**Pr. Khaled BELKADI**	(Université USTO Oran)
Directeur de thèse	**Pr. Safia NAIT-BAHLOUL**	(Université d'Oran1)

Résumé

Dwyer et al. ont proposé des patrons de spécification qualitatives qui permettent aux praticiens des outils de model-checking d'écrire des spécifications formelles principalement utilisées pour le model-checking automatique. Bien que cela implique des formalismes qui ne sont pas toujours faciles à manipuler par les ingénieurs, plusieurs techniques et langages ont été proposés pour faciliter l'intégration des méthodes formelles basées sur ces patrons de définition dans le domaine industriel. Cette thèse étudie un langage de description spécifique nommé CDL qui aide les non-experts à écrire des spécifications formelles d'une manière plus facile. Dans CDL, une propriété est transformée en un automate observateur pour effectuer une analyse d'accessibilité. Les patrons CDL existants permettent aux non-experts de raisonner sur l'occurrence et l'ordre des événements, mais pas suffisamment sur leur timing. De plus, la sémantique des patrons et des transformations n'est pas idéalement formalisée et reste complexe à comprendre et à utiliser. Enfin, l'exactitude des transformations des patrons aux observateurs proposées n'a pas été prouvée. Ce travail a pour but d'étendre le système CDL existant avec des patrons temporels ainsi que de formaliser la sémantique de ces patrons. En outre, de nouvelles règles de transformation sont définies et leur exactitude est prouvée à l'aide d'un outil de démonstration de théorèmes appelé Coq.

La contribution est illustrée sur un système industriel embarqué impliquant des navires autonomes. Les navires autonomes doivent adhérer aux règles du trafic maritime pour assurer la sécurité du trafic et réduire la responsabilité des fabricants. Cependant, les systèmes autonomes ne peuvent évaluer le respect des règles que si elles sont formulées de manière précise et mathématique. Dans cette thèse, nous formalisons les règles de trafic maritime de la Convention sur le Règlement International pour Prévenir les Collisions en Mer (COLREGS) en utilisant nos patrons de propriétés et les automates observateurs. En particulier, nous définissons les règles de prévention des collisions entre deux navires motorisés.

Mots-clés : Patrons de propriétés, Spécification des propriétés temporelles, Automates observateurs, Preuve de théorème, Validation des transformations, Navires autonomes

مُلَخَّصٌ

اِقْتَرَحَ أَنْمَاط لِلْمُوَاصَفَاتِ اَلنَّوْعِيَّةِ اَلَّتِي تَمَكَّنَ مُسْتَخْدِمِي أَدَوَاتِ فَحْصِ اَلنَّمَاذِجِ مِنْ كِتَابَةِ اَلْمُوَاصَفَاتِ اَلرَّسْمِيَّةِ اَلْمُسْتَخْدَمَةِ بِشَكْلٍ رَئِيسِيٍّ فِي فَحْصِ اَلنَّمَاذِجِ اَلْأُوتُومَاتِيكِيَّةِ . عَلَى اَلرَّغْمِ مِنْ أَنَّ هَذَا يَتَضَمَّنُ شَكْلِيَّاتٍ لَيْسَ مِنْ اَلسَّهْلِ دَائِمًا اَلتَّعَامُلُ مَعَهَا مِنْ قِبَلِ اَلْمُهَنْدِسِينَ ، فَقَدْ تَمَّ اِقْتِرَاحُ اَلْعَدِيدِ مِنْ اَلتِّقْنِيَّاتِ وَاللُّغَاتِ لِتَسْهِيلِ دَمْجِ اَلْأَسَالِيبِ اَلرَّسْمِيَّةِ بِنَاءً عَلَى أَنْمَاطِ اَلتَّعْرِيفِ اَلْمَذْكُورَةِ أَعْلَاهُ فِي اَلْمَجَالِ اَلصِّنَاعِيِّ . نُنَاقِشُ فِي هَذِهِ اَلْمُذَكِّرَةِ لُغَةً خَاصَّةً تُسَمَّى CDL تُسَاعِدَ غَيْرَ اَلْخُبَرَاءِ فِي كِتَابَةِ اَلْمُوَاصَفَاتِ اَلرَّسْمِيَّةِ دُونَ عَنَاءٍ . فِي CDL ، يَتِمَّ تَحْوِيلُ اَلْخَاصِّيَّةِ إِلَى مُرَاقِبٍ إِلَى (أُوتُومَاتْ) لِإِجْرَاءِ تَحْلِيلِ قَابِلِيَّةِ اَلْوُصُولِ . تَسْمَحَ أَنْمَاطَ CDL اَلْحَالِيَّةَ لِغَيْرِ اَلْخُبَرَاءِ اَلتَّعْبِيرُ عَنْ حُدُوثِ اَلْوَقَائِعِ وَتَرْتِيبِهَا ، وَلَكِنْ لَا تَأْخُذُ بِعَيْنِ اَلِاعْتِبَارِ تَوْقِيتَهَا . عِلَاوَةٌ عَلَى ذَلِكَ ، فَإِنَّ دَلَالَاتِ اَلْأَنْمَاطِ وَالتَّحَوُّلَاتِ لَيْسَتْ رَسْمِيَّةً بِشَكْلٍ مِثَالِيٍّ وَتَظَلَّ مُعَقَّدَةً فِي فَهْمِهَا وَالتَّعَامُلُ مَعَهَا . مِنْ جِهَةٍ أُخْرَى ، لَمْ يَتِمْ إِثْبَاتُ صِحَّةِ قَوَاعِدِ اَلتَّحْوِيلِ اَلْمُقْتَرَحَةِ اَلَّتِي تَسْمَحُ بِتَحْوِيلِ اَلْأَنْمَاطِ إِلَى آلَاتٍ نِهَائِيَّةٍ مُرَاقَبَةٍ . يَهْدِفُ هَذَا اَلْعَمَلِ إِلَى تَوْسِيعِ نِظَامِ CDL اَلْحَالِيَّ بِأَنْمَاطِ مُرْتَبِطَةٍ بِالْوَقْتِ بِالْإِضَافَةِ إِلَى إِضْفَاءِ اَلطَّابَعِ اَلرَّسْمِيِّ عَلَى دَلَالَاتِ هَذِهِ اَلْأَنْمَاطِ . بِالْإِضَافَةِ إِلَى ذَلِكَ ، يَتِمَّ تَحْدِيدُ قَوَاعِدِ اَلتَّحْوِيلَاتِ اَلْجَدِيدَةِ وَإِثْبَاتِ صِحَّتِهَا بِاسْتِخْدَامِ أَدَاةِ إِثْبَاتِ نَظَرِيَّةٍ تُسَمَّى Coq . يَتِمَّ تَوْضِيحُ اَلْمُسَاهَمَاتِ اَلْمُقْتَرَحَةِ فِي هَذَا اَلْعَمَلِ عَلَى نِظَامٍ صِنَاعِيٍّ يَتَضَمَّنُ سُفُن مُسْتَقِلَّةً ذاتية القيادة. يَتَعَيَّنَ عَلَى اَلسُّفُنِ اَلْمُسْتَقِلَّةِ اَلِالْتِزَامَ بِقَوَاعِدِ اَلْمُرُورِ اَلْبَحْرِيِّ لِضَمَانِ سَلَامَةِ اَلْمُرُورِ وَتَقْلِيلِ مَسْؤُولِيَّةِ اَلشَّرِكَاتِ اَلْمُصَنِّعَةِ . وَمَعَ ذَلِكَ ، لَا يُمْكِنُ لِلْأَنْظِمَةِ اَلْمُسْتَقِلَّةِ تَقْيِيمَ اَلِامْتِثَالِ لِلَّوَائِحِ إِلَّا إِذَا تَمَّتْ صِيَاغَتُهَا بِطَرِيقَةٍ دَقِيقَةٍ وَرِيَاضِيَّةٍ . فِي هَذِهِ اَلْأُطْرُوحَةِ ، نَقُومُ بِإِضْفَاءِ اَلطَّابَعِ اَلرَّسْمِيِّ عَلَى قَوَاعِدِ اَلْمُرُورِ اَلْبَحْرِيِّ لِاتِّفَاقِيَّةِ اَللَّوَائِحِ اَلدَّوْلِيَّةِ لِمَنْعِ اَلتَّصَادُمِ فِي اَلْبَحْرِ (COLREGS) بِاسْتِخْدَامِ أَنْمَاطِ اَلْمِلْكِيَّةِ اَلْخَاصَّةِ بِنَا وَآلِيَّاتُ اَلْمُرَاقَبَةِ . عَلَى وَجْهِ اَلْخُصُوصِ ، نُحَدِّدُ قَوَاعِدُ مَنْعِ اَلِاصْطِدَامَاتِ بَيْنَ سَفِينَتَيْنِ بِمُحَرِّكٍ .

الكلمات المفتاحيَّة: أنماط المواصفات، الات نهائيةٍ مراقبةٍ، مواصفات الخصائص الموقوتة، ،التحقق من صحة التحويلات ،السفن ذاتية القيادة مساعد إثبات .Coq

Abstract

Dwyer et al. proposed qualitative specification patterns that enable the practitioners of model-checking tools to write formal specifications mainly used for automatic model-checking. Although this involves formalisms that are not always easy to handle by engineers, several techniques and languages have been proposed to facilitate the integration of formal methods based on these definition patterns in the industrial field. This thesis studies a domain-specific language named CDL that helps non-experts effortlessly write formal specifications. In CDL, a property is transformed into an observer automaton to perform a reachability analysis. The existing CDL patterns allow non-experts to reason about the occurrence and order of events, but not enough about their timing. Furthermore, the semantics of patterns and transformations is not ideally formalized and remain complex. Moreover, the correctness of the proposed transformations from patterns to observers has not been proven. This work aims to extend the existing CDL system with time-related patterns as well as formalize the semantics of these patterns. In addition, new transformations rules are defined and their correctness is proved using a theorem proving tool called Coq.

The contribution is illustrated on an embedded industrial system involving autonomous vessels. Autonomous vessels have to adhere to marine traffic rules to ensure traffic safety and reduce the liability of manufacturers. However, autonomous systems can only assess compliance with regulations if they are formulated in a precise and mathematical manner. In this thesis, we formalize the maritime traffic rules of the Convention on the International Regulations for Preventing Collisions at Sea (COLREGS) using our property patterns and observer automata. In particular, we define the rules for preventing collisions between two motorized vessels.

Keywords : Property patterns, Observers automata, Timed properties specification, Coq proof assistant, Transformation validation, Autonomous vessels.

Remerciements

Je mesure la chance qui m'a été donnée d'être encadrée par une professeure aussi impliquée, ouverte et compétente, et sans qui ce travail n'aurait jamais vu le jour. Qu'elle trouve ici le témoignage de toute ma gratitude :

Safia NAIT-BAHLOUL, qui sait faire voler en éclat chaque idée reçue, qui s'est dévouée généreusement à ma thèse et qui m'a donnée goût à la recherche. Elle m'a fait énormément apprendre tant sur le plan scientifique qu'humain.

Un grand merci au professeur Karim SEHABA de l'université de Mostaganem pour avoir accepté d'être président du jury, au professeur Bouchra Kaid SLIMANE et professeur Moulay Driss MECHAOUI de l'université de Mostaganem, et au professeur Khaled BELKADI de l'université Mohamed Boudiaf d'Oran pour l'honneur qu'ils me font en acceptant d'être les rapporteurs de cette thèse. Je suis très honorée de l'intérêt qu'ils ont porté à mes travaux.

Mes remerciements vont également :

À mes ami(e)s et collègues sans exception, pour leur présence, leur respect et leur gentillesse.

Je ne saurais finir sans exprimer mes remerciements aux personnes qui me sont les plus proches : mes parents, qui m'ont tout donnée pour apprendre et pour réussir ; mes sœurs Zahira et Samiha et mes frères Mohamed et Abdelhamid, qui étaient toujours présents pour m'aider et me réconforter. Je remercie également mes beaux parents et mes belles soeurs Hadjer et Houaria et mon beau frère Khalil. Enfin, et surtout, un grand merci Ismail, mon mari qui a été à mes côtés, qui a cru en moi, et qui a toujours su me redonner le sourire dans les moments difficiles.

À tous ceux qui me sont chers :
Mes parents,
Mon mari,
Mes frères et sœurs,
Ma belle famille.

Table des matières

Liste des figures xix

Liste des tableaux xxii

Partie I Introduction Générale 1

Chapitre 1 Introduction et contexte 3

1.1 Introduction Générale 4

1.2 Contexte et problématique 5

1.2.1 Recherche de fiabilité des systèmes embarqués 5

1.2.2 L'approche OBP/CDL 6

1.3 Limites de l'approche OBP/CDL 6

1.3.1 Difficultés liées à l'expression des propriétés temporelles 7

1.3.2 Difficultés liées à la composition des automates observateurs . . . 7

1.4 Contributions 8

1.5 Organisation de thèse 9

Partie II État de l'art 13

Chapitre 2 État de l'art 15

2.1 Introduction 16

2.2 Intégration des méthodes formelles aux industries 16

2.3 Patrons de spécification de propriétés 17

2.3.1 Grammaires structurées 18

2.3.2 Automates observateurs 19
2.3.3 Transformations de patrons en automates 20
2.3.4 Approches compositionnelles 20
2.4 Vérification des systèmes temporels 21
2.4.1 Analyse statique 21
2.4.2 Preuves de théorèmes (Theorem proving) 22
2.4.3 Model-checking 23
2.4.3.1 Comment spécifier les propriétés ? 24
2.4.4 Approche hybride 24
2.5 Conclusion 26

Chapitre 3 Patrons de propriétés **29**
3.1 Introduction 30
3.2 Les patrons de propriétés 30
3.2.1 Le système de patrons de Dwyer 30
3.2.2 Approche OBP/CDL 32
3.2.2.1 Les patrons de propriétés CDL 33
3.2.3 Les propriétés temporelles 35
3.2.4 Besoin d'une grammaire structurée 36
3.3 Les observateurs 36
3.3.1 Bases théoriques 37
3.3.1.1 Systèmes de transitions 37
3.3.1.2 Automates temporisés 38
3.3.1.3 Exemple 39
3.3.1.4 Compositions et communications 40
3.3.2 Les automates observateurs 41
3.3.2.1 Propriétés 43
3.4 Conclusion 44

Partie III Contributions **47**

Chapitre 4 Extension CDL **49**
4.1 Introduction 50

4.2 Système de patrons de propriétés ECDL 50
4.2.1 Expression des patrons de propriétés ECDL 51
4.3 Patrons de propriétés ECDL 52
4.3.1 Patrons de propriétés qualitatives 53
4.3.1.1 Patron de réponse 53
4.3.2 Patron de précédence 54
4.3.3 Patron d'absence 55
4.3.4 Patron d'existence 55
4.3.5 Patrons de propriétés temporelles 56
4.3.6 Options ECDL 56
4.3.7 Définition des scopes 58
4.4 Grammaire structurée 59
4.5 Intérêts des patrons de propriétés ECDL 62
4.6 Conclusion 63

Chapitre 5 Patrons comme observateurs **65**
5.1 Introduction 66
5.2 Observateurs 66
5.3 Patrons observateurs 67
5.3.1 Non-accessibilité 67
5.3.2 Patron de précédence 68
5.3.2.1 Précédence AN 68
5.3.2.2 Précédence AllOrdered version acyclique 70
5.3.2.3 Précédence AllCombined version acyclique 71
5.3.3 Patron de réponse 72
5.3.3.1 Réponse AN : 72
5.3.3.2 Réponse AllOrdered version acyclique 73
5.3.3.3 Réponse AllCombined version acyclique 74
5.3.4 Existence bornée 75
5.3.5 Précédence bornée 76
5.3.5.1 Précédence bornée AN 76
5.3.5.2 Précédence bornée AllOrdered version acyclique 78
5.3.5.3 Précédence bornée AllCombined version acyclique . . . 78
5.3.6 Réponse bornée 79

5.3.6.1 Réponse AN 79
5.3.6.2 Réponse AllOrdered version acyclique 81
5.3.6.3 Réponse AllCombined version acyclique 82
5.4 Conclusion 83

Chapitre 6 Transformation des patrons 85
6.1 Introduction 86
6.2 Formalisation des patrons ECDL sans options et scopes 86
6.2.1 Détermination des états 87
6.2.2 Génération des transitions 89
6.2.3 Règles de combinaison de clauses *Pre* et *Post* 89
6.3 Le framework ECDL 92
6.4 Approche compositionnelle 94
6.4.1 Exemple de motivation 94
6.4.2 Formalisation des patrons ECDL 95
6.5 Composition de patrons et options 98
6.5.1 Règles de composition de patrons et options 98
6.5.1.1 Spécification des options comme observateurs 98
6.5.2 L'opération de composition 99
6.5.2.1 Les règles de substitutions 100
6.5.2.2 Les règles de composition 101
6.6 Composition des patrons et scopes 103
6.7 Conclusion 104

Chapitre 7 Validations des transformations 107
7.1 Introduction 108
7.2 Vérification formelle - "Preuve Bisimulation" 108
7.3 Description de Coq 109
7.4 Sémantique opérationnelle des automates observateurs 110
7.4.1 Automates 111
7.4.1.1 Sémantique d'occurrence des événements 111
7.4.2 Implémentation en Coq 114
7.5 Sémantique opérationnelle des patrons ECDL 116
7.5.1 Type de patron 117

7.5.2 Type d'occurrence des évènements . . . 119
7.5.3 Nombre d'occurrence des évènements . . . 120
7.5.4 Implémentation en Coq . . . 120
7.6 Construction de patrons . . . 121
7.7 Règles de traduction . . . 121
7.7.1 Implémentation en Coq . . . 122
7.7.2 Théorème à prouver . . . 123
7.8 Conclusion . . . 127

Partie IV Expérimentations, conclusion et perspectives 129

Chapitre 8 Exemple industriel 131
8.1 Introduction . . . 132
8.2 Vérification formelle des véhicules autonomes . . . 132
8.2.1 Concept . . . 132
8.2.2 Vérification des bateaux autonomes . . . 133
8.3 Les règles de COLREG . . . 134
8.3.1 Généralités . . . 134
8.3.2 Catégories des règles COLREG . . . 134
8.4 Prévention de collision . . . 135
8.4.1 Machine à états . . . 136
8.4.2 Expérimentations sur la machine d'état . . . 141
8.4.3 Patrons ECDL pour les règles de COLREG . . . 142
8.4.3.1 Propriétés d'accessibilité . . . 142
8.4.3.2 Propriété de croisement bâbord . . . 144
8.4.3.3 Propriété de croisement tribord . . . 145
8.4.3.4 Propriété de rencontre frontale "*Head-On*" . . . 145
8.4.3.5 Propriété de rencontre avec un dépassement "*Overtaking*" 146
8.5 Discussions . . . 148

Chapitre 9 Conclusion et perspectives 149
9.1 Conclusion . . . 150
9.1.1 ECDL et grammaire structurelle . . . 151

9.1.2 Observateurs de patrons de propriétés 151
9.1.3 Validation des transformations 152
9.1.4 Exemple d'illustration 152
9.2 Travaux futurs 153
9.2.1 Enrichissement du langage ECDL 153
9.2.2 Extension de l'application des patrons ECDL pour les navires autonomes 153

Annexes **155**

Partie V Annexes **155**

Annexe A Grammaire structurelle **157**
A.1 Grammaire de patrons de propriétés 157
A.1.1 Grammaire ECDL 157
A.1.2 Règle Pattern 158
A.1.3 Règles Arity 158
A.1.4 Règles de Temps 160
A.1.5 Options 160
A.1.6 Tokens 162

Annexe B Patrons observateurs **163**
B.1 Observateurs de patron de précédence 163
B.1.1 Précédence AllOrdered version cyclique 163
B.1.2 Précédence AllOrdered version strictement cyclique 163
B.1.2.1 Précédence AllCombined version cyclique 164
B.1.2.2 Précédence AllCombined version strictement cyclique . 165
B.2 Observateurs de patron de réponse 166
B.2.1 Réponse AllOrdered version cyclique 166
B.2.2 Réponse AllOrdered version strictement cyclique 166
B.2.3 Réponse AllCombined version cyclique 167
B.2.4 Réponse AllCombined version strictement cyclique 167
B.3 Observateurs de patron de Précédence Bornée 168
B.3.1 Précédence Bornée version cyclique 168

B.3.2 Précédence bornée AllOrdered version strictement cyclique 169
B.3.3 Précédence bornée AllCombined version cyclique 169
B.3.4 Précédence bornée AllCombined version strictement cyclique . . . 170
B.4 Observateurs de patron de réponse bornée 171
B.4.1 Réponse bornée AllOrdered version cyclique 171
B.4.2 Réponse AllOrdered version strictement cyclique 171
B.4.3 Réponse bornée AllCombined version cyclique 172
B.4.4 Réponse AllCombined version strictement cyclique 172

Abréviations **175**

Bibliographie **177**

Liste des figures

1.1 Organisation de thèse 11

2.1 Patron de réponse 18
2.2 Principe de vérification par observateur 19
2.3 Principe de model-checking 24
2.4 Combinaison des méthodes formelles 25

3.1 Système de patrons de propriétés de Dwyer et al. 31
3.2 Aperçu sur OBP [127] 32
3.3 Patron de propriété réponse CDL et options 34
3.4 Automate temporisé exemple [60] 40
3.5 Automate observateur 43

4.1 Schéma du système ECDL 51
4.2 Système de patrons ECDL 52
4.3 Classification des patrons ECDL 53

5.1 Automate Observateur pour le patron de “Précédence” AN 70
5.2 Automates observateurs pour le patron “Précédence” AllOrdered et AllCombined 72
5.3 Automate observateur pour le patron de “Réponse” AN 74
5.4 Automates Observateurs pour le patron “Réponse” AllOrdered et AllCombined 75
5.5 Automate observateur pour le patron “Existence Bornée” 75
5.6 Automates observateurs pour le patron “Précédence Bornée AN” 77
5.7 Automates observateurs pour le patron “Précédence bornée” AllOrdered et AllCombined 79

5.8 Automate observateur pour le patron de “Réponse Bornée” AN 81
5.9 Automates Observateurs pour le patron “Réponse” AllOrdered et AllCombined 82

6.1 Éléments d’un patron de réponse ECDL (exemple P1) 86
6.2 Observateur de patron de réponse bornée 91
6.3 Observateur sans options 92
6.4 L’outil de transformation ECDL 93
6.5 Automate correspondant à P1 généré par l’outil ECDL 94
6.6 Patron de **réponse** ECDL et l’observateur correspondant 95
6.7 Patron de réponse correspondant à **E1** 97
6.8 Options sous forme d’automate 99
6.9 Application des règles de composition sur l’exemple de Figure 6.7 102
6.10 Scopes automates [126] 103
6.11 Composition de patron de réponse ECDL avec scope "*After Q*" 104

7.1 Identification d’un type de patron en fonction de certains opérateurs 117
7.2 LTS correspondants aux patrons ECDL 118
7.3 Relation de simulation 124

8.1 Illustration des différentes situations d’abordage de deux navires et les actions à prendre 137
8.2 Interprétation des règles ECDLs de prévention de collision par une machine à état 138
8.3 Interprétation géométrique de COLREG 140
8.4 Statistiques sur les types de rencontre des navires 141
8.5 Observateur de Propriété de non-accessibilité de COLREG 143
8.6 Observateur de règle de croisement bâbord 144
8.7 Observateur de règle de croisement tribord 145
8.8 Observateur de croisement frontale (HeadOn) 146
8.9 Observateur de croisement avec dépassement (Overtaking) 147

A.1 Grammaire ECDL 158
A.2 Pattern 158
A.3 Arity 158

A.4 Le corps Arity 159
A.5 Type Arity 159
A.6 Occurence Arity 159
A.7 Expression Arity 159
A.8 Expression de Temps 160
A.9 Durée 160
A.10 Unité de temps 160
A.11 Type Options 161
A.12 Option Repeatability 161
A.13 Option Nullity 161
A.14 Occurence Probability 161

B.1 Automates observateurs pour le patron “Précédence” AllOrdered versions cycliques 164
B.2 Automates observateurs pour le patron “Précédence” AllCombined versions cycliques 165
B.3 Automates Observateurs pour le patron “Réponse” AllOrdered versions cycliques 167
B.4 Automates observateurs pour le patron “Réponse” AllOrdered versions cycliques 168
B.5 Automates observateurs pour le patron “Précédence Bornée” AllOrdered versions cycliques 169
B.6 Automates observateurs pour le patron “Précédence Bornée” AllCombined versions cycliques 170
B.7 Automates observateurs pour le patron “Réponse” AllOrdered versions cycliques 172
B.8 Automates observateurs pour le patron “Réponse bornée” AllCombined versions cycliques 173

Liste des tableaux

3.1 Tableaux récapitulatif de travaux sur les observateurs 42

4.1 Formulation de patron de réponse éventuelle 54
4.2 Formulation de patron de réponse immédiate 54
4.3 Formulation de patron de précédence 55
4.4 Formulation de patron d'absence 55
4.5 Formulation de patron d'existence 56
4.6 Aperçu du catalogue de patrons de spécification en temps réel 57
4.7 Grammaire structurée des patrons/options temporels 60
4.8 Grammaire structurée des patrons qualitatifs 61
4.9 Comparaison de CDL classique et ECDL 62

5.1 Patron de Non-Accessibilité 68
5.2 Patron de Précédence AN 69
5.3 Précédence AN version cyclique 69
5.4 Précédence AN version strictement cyclique 70
5.5 Patron de précédence AllOrdered version acyclique 71
5.6 Patron de précédence AllCombined version acyclique 71
5.7 Patron de Réponse AN version Acyclique 72
5.8 Patron de réponse AN version cyclique 73
5.9 Patron de réponse AN version strictement cyclique 73
5.10 Patron de réponse AllOrdered version Acyclique 74
5.11 Patron de réponse AllCombined version Acyclique 74
5.12 Patron existence bornée version cyclique 75

5.13 Patron de précédence bornée AN version acyclique 76
5.14 Patron de précédence bornée AN version cyclique 77
5.15 Patron de précédence bornée AN version strictement cyclique 77
5.16 Patron de précédence bornée AllOrdered version acyclique 78
5.17 Patron de précédence bornée AllCombined version acyclique 79
5.18 Patron de Réponse bornée AN version Acyclique 80
5.19 Patron de Réponse Bornée AN version cyclique 80
5.20 Patron de Réponse bornée AN version strictement cyclique 81
5.21 Patron de Réponse bornée AllOrdered version Acyclique 81
5.22 Patron de Réponse Bornée AllCombined version Acyclique 82

6.1 Exigence E1 96

8.1 Catégories des règles COLREG selon leurs champs d'application 135
8.2 Patron de Non-Accessibilité 143
8.3 Patron de croisement bâbord- *Règle 15* 144
8.4 Patron de croisement Tribord - *Règle 15* 145
8.5 Patron de rencontre frontale (*HeadOn*) - *Règle 14* 146
8.6 Patron de dépassement (*Overtacking*) - *Règle 13* 147

B.1 Patron de Précédence AllOrdered version Cyclique 163
B.2 Patron de Précédence AllOrdered version strictement Cyclique 164
B.3 Patron de précédence AllCombined version cyclique 165
B.4 Patron de précédence AllCombined version strictement cyclique 165
B.5 Patron de réponse AllOrdered version cyclique 166
B.6 Patron de résponse AllOrdered version strictement cyclique 166
B.7 Patron de réponse AllCombined version cyclique 167
B.8 Patron de réponse AllCombined version strictement cyclique 167
B.9 Patron de précédence bornée AllOrdered version cyclique 168
B.10 Patron de précédence bornée AllOrdered version strictement cyclique 169
B.11 Patron de Précédence AllCombined version cyclique 170
B.12 Patron de Précédence AllCombined version cyclique 170
B.13 Patron de Réponse bornée AllOrdered version cyclique 171

B.14 Patron de réponse bornée AllOrdered version strictement cyclique 171
B.15 Patron de réponse bornée AllCombined version cyclique 172
B.16 Patron de réponse bornée AllCombined version strictement cyclique 172

Première partie

Introduction Générale

Chapitre

1

Introduction et contexte

« Et au-dessus de tout homme détenant la science, il y a un savant plus docte que lui. »

— Joseph, v. 76

Sommaire

1.1 Introduction Générale **4**

1.2 Contexte et problématique **5**

1.2.1 Recherche de fiabilité des systèmes embarqués 5

1.2.2 L'approche OBP/CDL 6

1.3 Limites de l'approche OBP/CDL **6**

1.3.1 Difficultés liées à l'expression des propriétés temporelles 7

1.3.2 Difficultés liées à la composition des automates observateurs . . 7

1.4 Contributions **8**

1.5 Organisation de thèse **9**

1.1 Introduction Générale

Compte tenu de la nature omniprésente des systèmes embarqués [61] et de leur utilisation pour des applications critiques (par exemple, les dispositifs médicaux, les systèmes de transport, ...), ils doivent généralement atteindre un niveau élevé de robustesse et de fiabilité. De plus, les logiciels des systèmes embarqués impliquent généralement des fonctionnalités dépendantes du temps. Par conséquent, les méthodes de développement et de modélisation des systèmes embarqués et la vérification rigoureuse de leur comportement avant leur mise en action sont de plus en plus importantes.

Pour garantir le bon fonctionnement de ces systèmes, des techniques de spécification et de vérification formelles sont utilisées. Ces techniques sont basées sur des logiques temporelles et permettent d'établir la conformité d'un système et de ses propriétés. Cependant, pour que ce processus fonctionne, il faut d'abord maîtriser les logiques temporelles, ce qui est une tâche difficile et longue [25]. Souvent, les ingénieurs sont peu familiers avec les formalismes existants et considèrent qu'il est trop difficile d'utiliser ces logiques temporelles pour spécifier les propriétés d'un système. D'autre part, les outils de model-checking qui peuvent être utilisés pour vérifier ces propriétés nécessitent souvent des spécifications écrites en formules de logiques temporelles. Les patrons de spécification des propriétés [62] ont réussi à combler ce fossé entre les praticiens et les outils de vérification des modèles (model-checking). Un patron exprime une propriété temporelle sur une exécution, vue comme des séquences d'état/événement. Chaque patron peut être associé à une portée (scope) qui définit les parties du système d'exécution sur lesquels le patron doit s'appliquer. Pourtant, de nombreux auteurs ont révélé à quel point il est pénible pour des non experts d'exprimer des propriétés en utilisant des patrons de définition de Dwyer et al. [63].

Pour résoudre ce problème, Dhaussy et al.[51] ont défini un langage de description de contexte nommé CDL qui regroupe les principaux concepts proposés par [93, 62] [121] et permet l'expression de propriétés (sécurité, invariance ou vivacité) plus facilement. De plus, ils implémentent un outil appelé OBP (Observer-Based Prover) qui transforme les propriétés en automates observateurs non intrusifs avec des états de rejet. Le processus de vérification des propriétés consiste en une analyse d'accessibilité sur le graphe d'exploration issu de la composition du modèle testé et de l'observateur. Si l'un des états de rejet est atteint, cela signifie que la propriété vérifiée n'est pas valide. Le système de patrons CDL existant ne prend pas suffisamment en compte les informations relatives au temps et aux scopes.

Le travail présenté dans cette thèse montre que le langage CDL manque de sémantique formelle des patrons et des transformations ainsi qu'il ne couvrent pas des propriétés temporels et propose une nouvelle extension permettant d'exprimer d'autres propriétés temporelles en ajoutant de nouvelles variantes de patrons (scopes et options). En plus, le nouveau langage présenté dans ce travail définit une grammaire structurée facilitant l'expression et la formalisation des patrons proposés. La validation des transformations des patrons en automates

observateurs est prouvée à l'aide de l'assistant de preuve Coq. La contribution est illustrée sur un système industriel embarqué impliquant des navires autonomes. Dans cette thèse, nous formalisons les règles de trafic maritime de la Convention sur le Règlement International pour Prévenir les Collisions en Mer (COLREGS) en utilisant nos patrons de propriétés et les automates observateurs. En particulier, nous définissons les règles de prévention des collisions entre deux navires motorisés.

1.2 Contexte et problématique

La vérification des propriétés de sécurité aux premiers stades du développement d'un système est l'une des exigences les plus critiques qui permet d'obtenir des systèmes plus fiable. La conception de systèmes sécurisés est un paradigme émergent dans le développement de logiciels/matériels sécurisés qui fournit des outils et des procédures aux équipes de production de systèmes pour vérifier leurs spécifications de conception par rapport aux propriétés de sécurité requises. Dans cette approche, la spécification du système est modélisée avec un langage de spécification formel, puis un outil de vérification formelle qui vérifie si le modèle répond à une spécification donnée avant que l'équipe ne commence à mettre en œuvre un modèle vulnérable du système.

1.2.1 Recherche de fiabilité des systèmes embarqués

La fiabilité est définie comme la probabilité qu'un système fonctionne correctement pendant une période de temps spécifique en présence de défaillances probables. Différentes analyses de fiabilité et techniques d'amélioration ont été proposées dans la littérature [8, 101]. Pour répondre aux exigences de fiabilité toujours croissantes, de nouvelles méthodes de spécification, de conception et de développement de logiciels dans les systèmes embarqués ont évolué. Toutefois, face à la complexité croissante des systèmes embarqués et à leur dépendance souvent à des logiciels à fonctionnalités critiques, les développeurs de ces systèmes seront finalement contraints d'adopter des méthodes plus robustes pour leur vérification. Ces techniques de vérification connues sous le nom de méthodes formelles constituent une alternative viable aux tests traditionnels [118, 70, 72].

Au cours des dernières années, de nombreux efforts ont été déployés pour réduire l'écart entre les méthodes formelles en théorie et leurs applications industrielles. Ces méthodes sont passées d'une élégante théorie à une véritable application pratique. Cependant, tout passage de la théorie à la pratique est confronté à de nombreux défis. Les nouvelles techniques doivent rivaliser avec des pratiques bien établies et démontrer la nécessité de remplacer l'ancien par le nouveau, tout en surmontant les préoccupations et les doutes fréquents des ingénieurs. Le but principal des méthodes formelles est de garantir le comportement d'un système informatique en utilisant des approches d'analyse ou de construction rigoureuses. Le comportement est

décrit par une spécification, de préférence de nature mathématique.

En dépit de l'efficacité progressive de ces méthodes, leur intégration dans un processus d'ingénierie industrielle reste difficile due en partie à la complexité de la mise en œuvre des concepts théoriques dans un contexte industriel. De plus, beaucoup d'entre eux impliquent des formalismes qui peuvent être extrêmement complexes pour être adoptés par les ingénieurs de l'industrie dans leurs activités de vérification.

1.2.2 L'approche OBP/CDL

Il existe deux approches prédominantes en méthodes formelles qui visent à assurer une fiabilité mathématique : la vérification de modèle (model-checking) et la preuve de théorème (theorem proving). La première méthode utilise généralement des scénarios de test, combinés à des algorithmes astucieux pour atteindre cet objectif, tandis que la seconde méthode fournit un contexte permettant de raisonner mathématiquement (preuve) sur les spécifications et les implémentations. Les deux méthodes partagent le fait qu'elles nécessitent une description formelle à la fois de l'implémention, et de la spécification. Le model-checking nécessite que les exigences à vérifier soient exprimées, sous la forme de propriétés formelles.

Partant des constats précédents, des travaux [51, 50, 117] se sont penchés sur cette problématique en cherchant à faciliter l'utilisation des outils de model-checking. Ils ont proposé un DSL (Domain Specific Language) nommé CDL (Context Description Language) permettant de formuler les propriétés à l'aide de patrons de définition [63, 62]. Un patron est une structure syntaxique textuelle qui permet un mode d'expression d'une propriété plus proche des langages que les ingénieurs ont l'habitude de manipuler. Après leurs expressions, les propriétés sont transformées en automates observateurs. Ces observateurs sont ensuite simulés et explorés à l'aide d'un explorateur de modèle couplé à un analyseur d'accessibilité nommé OBP [1]. Cette contribution visait à faciliter l'expression des exigences dans un format compréhensible pour un non-expert des logiques temporelles ainsi de pouvoir mener des vérifications par model-checking sur des modèles de grande taille. Nous décrivons plus en détail cette approche dans Chapitre 3.

1.3 Limites de l'approche OBP/CDL

Suite aux diverses expérimentations menées avec l'approche OBP/CDL, nous établissons un constat et identifions les difficultés d'expression de certaines propriétés. Par ailleurs, nous proposons une extension des travaux précédents qui contribue à l'amélioration de cette approche. Deux difficultés majeures ont été relevées. La première consiste en l'absence des expressions permettant une description complète et cohérente des patrons de propriétés temporelles. Tan-

1. http://www.obpcdl.org/

dis que la deuxième concerne la formalisation des patrons en composant leurs automates observateurs pour exprimer des propriétés plus complexe.

1.3.1 Difficultés liées à l'expression des propriétés temporelles

Dans les systèmes temps réel, l'activation (le démarrage) des tâches est généralement réalisée par des signaux de déclenchement, c'est-à-dire des signaux de commande qui spécifient l'instant où une activité doit commencer dans un intervalle de temps (connu sous le nom de *scopes* dans les patrons de propriétés).

Deux principaux paradigmes sont couramment utilisés pour le déclenchement des activités d'un système temps réel, à savoir l'action déclenchée par un événement et l'action déclenchée par le temps. On dit qu'une action est déclenchée par un événement lorsque le signal de déclenchement est associé à l'occurrence d'un événement significatif, tel que la réception d'un message particulier, la réalisation d'une activité ou l'occurrence d'une interruption externe. Le contrôle déclenché par le temps s'appuie sur la progression du temps et dépend généralement du dépassement périodique d'une horloge.

Le langage CDL est considéré par les industriels comme un langage de plus bas niveau. En revanche, un patron CDL se réfère à des opérations de communication, exprimées sous forme de messages d'entrée et de sortie, dotés parfois de nombreux paramètres. Construire un patron CDL consiste alors à identifier tous les acteurs qui interagissent avec le système dans un contexte particulier, puis tous les événements échangés ainsi que leurs séquences. Les patrons CDL existants [50] permettent de raisonner sur l'occurrence et l'ordre des événements, mais pas suffisamment sur le temps. Par ailleurs, la notion de **scopes** en CDL est négligée.

1.3.2 Difficultés liées à la composition des automates observateurs

Pour mettre en œuvre la vérification de propriétés, l'outillage OBP (Observer-Based Prover) transforme les propriétés en automates observateurs non intrusifs dotés d'états de rejet. Un observateur est construit de manière à encoder une propriété logique [5] et a pour rôle d'observer, partiellement et de manière non intrusive, les évènements significatifs survenant dans le modèle du système à valider. Lors de la simulation, il est composé, de manière synchrone, au modèle à observer. La vérification de propriété consiste alors en une analyse d'accessibilité des états de rejet sur le graphe d'exploration qui résulte de la composition du modèle à valider et des observateurs.

Mais ce mode d'expression des patrons CDL pose deux problèmes majeurs.

— Dans le cas où nous voudrions apporter une variante à la sémantique d'un patron et de ces options, nous devons modifier l'automate de base et la fonction inductive associés à chaque combinaison.

— De plus, les patrons définis sont génériques et informels ce qui rend la transformation des ces patrons en formules génériques équivalentes de logique temporelle pour exprimer des propriétés plus complexe très difficile. L'extensibilité et la généricité sont les principales limites d'une telle sémantique translationnelle.

1.4 Contributions

Dans ce travail, nous désirons concevoir un langage de spécification (1) pour faciliter l'expression des propriétés temporelles en se basant sur les patrons prédéfinis en CDL [51, 50] ainsi que les patrons et les scopes de Dwyer et al.[63, 62]. Ce langage doit être facilement extensible (2) en ajoutant de nouvelles variantes de patrons et de scopes grâce à une sémantique compositionnelle. Enfin, nous souhaitons adopter une sémantique formelle basée sur les automates observateurs (3) qui est bien adaptée à la vérification des propriétés avec des model-checker comme Tina [28] ou OBP Explorer [52], et à la génération et l'évaluation des tests ainsi qu'à la validation des transformations des patrons de propriétés. Pour atteindre cet objectif, nous allons adresser chacune des difficultés listées dans la section précédente.

1. **Faciliter l'expression des patrons CDL** : La logique mathématique, bien qu'elle représente le moyen le plus efficace pour capturer systématiquement les propriétés d'un système, est, en général, trop stricte et la plupart des ingénieurs logiciels préfèrent le langage naturel pour formuler les exigences de leurs systèmes. D'un autre côté, un langage purement naturel peut être trop faible et sujet aux erreurs, car il n'a pas la rigueur nécessaire pour capturer efficacement les propriétés du système. Une grammaire structurée, en tant que interface exprimant les patrons de spécification des propriétés, permet de remédier à cette lacune. Cette grammaire aide à combler le fossé conceptuel entre les deux extrêmes : la spécification en langage naturel et le raisonnement mathématique pur. Nous alignons les propositions existantes de patrons de spécification de propriétés CDL et ceux de [63] dans le but de définir un catalogue unique, mais complet et cohérent, de patrons de spécification de propriétés en utilisant la procédure suivante :
   - *Revue de la littérature* : nous effectuons une analyse approfondie des approches et des patrons existants dans la littérature afin d'obtenir l'état actuel de l'art en matière de spécification des propriétés.
   - *Analyse des lacunes et élicitation des patrons CDL* : sur la base des solutions de modélisation connues pour les propriétés temporelles des systèmes, nous définissons une grammaire de spécification des propriétés qui aide à faciliter l'expression des propriétés temporelles.
2. **Étendre CDL** : Ajouter de nouvelles variantes de patrons et de scopes. Pour atteindre cet objectif nous proposons, d'une part, de clarifier et de mettre en évidence certains formalismes définis dans le langage CDL existant. D'autre part, nous suggèrons d'ajouter

des propriétés temporelles ainsi que la notion de scopes. Ce travail se base sur l'étude de [74] et considère les éléments suivants :

— Les patrons : l'objectif est de pouvoir spécifier davantage de propriétés temps réel, par exemple, "*Bounded Recurrence*" (indique le temps pendant lequel une formule d'état doit être vérifiée au moins une fois).

— Les options : l'intention est d'avoir des options plus précises avec des contraintes temporelles.

— Les scopes : désambiguïser la notion de scopes et faciliter leur utilisation en CDL pour délimiter la période de validité d'un patron.

3. **Adapter une approche compositionnelle** : Dans le langage CDL seule une sémantique informelle est définie. La sémantique des propriétés est définie en traduisant chaque combinaison d'un patron ainsi que les options en automates observateurs suivant des algorithmes de transformations qui n'ont pas été validés. Cette sémantique de transformation souffre de deux problèmes principaux. Elle n'est pas facilement extensible à d'autres patrons et options ou scopes, et elle n'est pas toujours fidèle à la sémantique naturelle (les automates résultants peuvent manquer des transitions ou avoir d'autres en plus). Nous proposons donc une approche basée sur des automates compositionnels définissant la sémantique de chaque patron et de chaque option/scope par un automate, après nous procédons à la composition des automates. Ainsi, la sémantique est compositionnelle et le langage est facilement extensible. Cette sémantique est étendue pour permettre la compositions de différents patrons afin d'exprimer des propriétés plus complexes.
4. **Validation de la sémantique de transformations** : afin de valider notre sémantique, nous fournissons des preuves constructives et vérifiées par un outil de preuve de théorème (Theorem-prover).

1.5 Organisation de thèse

Ce manuscrit de thése est organisé en cinq parties principales comme le montre la figure Figure 1.1. La première partie fixe le contexte du travail et présente une introduction générale sur la problématique étudiée ainsi que les contributions dans Chapitre 1. La deuxième partie résume les différentes approches formelles de la littérature, permettant de modéliser et de vérifier des patrons de propriétés ainsi de la composition des automates observateurs, elle dresse également un bilan des différentes insuffisances constatées en Chapitre 2. Chapitre 3 de cette même partie est consacré à la présentation des patrons de propriétés qui constituent la base de notre étude.

La troisième partie est dédiée à la présentation de nos contributions. Elle est subdivisée en quatre chapitres, dont Chapitre 4 présente ECDL qui est une extension de langage CDL que nous avons proposé afin d'améliorer et faciliter son expression. Dans ce même chapitre,

nous présentons ainsi la grammaire structurée proposée pour notre nouveau langage. Dans Chapitre 5 nous proposons pour chaque patron une syntaxe aussi lisible que possible par un être humain, afin que des ingénieurs non experts en méthodes formelles puissent les utiliser. De plus, nous montrons comment les traduire en propriétés d'accessibilité pures en utilisant des observateurs simples, c'est-à-dire des sous-systèmes supplémentaires qui observent certaines actions du système et qui peuvent également utiliser le temps. Chapitre 6 est consacré à la présentation de notre méthodologie de transformation des patrons en observateurs. Nous proposons aussi une approche compositionnelle qui sert à l'extension des patrons afin de pouvoir représenter des propriétés plus complexes. Dans le dernier chapitre de cette partie (Chapitre 7), nous proposons la validation des transformations en présentant un apperçu sur la preuve formelle avec l'assistant de preuve Coq. Dans la quatrième partie, nous décrivons les expérimentations menées sur le cas d'étude dans Chapitre 8. Puis, Chapitre 9 présente la conclusion générale de ce mémoire et dresse un bilan de nos contributions. Il présente également un aperçu sur les perspectives qui se dégagent de nos travaux. Dans la cinquième et dernière partie, nous regroupons l'ensemble des annexes.

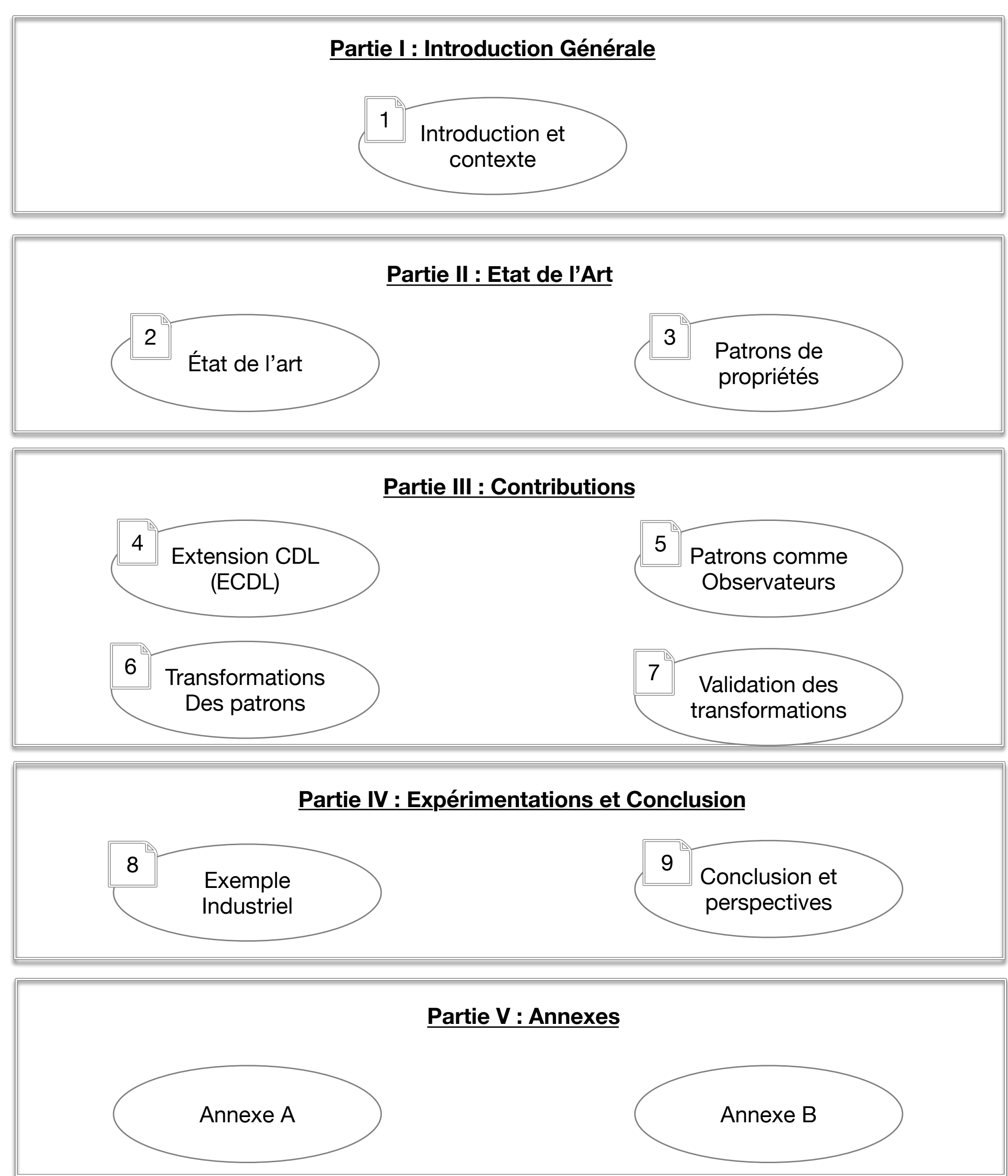


FIGURE 1.1 – Organisation de thèse

Deuxième partie

État de l'art

Chapitre

2

État de l'art

« The beginning of knowledge is the discovery of something we do not understand. »

— Frank Herbert

Sommaire

2.1 Introduction **16**
2.2 Intégration des méthodes formelles aux industries **16**
2.3 Patrons de spécification de propriétés **17**
2.3.1 Grammaires structurées 18
2.3.2 Automates observateurs 19
2.3.3 Transformations de patrons en automates 20
2.3.4 Approches compositionnelles 20
2.4 Vérification des systèmes temporels **21**
2.4.1 Analyse statique 21
2.4.2 Preuves de théorèmes (Theorem proving) 22
2.4.3 Model-checking 23
2.4.4 Approche hybride 24
2.5 Conclusion **26**

2.1 Introduction

Dans ce chapitre, nous présentons l'état de l'art relatif aux différents concepts qui nous ont permis de réaliser nos objectifs. Ces concepts sont liés en général à l'intégration des méthodes formelles dans le processus industriel. Particulièrement, nous nous intéressons d'une part aux approches liées à la spécification des propriétés d'un système embarqué en utilisant la notion de patrons de définition, ainsi qu'à la transformation de ces patrons en automates. D'autre part, nous nous intéressons aux approches de définition des grammaires facilitant la compréhension et l'utilisation des patrons de définition, ainsi qu'aux méthodes de vérification et theorem-proving.

2.2 Intégration des méthodes formelles aux industries

Les méthodes formelles et l'industrie ne sont pas si souvent associées dans la même phrase étant donné qu'elles ne sont pas considérées comme une technologie valorisante, mais plutôt comme difficiles à appliquer et liées à des coûts accrus. Les applications basées sur des méthodes formelles font encore exception à la règle générale en industrie. Le manque de compréhension, les employés insuffisamment formés, la difficulté d'intégrer les cycles de développement existants, l'absence d'exigence explicite du marché, sont des explications souvent entendues pour justifier l'absence de méthodes formelles. Par conséquent, le retour d'informations fourni par l'industrie aux chercheurs n'est pas nécessairement aussi constructif que souhaité.

Plusieurs exemples illustrent que l'application des méthodes formelles sur des cas pertinents sur le plan industriel devient possible. Dans le secteur de l'aviation, l'utilisation de méthodes formelles a été introduite dans les normes de développement et acceptée comme partie de la procédure de certification (obligatoire) [56, 57]. Des outils tels que Astrée [44, 107] et Frama-C [91] ont été utilisés avec succès pour analyser formellement des parties du code de plusieurs modèles d'avions, y compris le plus grand avion de transport de passagers actuellement, l'A380 [109, 122, 80]. Toujours dans le domaine de transport, soutenus et promus par la RATP, la méthode B et le langage Event-B ont été appliqués avec succès à cette industrie du transport, à travers les pilotes automatiques de métros installés dans le monde entier. Dans [9], les auteurs présentent l'introduction de la méthode B et du langage Event-B dans plusieurs processus de développement industriels avec plus ou moins de succès, même si de nouveaux outils pratiques étaient disponibles pour faciliter l'acceptation dans l'industrie.

En 2011, la division AWS d'Amazon a commencé à utiliser TLA+ pour répondre aux exigences énoncées dans leurs obligations contractuelles, en vérifiant à la fois leurs conceptions actuelles et celles optimisées de manière dynamique [110]. Amazon estime que les méthodes formelles "accélèrent à la fois le temps de mise au marché et la qualité de leurs projets". Depuis lors, ils ont étendu leurs efforts, utilisant récemment OPENJML pour l'analyse de certains de leurs composants [41].

En outre, les méthodes formelles ont également été utilisées avec succès dans diverses domaines, par exemple, pour améliorer la qualité des noyaux de systèmes d'exploitation [18, 92], dans la compilation [99, 108], dans les services de télécommunication [76, 124] pour prouver ou réfuter les propriétés des protocoles de cryptographie [105], dans la signalisation ferroviaire [17, 66], pour le transport par métro [22, 33], dans les systèmes de contrôle du barrage anti-tempête de Maeslant [88, 129] et du pont d'Algera [83], pour les interfaces utilisateur [131], dans la conception assistée par ordinateur [19], dans la défense [31], pour les systèmes d'éclairage[79], et dans une multitude d'autres domaines [14, 45, 112, 133].

Enfin, les tentatives de vérification formelle d'algorithmes, de protocoles et de leurs implémentations largement utilisés révèlent parfois qu'ils sont incorrects (par exemple, dans le cas du protocole Needham-Schroeder [102] ou Timsort [47].)

2.3 Patrons de spécification de propriétés

Les patrons de spécification de propriétés ont été proposés pour faciliter la formalisation des exigences, tout en permettant leur vérification automatique.

La proposition originale des patrons de propriétés était introduite en [62, 63]. Ce système de patrons est destiné à décrire la structure des comportements des systèmes et à fournir des expressions de ces comportements dans une gamme de formalismes communs. Un exemple de ces patrons est illustré à Figure 2.1, certaines parties ont été supprimées pour des raisons de lisibilité [2].

Un patron est composé d'un nom (*Réponse* en Figure 2.1), d'une déclaration (informelle) décrivant le comportement capturé par le patron et d'une déclaration (structurée en anglais)[93] qui doit être utilisée pour exprimer les exigences. Les formules LTL correspondant aux différentes déclinaisons du patron sont également données, où les lettres majuscules (P, S, T, etc.) représentent des états/événements. De manière plus détaillée, un patron est composé de deux parties : (i) la portée (le scope) et (ii) le corps.

Pour les scopes à états limités, l'intervalle dans lequel la propriété est évaluée est fermé à l'extrémité gauche et ouvert à l'extrémité droite. Le corps d'un patron décrit le comportement que nous voulons spécifier.

Dans [103], le framework *Property Specification Pattern Wizard* est présenté. Son objectif est la définition assistée par machine de formules temporelles capturant les propriétés de systèmes basés sur des patrons. Ce système offre une traduction en LTL des patrons encodés dans l'outil, mais il est destiné à faciliter la spécification, plutôt qu'à soutenir la vérification de la cohérence, et il ne peut pas traiter les signaux numériques.

2. Nous avons omis les aspects qui ne sont pas pertinents pour notre travail, par exemple, les traductions vers d'autres logiques comme CTL [62]. Le système complet des patrons est disponible sur `http://ps-patterns.wikidot.com/`

Response

Classification	Order Specification Pattern	
Structured English	*Scope*, if *P* [has occurred], then in response *S* eventually holds.	
Intent	To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause P, must be followed by an occurrence of the second, the effect S	
LTL Mapping	Globally	$\Box(P \rightarrow \Diamond S)$
	Before R	$\Diamond R \rightarrow (P \rightarrow (\neg R \mathcal{U} (S \wedge \neg R))) \mathcal{U} R$
	After Q	$\Box(Q \rightarrow \Box(P \rightarrow \Diamond S))$
	Between Q and R	$\Box((Q \wedge \neg R \wedge \Diamond R) \rightarrow (P \rightarrow (\neg R \mathcal{U} (S \wedge \neg R))) \mathcal{U} R)$
	After Q until R	$\Box(Q \wedge \neg R \rightarrow ((P \rightarrow (\neg R \mathcal{U} (S \wedge \neg R))) \mathcal{W} R)$

FIGURE 2.1 – Patron de réponse

Dans [93], une extension des patrons de [62] est présentée pour permettre la spécification de propriétés en temps réel. [93] ont enrichi leur travail par ajouter la possibilité de "*mappings*" vers la logique temporelle métrique (MTL), la logique arborescente computationnelle temporisée (TCTL) et la logique d'intervalle graphique en temps réel (RTGIL). Les travaux de [93] ont également inspiré un ensemble de travaux récents [59, 58] sur un outil, appelé VISpec. Cet outil permet à l'utilisateur de spécifier des exigences en utilisant une interface graphique, de les traduire en formules MITL [59], puis de déboguer la spécification en utilisant des techniques de vérification.

Une analyse plus approfondie des propriétés du temps réel a incité *Bellini* et ses collègues à développer une organisation unifiée des patrons de spécification [25]. Contrairement à Konrad et Cheng [93] qui considéraient que les patrons en temps réel étaient disjoints des patrons qualitatifs originaux, Bellini et al. ont cherché à les réunir dans un seul catalogue de patrons. Cette réorganisation a non seulement permis d'obtenir un catalogue de patrons uniforme et cohérent, mais elle a également révélé un nouveau patron lié à la précédence des événements sous contrainte temporelle, qui était absent du système original développé par Konrad et Cheng [93]. Ce nouveau catalogue de patrons de spécification unifié fait la distinction entre l'occurrence, la durée et la priorité des événements.

2.3.1 Grammaires structurées

Un catalogue de patrons de spécification des propriétés complet et cohérent ne suffit pas à garantir une application efficace dans des scénarios du monde réel. Les difficultés résident dans la complexité de la spécification et des techniques mathématiques sous-jacentes associées. Les

deux requièrent une expertise considérable. Les ingénieurs de systèmes n'ont pas toujours le niveau de formation nécessaire pour maîtriser leur utilisation [35]. C'est pourquoi plusieurs chercheurs ont décidé d'équiper leurs catalogues avec une interface en langage naturel (plus précisément une "grammaire anglaise structurée") pour permettre une traduction syntaxique des instances de modèles concrets en formules d'une logique temporelle. En [93], les auteurs ont introduit une grammaire anglaise structurée pour capturer la spécification en termes de langage naturel afin de faciliter davantage la compréhension de la signification d'une spécification.

En s'inspirant de la grammaire de [93], les auteurs de [15] ont développé un Framework de patrons de propriétés de spécification (PSPFramework) qui comprend un catalogue unifié de patrons, une interface en langage naturel, et une collection de mappings associés à des formalismes logiques. Pour aider les ingénieurs, le PSPFramework est équipé de PSPWizard [103], un outil qui aide et guide les ingénieurs à exprimer les propriétés du système par le biais de la Grammaire Anglaise Structurée, et à traduire les spécifications en formules de logique temporelle.

2.3.2 Automates observateurs

La notion d'observateur a fait l'objet de nombreux travaux et leurs utilisations sont variées mais reposent toujours sur le même principe : un observateur a pour objectif de surveiller le comportement d'un système afin de vérifier des propriétés, d'interdire des exécutions ou encore de donner un diagnostic.

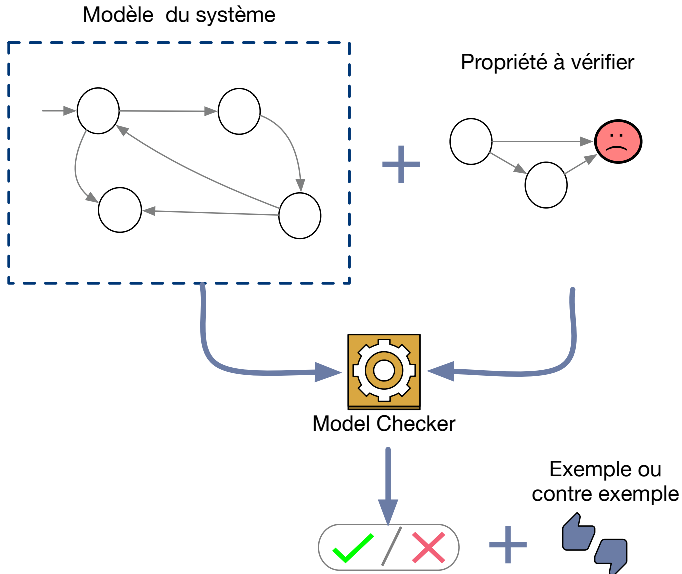


FIGURE 2.2 – Principe de vérification par observateur

Une méthode d'expression des exigences est popularisée par le langage LUSTRE, est basée sur la notion d'observateur [77, 78]. Le processus de test est basé sur une composition parallèle

de l'observateur et du système testé et consiste à effectuer une analyse d'accessibilité sur le système composé (Figure 2.2). L'expression des propriétés à vérifier est plus facile à gérer sous la forme d'un observateur. Le concept d'observateur a d'abord été proposé pour les systèmes parallèles non distribués [53], puis étendu aux systèmes distribués[16]. Les auteurs en [54] généralisent les travaux précédents en présentant une méthodologie de conception générale, applicable à tout système distribué. [54] ont proposé le concept d'observateur pour valider les comportements d'exécution des systèmes distribués, et plus précisément pour concevoir des systèmes communicants auto-valides, c'est-à-dire des systèmes qui détectent les comportements erronés dès que les erreurs agissent sur un certain niveau de sortie observable. Ils montrent que la vérification de satisfiabilité d'une propriété ϕ sur un système S peut être réduit à une analyse d'accessibilité de l'automate de test T_ϕ qui représente ϕ. Cette analyse est effectué en composant l'automate T_ϕ avec le système S. Dans le même optique, des automates de tests ont été proposé en [4, 5].

2.3.3 Transformations de patrons en automates

Les patrons de propriétés ont été conçue pour être simples en termes de clarté et de complexité de calcul. Certains auteurs ont choisi de faire correspondre chaque patron à un problème de model-checking décidable pour qu'il soit facile de les vérifier. Pour atteindre cet objectif, ils ont fourni des méthodes de transformations des patrons aux systèmes de transitions faciles à vérifier par les outils de model-checking (par exemple UPPAAL [23]). Ci-dessous, quelques travaux qui ont suit cette optique.

[74] ont développé un catalogue de patrons de spécification de propriétés en temps réel qui utilise des automates observateurs temporisés comme formalisme de spécification sous-jacent, tandis que, [1] ont fourni une formalisation des patrons de spécification en temps réel sous forme de systèmes de transition temporelles, qui est une généralisation des réseaux de Petri temporels. L'approche **PROPEL** [121] a pour but de faciliter l'écriture et la compréhension des propriétés en fournissant des patrons qui capturent explicitement les détails de propriétés sous forme d'options ou de patrons de propriété. Ces patrons sont représentés en utilisant à la fois un langage naturel "discipliné" et des automates à états finis, ce qui permet au spécificateur de passer facilement d'une représentation à l'autre.

2.3.4 Approches compositionnelles

Dwyer et al. [63] définissent la sémantique de leur catalogue de propriétés en traduisant chaque composition d'un patron et d'un scope en logiques temporelles (LTL, CTL, etc.). Cependant, cette sémantique est translationnelle et n'ont pas compositionnelle et donc difficilement extensible à d'autres patrons/scope. Dans [126], Taha et al. proposent une approche basée sur la composition d'automates définissant la sémantique de chaque patron et de chaque scope par un automate. Ensuite, ils proposent une opération de composition de manière à ce que la

sémantique des propriétés soit définie par la composition des automates (patron, scope). Ainsi, la sémantique est compositionnelle et facilement extensible comme ils le montrent en gérant de nombreuses extensions du langage de Dwyer et al.

Cette proposition ne discute pas la possibilité de composition de différents patrons.

Bien que [63] affirme que les patrons peuvent être imbriqués, aucune autre étude n'a été réalisée sur la définition des patrons composés et leur sémantique. PROPOLS [134] raffine/étend le système de patrons original de[63] en introduisant la composition logique de patrons. Ce mécanisme permet de définir des exigences complexes en termes de patrons de propriétés. PROPOLS utilise le langage d'ontologie web (OWL) comme langage de base. Cela rend les propriétés de PROPOLS partageables et réutilisables dans/à travers les domaines d'application.

De même, [1] ont discuté la possibilité de composer des patrons à l'aide d'opérations logiques. Ils expliquent que la composition de deux patrons consiste en la composition de leurs systèmes de transitions temporisés correspondants. Cependant, comme leurs résultats ne reposent pas réellement sur la définition de la composition, ils ne fournissent pas la définition formelle de l'opération de composition.

2.4 Vérification des systèmes temporels

Afin de vérifier un système, il faut d'abord définir les exigences à vérifier. Les propriétés peuvent être classées en deux grandes catégories :

— Les propriétés de sécurité (*"safety"*) qui expriment que rien de mauvais n'arrive jamais ;
— Les propriétés de vivacité (*"liveness"*) qui affirment que quelque chose de bien finit par se produire.

Généralement, les propriétés sont exprimées à l'aide de logiques ou de modèles temporels. Reste à vérifier ces propriétés !. Dans la littérature, il existe trois approches principales couramment utilisées pour vérifier les propriétés : L'analyse statique, la vérification de théorèmes (*"Theorem-proving"*) et la vérification de modèles (*"Model-checking"*). La différence entre ces approches est le niveau d'abstraction utilisé pour présenter le système à vérifier. Chacune des approches citées ci-dessus (analyse statique, vérification de théorèmes et vérification de modèles) est utilisée pour vérifier un certain type de propriétés dans un système. Par exemple, l'analyse statique est utilisée pour vérifier le code assembleur ou le code source, alors que le Model-checking et le Theorem-proving sont utilisés pour vérifier la spécification.

2.4.1 Analyse statique

L'analyse statique [42, 90] regroupe l'ensemble des techniques permettant de déduire de manière algorithmique un ensemble de propriétés à partir de l'analyse du code source et/ou du code d'assemblage d'un logiciel. Typiquement, l'analyse statique est utilisée à l'étape de la

compilation afin de détecter les bugs courants et d'optimiser le code obtenu sans influencer le comportement du programme. Le compilateur construit à partir du code source un arbre syntaxique abstrait représentant toutes les exécutions possibles. Cette structure est facilement convertie en code assembleur par la suite. Le compilateur vérifie les propriétés de cet arbre et le modifie afin d'optimiser le code assembleur sans modifier le comportement du système (le programme original et le programme optimisé doivent être similaires).

Cependant, cette technique a ses limites. Comme son nom l'indique, il s'agit d'une analyse statique et non d'une analyse dynamique. Cela signifie que nous ne pouvons vérifier que les variables statiques (variables initialisées statiquement dans le programme). Cela élimine la vérification des variables dynamiques car l'analyse statique explore toutes les exécutions possibles, et dans le cas des variables dynamiques, le nombre d'exécutions possibles est potentiellement infini. L'analyse statique peut seulement vérifier des propriétés concrètes sur le code assembleur mais elle rend inaccessibles certaines propriétés représentant le comportement de l'algorithme. Actuellement, il existe des approches qui combinent l'analyse statique et l'interprétation abstraite afin d'inclure certaines de ses variables dynamiques dans la vérification [43]. Mais cette solution est une approximation qui indique des problèmes probables qui doivent être vérifiés par le programmeur lui-même. En conclusion, seules les techniques liées à un haut niveau d'abstraction comme le Theorem Proving ou le model-checking peuvent vérifier les propriétés générales du système.

2.4.2 Preuves de théorèmes (Theorem proving)

Le Theorem Proving [120] est un ensemble de techniques permettant de déduire, par l'utilisation d'un assistant de preuve (PVS [87], Coq [82]), les propriétés sur le comportement d'un logiciel, d'un algorithme ou d'un protocole à partir de l'analyse de son modèle mathématique. La démarche consiste, dans un premier temps, à exprimer le programme et son environnement sous forme d'un modèle mathématique. Ce modèle est obtenu soit par un traitement automatique du code source, soit par interprétation de l'utilisateur lorsque le programme est trop complexe pour être traité automatiquement. Dans les deux cas, il est important de prouver que les propriétés à vérifier se comportent de manière identique dans le programme et dans le modèle mathématique obtenu. L'étape suivante consiste à traduire les propriétés souhaitées dans le même formalisme que le programme, puis à les vérifier à l'aide d'un assistant de preuve. Les propriétés sont considérées comme des énoncés d'un théorème à prouver en utilisant le modèle mathématique du programme et de son environnement en tant que "*axiomes*".

$$\text{Théorème} : \textit{environnement} + \textit{programme} \mid= \text{ propriétés}$$

L'approche par preuve de théorème permet la vérification d'un grand nombre de propriétés du fait qu'elle est basée sur l'intervention humaine. Les étapes triviales de cette approche sont déduites automatiquement par l'assistant de preuve. La plupart des théories mathématiques

utilisées pour représenter les langages de programmation incluent inévitablement l'arithmétique. Cependant, le théorème de Godël [73] affirme que toute théorie mathématique contenant de l'arithmétique est indécidable. Pour cette raison, la preuve de théorèmes présente deux problèmes majeurs liés à l'indécidabilité : (1) ce n'est pas sûrs d'obtenir le résultat en raison du problème d'indécidabilité ; (2) l'intervention humaine est nécessaire dans la plupart des cas car l'assistant de preuve est partiellement automatique. En fait, la preuve de théorèmes est utilisée en parallèle avec le model-checking en générant automatiquement une abstraction finie du système à vérifier. Cette méthode permet de réduire la complexité du système et de résoudre le problème d'indécidabilité [119].

2.4.3 Model-checking

Le Model-checking est l'ensemble des techniques automatisées qui vérifient si le comportement d'un système (logiciel, algorithme ou protocole) répond à ses exigences en explorant son modèle de représentation (automate temporisé, réseaux de Petri temporels, etc.). Les techniques de Model-checking ont été proposées pour la première fois par Clarke et al. dans [39] et Sifakis et al. dans [113]. Depuis lors, cette approche attire de plus en plus l'attention du monde académique comme de l'industrie, principalement parce qu'elle offre une solution rapide et puissante pour la vérification des systèmes finis.

L'approche model-checking est basée sur trois étapes Figure 2.3. La première étape vise à définir un modèle formel du système à vérifier. Les structures couramment utilisées pour représenter le modèle formel sont les structures de Kripke [100] ou les systèmes de transitions étiquetés [40]. Une structure de "Kripke" et un système de transitions étiqueté sont composés d'un ensemble d'états, de transitions entre états et d'une fonction. La seule différence entre eux est que la fonction associe un ensemble de propriétés vérifiées à l'état correspondant dans le cas des structures de Kripke, alors qu'elle associe des actions à chaque état dans le cas des systèmes de transitions étiquetés.

Enfin, la troisième étape vise à montrer que la propriété définie est vérifiée par le modèle. En général, cette étape est réalisée automatiquement par le logiciel qui combine le modèle avec la propriété à vérifier. L'exigence satisfait le système s'il existe un chemin entre l'état initial du système et l'ensemble des états qui vérifient la propriété. Dans le cas contraire, un contre-exemple peut être fourni afin de reproduire un chemin menant à une erreur donnée.

L'avantage d'utiliser des techniques de model-checking est leur approche automatique de type "*push-button*", qui ne nécessite pas de preuves construites à la main [38], ce qui peut être assez fastidieux et difficile à mettre à l'échelle. De plus, un contre-exemple est fourni lorsque la propriété n'est pas vérifiée, ce qui peut aider à trouver l'origine de l'erreur. Un inconvénient du model-checking est qu'il souffre du problème de l'explosion combinatoire des états. C'est-à-dire que le nombre d'états à inspecter peut croître de manière exponentielle en fonction de la complexité du système et peut dépasser la capacité des ressources informatiques disponibles.

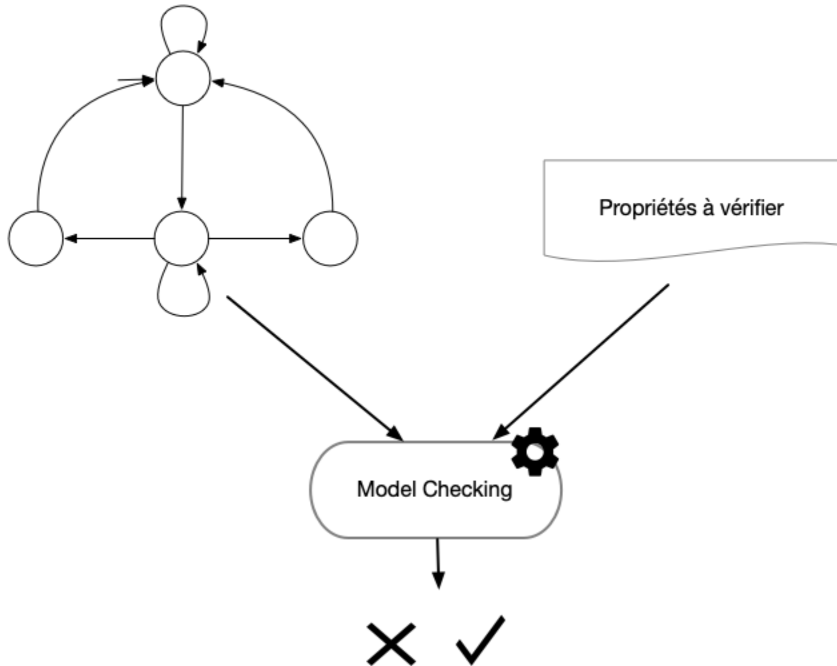


FIGURE 2.3 – Principe de model-checking

Cependant, il est possible de réduire le nombre d'états en choisissant une abstraction qui préserve les propriétés à vérifier, tout en réduisant le nombre d'états à vérifier.

2.4.3.1 Comment spécifier les propriétés ?

Les propriétés sont exprimées à l'aide de deux méthodes principales : les logiques temporelles et les patrons de spécification. Dans cette thèse nous sommes intéressés par les patrons de spécifications, pour cette raison nous n'allons pas tarder sur les logiques temporelles. Chapitre 3 est consacré aux patrons de spécification.

2.4.4 Approche hybride

La vérification des modèles (model-checking) est automatique tandis que, la preuve par théorème ne l'est pas. La preuve par théorème peut traiter des formalismes complexes or, la vérification de modèle ne peut pas le faire. Les forces et les faiblesses de la vérification de modèle et de la preuve de théorème sont clairement complémentaires. Au cours de la dernière décennie, de nombreuses recherches sur la vérification formelle ont tenté de combiner les deux approches de manière synergique, avec divers degrés de succès. Dans cette thèse, nous nous concentrons sur les moyens d'intégration des vérificateurs de modèles et des vérificateurs de théorèmes, plutôt que sur le développement de techniques qui exploitent une telle intégration.

La vérification de modèles est entièrement automatisée étant donné que la logique pro-

positionnelle (ou la vérification du contenu dans les automates) est décidable. Toute formule d'une logique (ou fragment de logique) ne comportant que des types finis peut être définie en logique propositionnelle, mais l'écart de concision entre celle-ci et la logique propositionnelle provoque une explosion de la taille de la formule générée.

Les recherches récentes se sont concentrées sur la réalisation des preuves de base dans les vérificateurs de modèles, car la plupart des recherches de vérification formelle sont motivées par la demande industrielle, qui place la recherche rapide de bugs au-dessus des preuves d'exactitude globale. En effet, les capacités d'automatisation et de comptage des vérificateurs de modèles sont idéales pour la vérification en guise de debugging. Des combinaisons de ce type ont été réalisées avec succès [27, 30, 55, 104, 111, 115].

Cependant, pour un démonstrateur de théorème assez puissant, la vérification de modèle n'est qu'un cas particulier. Idéalement, nous voudrions une situation où un sous-ensemble vérifiable d'un problème de vérification de théorème peut passer directement à un vérificateur de modèle, et ses résultats manipulés dans le démonstrateur de théorème. De cette façon, il est possible d'exploiter toute la puissance de la vérification de modèle sans sacrifier la puissance expressive des démonstrateurs de théorèmes. Le problème consiste à obtenir une traduction fluide de la logique du théorème à un formalisme que le vérificateur de modèle peut comprendre et une traduction fluide des résultats (que ce soit un succès ou un contre-exemple) dans la logique du théorème. Une traduction "fluide", contient plusieurs points :

- La traduction doit être correcte par construction,
- Elle doit être bidirectionnelle et permettre de travailler à plusieurs niveaux d'abstraction.
- Elle doit être efficace.
- Le cadre général doit être suffisamment puissant pour intégrer la plupart des technologies de vérificateurs des modèles.

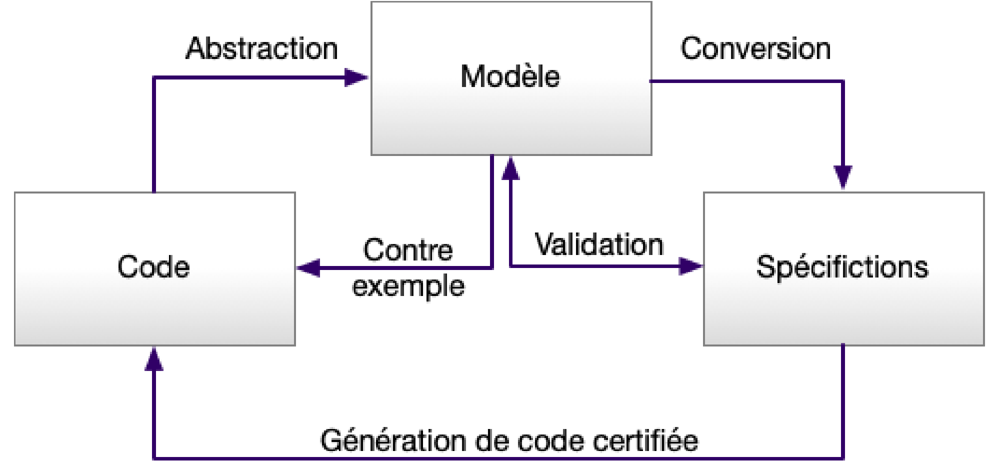


FIGURE 2.4 – Combinaison des méthodes formelles

Les démonstrateurs de théorèmes sont certainement assez puissants pour pouvoir exprimer n'importe quel formalisme de vérification de modèle, et ils sont devenus assez efficaces pour

que la plupart du travail de vérification de modèle puisse être effectué par la preuve dans le démonstrateur de théorèmes sans grande perte d'efficacité. Ainsi, les critères de justesse par construction, d'efficacité, de flexibilité et d'expressivité peuvent être satisfaits.

Basant sur ces critères, nous pensons que l'utilisation d'un démonstrateur de théorème scriptable comme plate-forme de programmation pour les techniques de vérification de modèles est une solution à ce problème. Figure 2.4 présente le principe de la méthodologie de combinaison des deux méthodes (model-checking) et (theorem-proving)

La plupart des programmes informatiques étant trop volumineux pour être entièrement vérifiés, le système logiciel est d'abord analysé et les problèmes potentiels sont identifiés. Des spécifications formelles sont établies pour ces points problématiques. Ces modèles peuvent être obtenus directement à partir du code source ou dérivés de documents d'exigences moins formels. Cependant, en raison de l'explosion de l'espace des états, il peut être conseillé d'utiliser des techniques d'abstraction pour réduire le nombre d'états à traiter. Ces techniques utilisent soit la connaissance du domaine pour filtrer les parties pertinentes du modèle, soit une approche d'évaluation symbolique pour compresser les informations sur l'état. Ensuite, le modèle formel est vérifié par un vérificateur de modèle (model-checker). Si des anomalies sont constatés, l'outil de vérification fournit généralement un contre-exemple. Ce contre-exemple peut être utilisé pour corriger le modèle et le code source. Les solutions sont à nouveau soumises à une vérification avec le vérificateur de modèle. Ces modèles sont ensuite convertis en modèles appropriés pour un vérificateur de théorèmes. Une preuve formelle complète de la spécification est effectuée sur ces modèles. À partir des modèles vérifiés et de leurs preuves, le code est généré. Comme ce code correspond directement aux modèles vérifiés, il sera fiable.

2.5 Conclusion

La fiabilité des logiciels est particulièrement importante lorsque le coût d'une défaillance est élevé en termes monétaires. Elle est encore plus importante lorsque la sécurité humaine est en jeu. Les logiciels sont omniprésents dans les domaines qui ont un impact sur la vie humaine tel que les transports modernes et les systèmes médicaux qui reposent sur des logiciels embarqués. Les causes récurrentes de certains échecs de ces logiciels sont : un manque de protections contre les entrées inattendues ou inappropriées et une inadéquation entre les spécifications (implicites) et le logiciel réel. Un grand nombre de ces défaillances peut être évitées par un respect rigoureux des normes de codage et de conception, ainsi que par des examens approfondis. Cependant, ces méthodes se heurtent à des limites liées à la fois à l'imprévisibilité de certains événements et au manque de rigueur des révisions et des tests humains. L'idée principale des méthodes formelles est de garantir le comportement d'un système informatique en utilisant des approches rigoureuses d'analyse ou de construction. Le comportement est décrit par une spécification, de préférence de nature mathématique.

Dans ce chapitre, nous avons discuté l'intégration des méthodes formelles aux industries

toute en focalisant sur les approches liées à la spécification des propriétés d'un système embarqué en utilisant la notion de patrons de définition. Nous avons aussi cités les travaux relatifs à la transformation de ces patrons en automates.

Chapitre

3

Patrons de propriétés

« To do science is to search for repeated patterns, not simply to accumulate facts. »

— Robert MacArthur

Sommaire

3.1 Introduction **30**

3.2 Les patrons de propriétés **30**

3.2.1 Le système de patrons de Dwyer 30

3.2.2 Approche OBP/CDL 32

3.2.3 Les propriétés temporelles 35

3.2.4 Besoin d'une grammaire structurée 36

3.3 Les observateurs **36**

3.3.1 Bases théoriques 37

3.3.2 Les automates observateurs 41

3.4 Conclusion **44**

3.1 Introduction

Un des problèmes limitant l'adoption des techniques de model-checking par l'industrie est la complexité, pour les non-experts, d'exprimer leurs exigences en utilisant les langages de spécification supportés par les outils de vérification. En effet, il existe souvent un écart important entre les modèles utilisés dans les déclarations d'exigences et les formalismes de bas niveau utilisés par les outils de model-checking, ces derniers reposant généralement sur la logique temporelle. Cette limitation a motivé la définition des langages d'assertion dédiés pour exprimer les propriétés à un niveau supérieur. Cependant, seul un nombre limité de langages d'assertion supportent la définition de contraintes temporelles et encore moins sont associés à un outil de vérification automatique, tel qu'un model-checker.

Ces méthodes de model-checking offrent un moyen puissant de détecter des erreurs souvent subtiles et difficiles à reproduire. Néanmoins, le passage de cette technologie de la recherche à la pratique a été lent et pénible pour diverses raisons. Cependant, les chercheurs montrent que la cause principale réside dans le manque de familiarité des praticiens avec les processus, les notations et les stratégies de spécification. Afin de remédier à ce problème, des approches basées sur des patrons pour la présentation, et la réutilisation des spécifications de propriétés ont été proposées.

Ce chapitre est consacré à l'explication des patrons de propriétés et se concentre particulièrement sur l'approche OBP/CDL.

3.2 Les patrons de propriétés

3.2.1 Le système de patrons de Dwyer

Dwyer et ses collègues de l'Université d'État du Kansas ont développé un système de patrons pour la spécification des propriétés [63, 62]. Ils ont collecté 555 spécifications provenant de plusieurs sources et ont constaté que 92 d'entre elles correspondaient à l'un des patrons de leur système. Ce système de patrons permet à des personnes qui ne sont pas des experts en logiques temporelles de lire et d'écrire des spécifications formelles dans une variété de formalismes. A l'aide de ce système, des propriétés telles que "*Une occurrence de l'événement A doit être suivie d'une occurrence de l'événement B*" peuvent être exprimées.

Un patron de propriété de spécification est une combinaison d'un patron et d'un scope. Le patron spécifie ce qui doit se produire et le scope (*scope*) détermine le moment où le patron se maintient par rapport à d'autres événements.

Dwyer et al. proposent la classification des patrons en deux grandes catégories ; des patrons d'occurrence et des patrons d'ordre, comme le montre la partie gauche de Figure 3.1.

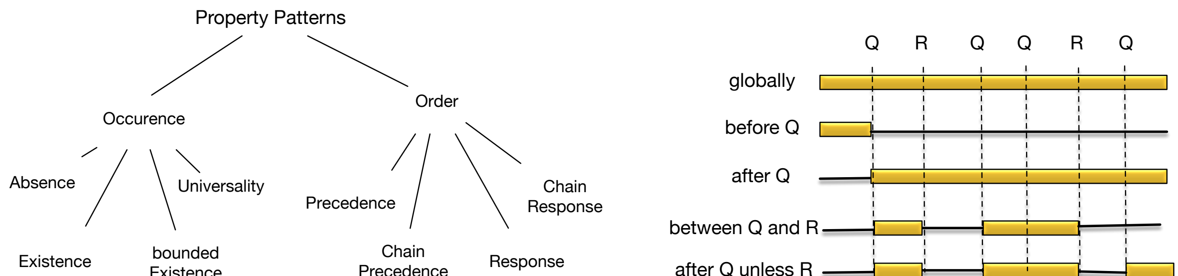


FIGURE 3.1 – Système de patrons de propriétés de Dwyer et al.

Chaque classe est composée de quatre patrons qui sont expliqués dans ce qui suit (variables P ou P' représente un état/évènement donné).

- **Absence** : P ne doit pas se produire dans un contexte déterminé (tout dépend de le scope).
- **Existence** : P doit se produire dans le contexte déterminé par le scope.
- **Existence bornée** : P doit se produire au moins/exactement/au plus k fois en respectant les exigences de le scope.
- **Universalité** : P se produit sur l'ensemble de scope.
- **Précédence** : P doit toujours être précédé par P' selon l'exigence de le scope.
- **Réponse** : P doit toujours être suivi par P' en respectant le scope.
- **Chaîne de précédence/Chaîne de réponse** : une séquence $P_1, ...P_n$ doit toujours être précédée/suivie par une séquence $P'_1, ...P'_m$.

Les scopes permettent de définir quand les patrons ci-dessus doivent être maintenus. Il existe cinq types de scope de base (la variable pt représente n'importe quel patron) :

- pt **globally** : le modèle pt doit se maintenir sur l'ensemble de l'exécution du programme ;
- pt **before** (P) : pt doit se maintenir jusqu'à la première occurrence de P.
- pt **after** (P) : pt doit se maintenir après la première occurrence de P.
- **between** (P) **and** (P') : le patron doit se maintenir entre P et P'.
- **after** (P) **unless** (P') : se comporte comme entre P et P', mais le patron doit tenir même si P' ne se produit jamais.

Ces patrons ont été définis dans plusieurs logiques temporelles telles que (LTL, CTL, QRE ...) et simplifient la spécification des propriétés temporelles.

En effet, Dwyer et al. adoptent la sémantique de translation et fournissent une bibliothèque complète [7] qui contient des centaines de spécifications. Cependant, en raison de cette sémantique, ils ne peuvent considérer et traduire qu'un nombre limité de cas. Nous notons également que pour les utilisateurs d'outils de vérification à états finis, il est difficile de spécifier les exigences du système dans les formalismes temporels (LTL, ...).

3.2.2 Approche OBP/CDL

Partant des constats précédents, des travaux [117, 51, 50, 48] se sont penchés sur cette problématique en cherchant à faciliter l'utilisation des outils de model-checking tout en simplifiant les patrons de Dwyer et al. [63].

Les auteurs de CDL se proposaient d'étudier les conditions et les techniques permettant à un ingénieur de vérifier les exigences sur un modèle logiciel dans un contexte industriel. La contribution souhaitée était de permettre d'exprimer facilement des exigences sous une forme compréhensible par un non-expert des logiques temporelles et de pouvoir effectuer des vérifications par model-checking sur des modèles de grande taille. Le problème d'une telle solution (comme toute approche basée sur des outils de model-checking et systèmes de transition) était l'explosion combinatoire. Afin de résoudre ce problème, d'autres travaux [116] ont essayé de contourner l'explosion combinatoire lors de l'exploration des modèles. Ils ont essayé, de réduire le comportement des modèles lors de leur exploration en considérant leur environnement. Cette réduction est basée sur une description de cas d'utilisation particuliers de l'environnement, appelés *contextes*, avec lesquels le système interagit. L'objectif est de guider le modélisateur à concentrer ses efforts non pas sur l'exploration d'un automate global mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques.

Le langage de description du contexte (CDL) a été introduit pour formaliser la spécification de l'environnement [49]. Le noyau du langage est basé sur le concept de contexte, qui a un comportement *acyclique* communiquant de manière asynchrone avec le système. L'environnement est spécifié dans un certain nombre de contextes de ce type. L'imbrication de ces contextes génère un système de transitions étiquetées représentant l'ensemble des comportements de l'environnement, qui peut être alimenté en entrée par des contrôleurs de modèles traditionnels. De plus, le CDL permet la spécification des exigences grâce à des propriétés qui sont vérifiées par le moteur d'observation OBP (Observer-Based Prover). Figure 3.2 montre un aperçu sur le principe de fonctionnement de l'explorateur OBP.

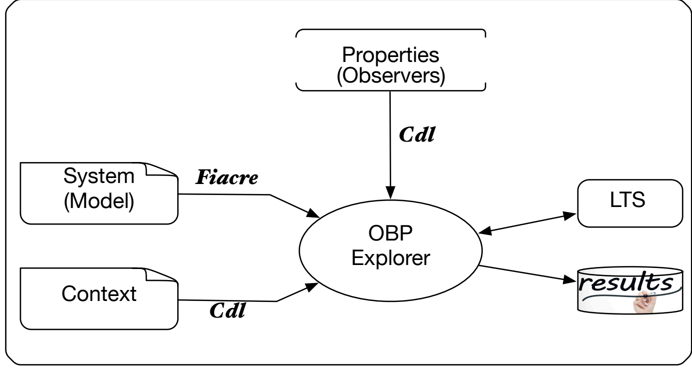


FIGURE 3.2 – Aperçu sur OBP [127]

Le système sous étude est décrit à l'aide du langage formel *Fiacre* [29], qui permet de spécifier les comportements d'interaction et les contraintes par le biais d'automates observa-

teurs. L'environnement et les exigences sont spécifiés à l'aide des patrons de propriétés CDL. Ensuite, l'OBP lance des vérifications sur le système de transitions étiquetées (LTS) obtenu de la transformation des propriétés aux observateurs. Par la suite, l'explorateur OBP fournis le résultat qui peut être un contre-exemple dans le cas où un état de rejet est atteint. Dans le cas échéant, une information sur la validité de propriétés à vérifier sera retournée.

[49] définissent deux formats de propriétés CDL. Le premier se base sur le système de patrons de définition de [63], tandis-que le deuxième est défini par les observateurs.

La section suivante se focalise sur l'explication des propriétés CDL en format de patrons de définition. Le détail sur les observateurs va être discuté en Section 3.3.

3.2.2.1 Les patrons de propriétés CDL

Les patrons CDL combinent des concepts des patrons de [63] (voir Section 3.2.1) et ceux définis en [85].

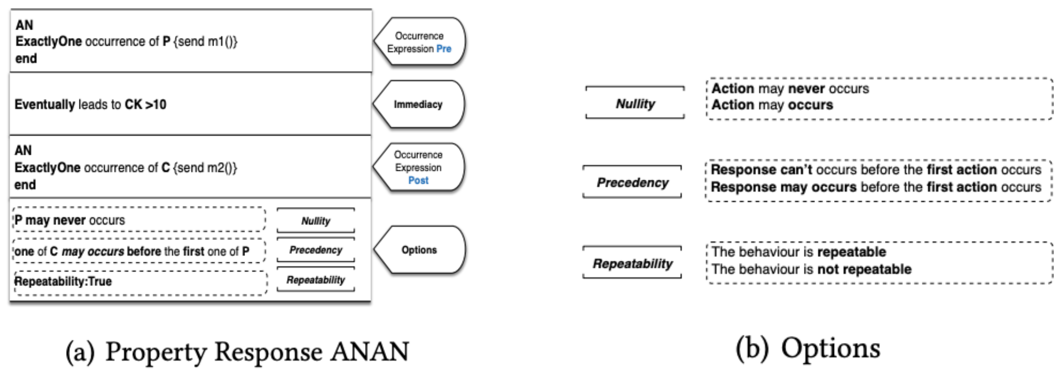
Dhaussy et al. [49] rappellent quatre principaux patrons de [63] qui sont : *Réponse*, *Existence*, *Précédence* et *Absence*.

- ***Le patron réponse*** : définit une *Action* (X) qui doit être suivie par une *Rponse* (Y) "*Après chaque occurrence de X il doit y avoir une occurrence de Y*". Cette propriété représente la propriété la plus commune dans les 555 exemples de spécifications collectionnés par [63]. Nous constatons un exemple typique de patron de réponse exprimée en langage naturel, comme suit : " *Après l'appui sur le bouton de l'ascenseur, l'ascenseur fermera ses portes*".
- ***Le patron précédence*** : ce patron capture une relation d'ordre de base entre une paire d'événements/états. Il modélise le fait que l'occurrence d'une action est une condition préalable nécessaire à l'occurrence d'une autre. En d'autres termes, une action ne peut se produire que si une autre action a déjà eu lieu.
  Les patrons de précédence et de réponse sont bien distincts, même s'ils paraissent similaires. La réponse permet aux effets de se produire sans causes, alors que la précédence exige des causes sans effets subséquents.
- ***Le patron absence*** : il décrit l'étendue de l'exécution d'un système qui est exempt d'un événement/état spécifique (une action donnée ne doit pas se produire durant l'exécution du système).
- ***Le patron existence*** : les patrons d'existence indiquent qu'une action doit se produire durant l'exécution du système. En d'autres termes, il décrit l'étendue d'une exécution du système contenant des événements ou des états spécifiés.

Après des examens plus étroits, la propriété de réponse (présentée dans l'exemple précédent) révélera, qu'il y a plusieurs détails cachés nécessitant plus d'attention relatives aux questions à répondre sur le sens précis, bien qu'il semble simple. Par exemple, Si le bouton garde l'état 'Appuie' de manière successive, les portes doivent-elles se fermer de façon répétée ? Que se

passe-t-il si un événement est autorisé à se produire après l'appui sur le bouton, mais avant que les portes soit fermées ? Dans [121], Smith et al. se rendent compte que ces questions peuvent être capturées en utilisant la notation de la propriété FSA qui vise à rendre l'écriture et la compréhension des propriétés plus facile en fournissant des modèles qui capturent explicitement ces détails et produisant des patrons. Ces patrons sont représentés en utilisant à la fois la langue naturelle "disciplinée" et les automates à état fini toute en permettant aux spécificateurs de se déplacer facilement entre ces deux représentations [49]. inspirent de ce travail, et énoncent les définitions suivantes : trois options avec deux valeurs possibles chacune (repeatability, precedency et nullity).

- **Precedency** : détermine si une réponse peut se produire avant la première occurrence d'une action (ou non).
- **Nullity** : détermine si une action ne devra jamais se produire (ou peut se produire).
- **Repeatability** : détermine si les occurrences d'une action doivent (ou non) être suivies d'une réponse après la première occurrence de la réponse.



(a) Property Response ANAN

(b) Options

FIGURE 3.3 – Patron de propriété réponse CDL et options

Nous proposons de décomposer le patrons CDL en plusieurs parties comme montré dans Figure 3.3 (a) : *PreClause* (resp. *PostClause*) détermine si un seul (*AN*) événement de l'ensemble doit être considéré ou tous (*ALL*) les événements de l'ensemble, ordonnés (*ordered*) ou non (*combined*). *Immediacy* détermine si un événement, autre que ceux présents dans les clauses *Pre* et *Post*, peut se produire éventuellement (*Eventually*) ou non (*immediately*) avant un événement dans la clause *Post*. *Immediacy* spécifie également le délai requis pour l'occurrence de réponse.

La partie *options* (Figure 3.3 (b)) contient : *Nullity* qui détermine si un événement dans la clause *Pre* doit se produire (*P* doit se produire) ou non (*P* ne peut jamais se produire) dans le délai spécifié par *Immediacy*. *Precedency* qui détermine si un événement dans la clause *Post* peut se produire (*C* peut se produire avant la premiére occurence de *P*) ou non. Enfin, l'option *Repeatability* indique si le comportement est répétable ou pas (*true*, *false*).

En plus des options, une autre extension de CDL a également été discutée [50], il s'agissait de la possibilité de manipuler des ensembles d'événements, ordonnés ou non (similaire à la

proposition [85]. On dit que : AN x conduit à y, et cela signifie "Existe-t-il AN x qui conduit à y". L'opérateur ***AN*** agit comme un quantificateur existentiel. D'autre part, si nous demandons si TOUTES les actions mènent à y, nous voulons dire que nous demandons si TOUTES les actions ensemble mènent à y. Ainsi, si toutes sont exécutées, y sera-t-il exécuté ?

Figure 3.3 (a) montre l'exemple de l'ascenseur comme un patron de Réponse AN_AN. Dans cet exemple, les deux clauses (pre, post) contiennent un seul événement à exécuter P, C. L'événement P envoie la requête $m1()$ qui représente dans notre exemple, le fait d'appui sur le bouton. La fermeture des portes est représenté par $m2()$ qui est une réponse à $m1()$ transmise par C. Figure 3.3(b) montre les différentes options de CDL.

Les deux systèmes de patrons cités auparavant (patrons de définition de [63] et patrons CDL [50]) ne se prêtent pas à un raisonnement quantitatif sur le temps.

3.2.3 Les propriétés temporelles

Le catalogue de patrons de propriété de spécification original de Dwyer et al. [63, 62] ne prennent pas en compte les informations relatives au temps. Désormais, ces patrons de spécification de Dwyer et al. sont considérés comme patrons de spécification *qualitatifs* car ils spécifient des propriétés qualitatives qui ne se prêtent pas à un raisonnement quantitatif sur le temps. Plusieurs chercheurs ont travaillé sur cet aspect de temps, et ils ont proposé des patrons étendu en se basant sur les patrons de Dwyer et al. Sur la base d'une analyse des exigences temporelles de plusieurs applications industrielles de systèmes embarqués, [93] ont proposé de créer des patrons de spécification en temps réel selon trois logiques temporelles couramment utilisées. Ils ont illustré l'utilisation de ces patrons dans le contexte des spécifications de propriétés d'un système embarqué pour l'automobile (du monde réel).

[74] ont utilisé un formalisme basé sur les événements. Cela leur permet de décrire des propriétés de la forme : "le point du temps, quand l'événement P se produit". En abrégeant ce point de temps par $t(P)$ et utilisant des termes comme $t(P) \pm k$ pour "k unités de temps après/avant l'occurrence de P".

[74] ont utilisé le concept d'automates observateurs (temporisés) pour décrire le comportement souhaité du système. Intuitivement, les observateurs fonctionnent en parallèle avec le modèle en cours de vérification. Ils atteignent un certain état si et seulement si certaines propriétés peuvent être violées dans le modèle.

En [1], les auteurs proposent une extension temporelle au langage de spécification de de Dwyer et al. [63]. Ils fournissent un langage formel de patrons de spécification qui est suffisamment simple pour faciliter la spécification des exigences par des non-experts et suffisamment riche pour exprimer les contraintes temporelles générales trouvées couramment dans les systèmes réactifs, comme le respect des délais, les limites sur le temps d'exécution le plus défavorable, etc. Pour chaque patron, ils proposent une définition précise basée sur trois formalismes différents : une interprétation dénotationnelle basée sur des formules du premier

ordre sur des traces temporelles, une définition basée sur une notation graphique non ambiguë, et une définition logique basée sur une traduction dans une logique temporelle en temps réel.

Malgré la notion de temps ajoutée pour les patrons CDL [50], ces patrons manquent beaucoup de clarification sur cette notion ainsi que plusieurs propriétés temporelles nécessaires pour la spécification de systèmes embarqués. Nous discuterons les extensions temporelles ajoutées pour CDL dans Chapitre 4.

3.2.4 Besoin d'une grammaire structurée

La logique mathématique, bien qu'elle représente le moyen le plus efficace pour capturer systématiquement les propriétés d'un système, est, en général, trop stricte et la plupart des ingénieurs logiciels préfèrent le langage naturel pour formuler les exigences de leurs systèmes [93, 75]. D'autre part, un langage purement naturel peut être trop faible et sujet aux erreurs, car il n'a pas la rigueur requise pour capturer efficacement les propriétés du système. Une grammaire structurée, en tant qu'interface avec les patrons de spécification des propriétés, permet de remédier à cette lacune [93, 75]. La grammaire anglaise structurée utilise des phrases anglaises de base, mais limite leur composition aux termes dénotables dans les logiques temporelles. Par conséquent, la grammaire anglaise structurée aide à combler le fossé conceptuel entre les deux extrêmes : la spécification en langage naturel et le raisonnement mathématique pur.

Par conséquent, plusieurs chercheurs ont choisi d'équiper leurs catalogues de patrons de spécification par une interface en langage naturel (c.-à-d. une "grammaire anglaise structurée") pour permettre une traduction dirigée par la syntaxe des instances de modèles concrets en formules d'une logique temporelle de choix.

Pour réaliser cette approche, [15] ont développé un framework de patrons de propriétés de spécification (PSPFramework) qui comprend un catalogue unifié de patrons, une interface en langage naturel avec une grammaire structurée et une collection de fonctions de mapping associées à des formalismes logiques.

3.3 Les observateurs

Le principe de vérification étudié dans cette thèse est fondé sur la technique de vérification par observateurs. Les observateurs sont des automates déterminés pour spécifier les propriétés à vérifier. Un observateur peut exprimer des propriétés de sécurité, de vivacité bornée et d'accessibilité. Un observateur [77, 3] est une entité permettant de surveiller le comportement du système afin de détecter ses défaillances. Il est représenté graphiquement sous la forme d'un automate qui se compose avec le système à vérifier. Il est conçu pour coder une propriété logique et est utilisé pour observer tous les événements pertinents liés à cette propriété.

3.3.1 Bases théoriques

L'intérêt majeur des observateurs repose sur leur simplicité d'utilisation et de mise en œuvre. En effet, pour obtenir un résultat, il suffit de réaliser un produit synchrone (entre le système à analysé et l'observateur qui représente la propriété à vérifier sur le système), puis de réaliser une analyse d'accessibilité sur le système composé. En outre, un contre-exemple peut être extrait de façon triviale. Cependant, les observateurs ne peuvent exprimer que des propriétés de sûreté et de vivacité bornée [77, 34, 4]. Les informations provenant d'un contre-exemple sont à la fois complexes et difficiles à interpréter.

Un observateur est un automate temporisé qui est lié au système à valider par composition synchrone. Cet automate a la capacité de capturer des messages et des informations provenant du système. Il permet de décrire une propriété en utilisant des états spéciaux qui sont : l'état de *rejet* et l'état de *succès*. Si au moins un état de *succès* est atteint, cela signifie que la propriété est valide. Un état de rejet permet de vérifier les propriétés de sûreté et de vivacité bornée. Un état de succès permet de vérifier les propriétés d'accessibilité.

> **Remarque.** *La présence de ces deux types d'états dans un observateur est en effet bien souvent problématique. Il faut éviter cette situation qui peut conduire à des résultats peu concluants : soit aucun des états spéciaux n'est accessible, soit les deux types d'états spéciaux sont accessibles. Pour cette raison, dans cette thèse, la majorité des observateurs utilisés n'ont que des états de rejet car nous sommes intéressés par l'analyse d'accessibilité.*

Dans cette section, nous commençons par aborder les notions théoriques de base des systèmes de transition et les automates temporisés [10]. Ensuite, nous discutons la vérification des ces systèmes. Nous entamons ensuite les observateurs [77, 117].

3.3.1.1 Systèmes de transitions

Afin de discuter le comportement des systèmes de manière formelle, il est nécessaire de les représenter en termes de structure mathématique. La façon la plus simple par laquelle on peut représenter un tel comportement consiste à utiliser des automates ou des systèmes de transition étiquetés. Il s'agit simplement de graphes contenant des nœuds et des arêtes orientées et étiquetées. Les nœuds représentent les états possibles du système et les arêtes (ou transitions) représentent les actions sous forme de transitions entre deux nœuds. Un système de transitions étiquetées est un graphe dont les arêtes sont étiquetées par une action appartenant à un alphabet. L'ensemble des traces obtenues par le parcours d'un système de transitions est son langage, noté $\mathcal{L}(S)$, avec S un système de transitions.

Définition 3.3.1 (Système de transition étiqueté [2]). *Un système de transition étiqueté est un n-uplet* $(S, \Sigma, \longrightarrow, s_0)$ *où :*

- S *est un ensemble des états,*
- Σ *est un ensemble fini d'étiquettes (actions de S)*
- $\longrightarrow \subseteq S \times \Sigma \times S$ *est une relation de transition, écrite sous la forme :* $s \xrightarrow{a} s\prime$ *pour* $\langle s, a, s\prime \rangle \in \longrightarrow$
- s_0 *est l'état initial.*

Pour $s \in S$ et $\alpha \in \Sigma$, l'ensemble α-successeurs de s noté $Post(s, \alpha)$ est défini comme suit :

$$Post(s, \alpha) = \{s\prime \in S : s \xrightarrow{\alpha} s\prime\}$$

et l'ensemble de successeurs de s noté $Post(s)$ est défini comme suit :

$$Post(s) = \bigcup_{\alpha \in \Sigma} Post(s, \alpha)$$

De même, l'ensemble de α-prédécesseurs et les prédécesseurs de s sont définis comme suit :

$$Pre(s, \alpha) = \{s\prime \in S : s\prime \xrightarrow{\alpha} s\}, \quad Pre(s) = \bigcup_{\alpha \in \Sigma} Pre(s, \alpha)$$

Remarque. *Un état s est appelé* ***terminal*** *si est seulement si :* $Post(s) = \emptyset$

3.3.1.2 Automates temporisés

Les automates temporisés ont été proposés dans [10] comme une extension de l'approche théorique des automates à la modélisation des systèmes en temps réel. Depuis lors, la théorie et les outils de vérification des automates temporisés constituent un domaine de recherche intensif en informatique. Un automate temporisé est un automate fini équipé d'un ensemble fini d'horloges. Les horloges sont des fonctions continues à valeur réelle du temps qui enregistrent précisément le temps écoulé. Toutes les horloges avancent au même rythme. Les comportements des systèmes en temps réel sont capturés à l'aide d'automates temporisés en réinitialisant explicitement les horloges et en comparant la lecture d'une horloge avec des constantes.

Définition 3.3.2 (Automate temporisé [10]). *Un automate temporisé est un 7-uplet* $(S, init, F, \Sigma, C, Inv, T)$ *où :*

- S *est un ensemble finis des états,*
- $init \in S$ *est l'état initial,*
- $F \subseteq S$ *est un ensemble d'états finis,*
- Σ *est un ensemble d'actions/événements,*
- C *est un ensemble d'horloges,*
- Inv *est une fonction d'affectation de propositions qui, pour chaque état, attribue une proposition vraie à l'état,*
- $T \subseteq S \times \Sigma \times 2^C \times \Phi(C) \times S$ *est une relation de transition,*

$\Phi(C)$ est un ensemble de contraintes d'horloge qui est défini par la grammaire suivante : $\varphi := true | x \leq c | c \leq x | x < c | x < c | \varphi_1 \wedge \varphi_2$ où x est une horloge et c est un nombre réel. Les contraintes d'horloges, incluses dans les gardes, sont des conjonctions de contraintes atomiques. Une contrainte atomique compare la valeur d'une seule horloge avec une constante de temps.

Une transition $(s, a, \lambda, \delta, s\prime)$ représente une transition de s vers $s\prime$ avec l'évènement a, λ représente l'ensemble d'horloges à réinitialiser avec cette transition, et δ est une contrainte d'horloge sur C qui spécifie quand le commutateur est activé.

Notons que F peut être vide si l'automate temporisé ne se termine jamais (c'est-à-dire s'il n'y a pas d'état de blocage). La sémantique d'un automate temporisé est définie comme un système de transitions. Une exécution de l'automate temporisé commence par l'état initial. Le contrôle peut rester à un état s tant que l'état $Inv(s)$ n'est pas violé. Une transition $(s, a, \lambda, \delta, s\prime)$ est activée si et seulement si le contrôle est à l'état s, si la condition de garde δ est satisfaite (par la valorisation des horloges) et si l'événement a est activé. La transition est non gardée si δ est vrai. Après avoir pris la transition, la commande passe à l'état $s\prime$ et les horloges en λ se réinitialisent.

3.3.1.3 Exemple

L'illustration de Figure 3.4 montre un exemple d'automate temporisé spécifiant une porte de wagon de train [60].

Initialement, la commande est à l'état "Close", indiqué par le cercle double. La transition est déclenchée dès qu'une entrée est reçue sur le canal ouvert (car la transition n'est pas contrôlée par le temps), la commande passe à l'état *ToOpen* et l'horloge x est remise à zéro. Dans les 2 secondes (contraintes par l'invariant d'état), la commande passe à l'état Open. La porte reste ouverte pendant 10 secondes, c'est-à-dire que les passagers ont 10 unités de temps pour monter à bord, puis la commande passe à l'état Close.

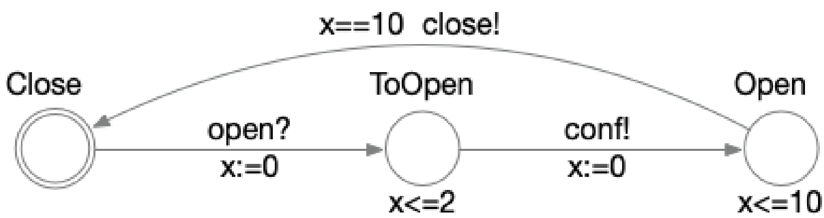


FIGURE 3.4 – Automate temporisé exemple [60]

3.3.1.4 Compositions et communications

Pour modéliser les systèmes concurrents, les automates temporisés peuvent être étendus avec la composition parallèle. Dans les algèbres de processus, divers opérateurs de composition parallèle ont été proposés pour modéliser différents aspects de la concurrence (voir par exemple CCS et CSP [106, 81]). Ces opérateurs algébriques peuvent être adoptés dans les automates temporisés. Dans le langage de modélisation UPPAAL [96], l'opérateur de composition parallèle CCS [106] est utilisé.

1. ***Sémantique opérationnelle :*** la sémantique d'un automate temporisé est définie comme un système de transition où un état ou une configuration consiste en l'emplacement actuel et les valeurs actuelles des horloges. Il existe deux types de transitions entre états.
   - discrète : $\{s_1, a, \lambda, \delta, s_2\}$ représente une transition de l'état s_1 vers l'état s_2 avec une étiquette a. δ est une contrainte d'horloge spécifiant que la transition est possible, et l'ensemble $\lambda \in C$ est l'ensemble des horloges réinitialisées sur la transition.
   - de délai : $\{s,\ \epsilon(d)\}$ représente le passage du temps de d unités dans l'état s.

Définition 3.3.3 (Sémantique opérationnelle [106]). *La sémantique d'un automate temporisé est un système de transitions où les états sont des paires $\langle l, u\rangle$ et les transitions sont définis comme suit :*

- $\langle l, u\rangle \xrightarrow{d} \langle l, u + d\rangle$ *si* $u \in I(l)$ *et* $u + d \in I(l)$ *pour* $d \in \mathbb{R}_+$
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ *si* $l \xrightarrow{g,a,r} l'$ *et* $u \in g$, $u\prime = [r \mapsto 0]u$ *et* $u' \in I(l')$

u, v sont des fonctions connues sous le nom d'affectations d'horloge qui font correspondre C à des réels non négatifs $\mathbb{R}_+$. $u \in g$ signifie que les valeurs d'horloge désignées par u satisfont la garde g. Pour $d \in \mathbb{R}_+$, $u + d$ désigne l'affectation d'horloge qui relie tous les $x \in C$ à $u(x) + d$, et pour $r \subseteq C$, $[r \mapsto 0]u$ désigne l'affectation d'horloge qui fait correspondre toutes les horloges dans r à 0 et qui s'accorde avec u pour les autres horloges de $C \setminus r$.

2. ***Analyse d'accessibilité :*** la question la plus pertinente à se poser sur un automate temporisé est peut-être celle de l'accessibilité d'un état final donné (ou d'un ensemble d'états finaux). Ces états finaux peuvent être utilisés pour caractériser les propriétés de sécurité d'un système.

   Le but de l'analyse d'accessibilité est de vérifier si certaines propriétés sont satisfaites ou non. Pour un modèle et une propriété donnés, nous construisons un observateur, un automate qui surveille le comportement global et signale une erreur en cas de violation de la propriété. Le processus de vérification consiste à explorer l'espace d'état du système produit. En cas de violation de propriété, on obtient une trace d'erreur qui peut être visualisée avec certains outils de model-checking [89].

Définition 3.3.4. *On écrit* $\langle l, u \rangle \rightarrow \langle l', u' \rangle$ *si* $\langle l, u \rangle \xrightarrow{\sigma} \langle l', u' \rangle$ *pour* $\sigma \in \Sigma \cup \mathbb{R}_+$ *pour un automate avec un état initial* $\langle l_0, u_0 \rangle$, $\langle l, u \rangle$, *est accessible ssi* $\langle l_0, u_0 \rangle \rightarrow^* \langle l, u \rangle$

3. ***Composition parallèle :*** la composition parallèle d'un ensemble d'automates est essentiellement le produit des automates. La construction de l'automate produit est une opération entièrement syntaxique mais coûteuse en termes de calcul.

   Un état dans le produit de deux automates temporisés est une paire d'états, chacun représentant le progrès que l'un des automates temporisés a fait à ce stade. Une transition dans le produit est soit une transition locale de l'un ou l'autre automate, soit une synchronisation entre les deux composants. Ce qui suit définit formellement la composition parallèle.

Définition 3.3.5 (Composition Parallèle). *soit* $A_i = (S_i, init_i, F_i, \Sigma_i, C_i, Inv_i, T_i)$ *où* $i \in 1, 2$ *deux automates temporisés. La composition parallèle de* A_1, A_2 *est :*
$A_1 || A_2 = (S_1 \times S_2, (init_1 \times init_2), F_1 \times F_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, Inv, T)$ *où :* $\forall (s_1, s_2) : S_1 \times S_2 \bullet Inv(s_1, s_2) = Inv_1(s_1) \wedge Inv_2(s_2)$ *et* T *est la plus petite relation de transition qui satisfait ces deux conditions :*

- *Si* $(s, a, \lambda, \delta, s\prime) \in T_i \wedge a \notin \Sigma_{3-i}$ *avec* $i \in 1, 2$, *puis,* $((s, t), a, \lambda, \delta, (s', t'))$
- *Si* $(s_1, a, \lambda, \delta, s_2) \in T_i$ *et* $(s_1', a, \lambda', \delta', s_2') \in T_{3-i}$, *alors,* $((s_1, s_1'), a, \lambda \cup \lambda', \delta \wedge \delta', (s_2, s_2')) \in T$

3.3.2 Les automates observateurs

La notion d'observateur a fait l'objet de nombreuses études comme [54] pour la conception de systèmes distribués auto-vérifiants. Elle a également été utilisée dans les automates de test

comme mentionné dans [5], ainsi que dans les hypothèses environnementales [77, 78]. Les utilisations des observateurs sont diverses [32], mais reposent toujours sur le même principe. Tableau 3.1 regroupe les différents travaux sur les observateur.

	Références	Propriétés
Observateurs basés sur les automates	Automates de Büchi [71]	Utilisés dans la vérification de propriétés de logique linéaire
	Automates de test [3, 5]	Utilisés pour la vérification de propriétés de sûreté.
Autres Observateurs	Observateurs de l'algèbre de processus Estelle [13]	Implantés dans le simulateur Véda [86]
	Observateurs pour systèmes auto-validés [54]	Ils sont décrit comme un concept permettant la création de systèmes distribués auto-validés.
	Observateurs de machines synchrones [77].	Vérification de propriétés de sûreté, à l'aide d'observateurs sur des programmes réactifs et synchrones.

Tableau 3.1 – Tableaux récapitulatif de travaux sur les observateurs

De manière informelle, l'observation d'un système est réalisée par la composition de l'automate qui représente le modèle du système *testé* avec un *observateur*. Le processus de test consiste ensuite à effectuer une analyse d'accessibilité dans le système composé.

Afin d'effectuer cette analyse, nous devons vérifier deux propriétés principales. La propriété de non-accessibilité prévoit que l'observateur ne peut atteindre un état donné (*Reject*) dans aucune des exécutions possibles. Une violation de la spécification (*un contre-exemple*) est détectée lorsque l'observateur peut atteindre un tel emplacement. Alors que la propriété d'accessibilité exige que chaque parcours se termine dans un état (*Success*).

La notion d'automate observateur que nous utilisons dans cette thèse est une variation des automates de test originaux présentés dans [5], et ceux introduits dans [117].

Définition 3.3.6. *Soit $\mathbb{X}$ un ensemble d'horloges. $\mathbb{N}$ et $\mathbb{R}_{\geqslant 0}$ représentent respectivement l'ensemble des naturels et des réels positifs. $\mathcal{BX}$ est l'ensemble des expressions booléennes sur les formules de la forme $a \sim b$ avec $a \in \mathbb{X}$, $b \in \mathbb{N}$ et $\sim \in \{<, >, =\}$. Une évaluation de l'horloge v est une fonction de $\mathbb{X}$ vers $\mathbb{R}_{\geqslant 0}$.*
Donnant une condition $g \in \mathcal{BX}$ et une évaluation d'horloge v, la valeur booléenne $g(v)$ décrit le fait que g est satisfaite par v ou non.

Un observateur définit dans ce travail décrit une propriété à l'aide d'états spéciaux qui sont *rejet* ou *succès*. L'état de rejet permet de statuer sur la vérification ou non d'une propriété. Une propriété est vérifiée si elle aucun état de rejet n'est atteint. Un état de rejet permet de vérifier des propriétés de sûreté et de vivacité bornée. Un état succès permet de vérifier des propriétés d'accessibilité.

Nous utilisons les mêmes observateurs de [117]. Par conséquent, les observateurs de rejet (resp. succès) sont ceux dont l'analyse d'accessibilité est effectuée sur les états de rejet (resp. succès).

Définition 3.3.7 (Observateur)**.** *Un automate observateur est un 6-uplet* $\mathcal{O} = \langle S, s_0, \Sigma, \mathbb{X}, E, F \rangle$ *où :*

- S *est un ensemble fini d'états,*
- $s_0 \in S$ *est l'état initial,*
- $\mathbb{X}$ *est un ensemble fini d'horloges,*
- Σ *est un ensemble finit d'alphabet (évènements)*
- $E \subseteq S \times \Sigma \times \mathbb{X} \times 2^{\mathbb{X}} \times \mathscr{BX} \times \mathbb{S}$ *est l'ensemble des transitions,* $\langle s, l, \varphi, g, s' \rangle$ *est une transition entre* s *et* s' *avec l'action* l*,* φ *représente l'ensemble d'horloges à réinitialiser* 0 *ou désactiver* -1 *(on utilisera plutôt* $s \xrightarrow{l,\varphi,g} s'$ *au lieu* $\langle s, l, \varphi, g, s' \rangle$*).*
- $F \subseteq S$ *est l'ensemble des états finaux :* $F = \{success, reject\}$*.*

Figure 3.5 illustre un observateur correspondant à un patron CDL. Il est composé d'un ensemble d'états, et de transitions, ainsi qu'un état final de rejet. Cet observateur peut être utilisé pour une analyse d'accessibilité pour vérifier si une propriétés est valide ou non. Si au moins un chemin d'exécution mène à l'état de rejet, cela indique que la propriété n'est pas vérifiée.

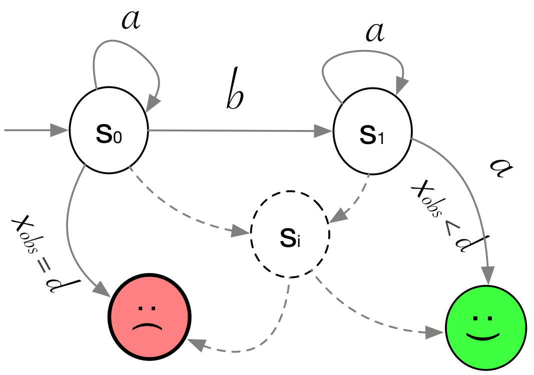


FIGURE 3.5 – Automate observateur

3.3.2.1 Propriétés

Le but de la validation est de vérifier que la description d'un modèle satisfait certaines propriétés. Le modèle peut être modélisé à l'aide d'automates et les propriétés peuvent être

décrites à l'aide de logiques de spécification. Cependant, dans notre travail nous sommes intéressés par la description des propriétés à l'aide des observateurs plutôt qu'à l'aide des patrons de définition. Nous donnons ci-dessous des définitions brèves des propriétés les plus pertinentes pour la vérification des systèmes embarqués et à temps réel.

N.B : Nous notons la relation de satisfaction d'une propriété P sur un système $\mathcal{S}$ comme suit : $\mathcal{S} \models P$. L'ensemble des exécutions de $\mathcal{S}$ est un système de transitions $\mathcal{G}(\mathcal{S})$, nommé *graphe d'accessibilité*.

Définition 3.3.8 (Propriété d'accessibilité). *P est une propriété d'accessibilité, elle assure l'accessibilité d'une configuration particulière.*

Exemple de propriété d'accessibilité pour un ascenseur : La porte du rez-de-chaussée peut s'ouvrir.

Définition 3.3.9 (Propriété de vivacité). *P est une propriété de vivacité si elle assure qu'une configuration sera fatalement atteinte sous certaines conditions.*

Exemple de vivacité pour un ascenseur : S'il y a une demande pour un étage particulier, l'ascenseur le desservira éventuellement.

Définition 3.3.10 (Propriété de sûreté). *P s'assure qu'une configuration particulière n'est jamais accessible.*

Exemple de propriété de sûreté pour un ascenseur : l'ascenseur ne doit jamais se déplacer avec des portes ouvertes.

Il existe d'autres propriétés qui sont temporelles, elles sont présentées dans [10].

3.4 Conclusion

Les patrons CDL permettent d'exprimer des propriétés de réponse, de précédence, d'absence et d'existence. Les propriétés font référence à des prédicats et des évènements détectables tels que des envois ou réceptions de valeurs entre processus du modèle, des changements d'état de processus ou de variables. Ces patrons (ou formes) de base ont été enrichis par des éléments optionnels (Precedency, Nullity, Repeatability) à l'aide d'annotations dans l'esprit de [121]. Des opérateurs *an* et *all* précisent respectivement si un évènement ou tous les évènements, ordonnés (ordered) ou non (combined), d'un ensemble d'évènements sont concernés par la

propriété comme proposé par [85]. Pour mettre en œuvre la vérification de propriétés, l'outillage OBP [52] transforme les propriétés en automates observateurs non intrusifs dotés d'états de rejet (reject). Un observateur est construit de manière à encoder une propriété logique [5] et a pour rôle d'observer, partiellement et de manière non intrusive, les évènements significatifs survenant dans le modèle du système à valider. Lors de la simulation, il est composé, de manière synchrone, au modèle à observer. La vérification de propriété consiste alors en une analyse d'accessibilité des états de rejet sur le graphe d'exploration qui résulte de la composition du modèle à valider et des observateurs. Dans ce chapitre, nous avons présenté un aperçu sur les patrons de propriétés en général, en accordant une attention particulière aux patrons CDL et automates observateurs. Les notions présentées dans ce chapitre sont utilisés par la suite dans nos contributions qui commencent dans Chapitre 4 par une extension de langage CDL pour exprimer des propriétés plus complexes.

Troisième partie

Contributions

Chapitre 4

Extension CDL

« I believe in innovation and that the way you get innovation is you fund research and you learn the basic facts. »
— Bill Gates

Sommaire

4.1 Introduction . . . 50
4.2 Système de patrons de propriétés ECDL . . . 50
4.2.1 Expression des patrons de propriétés ECDL . . . 51
4.3 Patrons de propriétés ECDL . . . 52
4.3.1 Patrons de propriétés qualitatives . . . 53
4.3.2 Patron de précédence . . . 54
4.3.3 Patron d'absence . . . 55
4.3.4 Patron d'existence . . . 55
4.3.5 Patrons de propriétés temporelles . . . 56
4.3.6 Options ECDL . . . 56
4.3.7 Définition des scopes . . . 58
4.4 Grammaire structurée . . . 59
4.5 Intérêts des patrons de propriétés ECDL . . . 62
4.6 Conclusion . . . 63

4.1 Introduction

Les patrons CDL existants identifiés dans [50, 48] permettent de raisonner sur l'occurrence et l'ordre des événements, mais pas suffisamment sur leur timing. Par ailleurs, la notion de scopes en CDL est un peu négligée. Dans ce chapitre nous proposons, d'une part, de clarifier et de mettre en évidence certains formalismes définis dans CDL existant. D'autre part, nous suggérons d'étendre CDL en un langage nommé ECDL qui couvrent des propriétés temporelles ainsi que la notion des scopes.

Un autre aspect important de notre étude porte sur la définition d'une grammaire en langage naturel qui est utilisée pour dériver des phrases en langage naturel qui peuvent être mises en correspondance avec des spécifications formelles structurées en termes d'un système de patrons de spécifications. En effet, seule une bonne intégration du formalisme dans un processus encadré peut permettre l'exploitation effective, par l'utilisateur, de la technique de vérification que nous avons proposée. Dans cette partie, nous décrivons également quelques règles méthodologiques pour encadrer la pratique de l'utilisateur et faciliter l'expression ainsi que la vérification des propriétés CDL.

Cette grammaire offre également une aide à la spécification des propriétés ainsi qu'à leurs transformations, étant donné la difficulté de les encoder en observateurs. Nous développons une proposition d'expression des propriétés à partir de patrons identifiés en CDL. L'idée, à terme, est de pouvoir proposer, dans l'environnement utilisé, une bibliothèque de patrons de spécification de propriétés simples à exprimer à l'aide d'une grammaire de langage naturel. Ce chapitre présente le processus d'expression de patrons de propriétés basé sur la grammaire proposée dans cette thèse.

4.2 Système de patrons de propriétés ECDL

L'objectif principal de ce travail est de simplifier et enrichir la spécification des propriétés des systèmes définis en CDL. À cette fin, nous proposons une approche à trois niveaux comprenant un catalogue unifié de patrons, une interface en langage naturel basée sur la grammaire anglaise structurée, et une collection de fonctions de transformations vers des automates observateurs appropriés. Figure 4.1 illustre la structure de notre système de patrons ECDL.

Notre système de patrons ECDL favorise un processus de rédaction de spécifications qui non seulement aide les ingénieurs des systèmes à capturer les propriétés d'une manière concise et correcte, mais fournit également les moyens d'effectuer des spécifications de propriétés au bon niveau d'abstraction. Plus précisément, lorsqu'un ingénieur commence à définir une propriété du système, il peut le faire de manière descendante en commençant par la catégorie de propriété la plus générale et la plus abstraite. Une fois qu'une structure initiale de propriété a été choisie, il peut affiner la propriété en se concentrant sur des attributs subordonnés spécifiques

(par exemple, temps borné, ordre d'occurrence des évènements). Contrairement à ce qui est proposé en [75] [93], où l'ingénieur doit s'engager dans une catégorie spécifique (c'est-à-dire une spécification qualitative, temps réel ou probabiliste) comme première action.

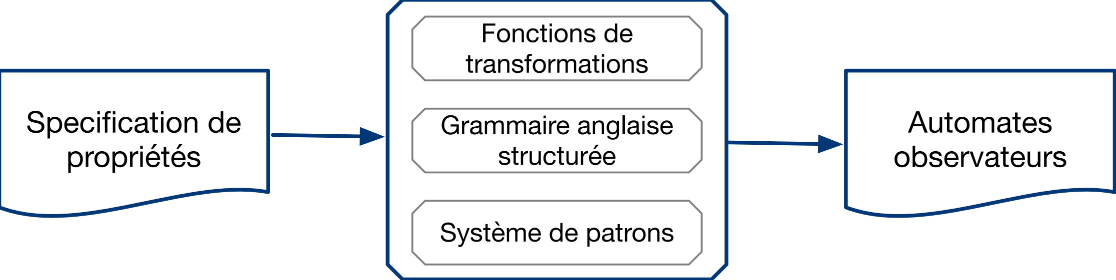


FIGURE 4.1 – Schéma du système ECDL

Cette section donne l'ensemble des patrons de spécification en temps réel que nous avons proposés. Nous décrivons également notre grammaire anglaise structurée, et nous donnons un exemple de modèle de spécification en temps réel tout en comparant le CDL classique avec notre langage ECDL.

4.2.1 Expression des patrons de propriétés ECDL

Les patrons de propriétés proposent des templates qui doivent être adaptés à des fins spécifiques. Par conséquent, lorsque l'on tente d'appliquer les patrons à un domaine spécifique, deux problèmes se posent :

- Les patrons doivent capturer les connaissances pertinentes pour le domaine spécifique considéré. Est-ce que toutes les connaissances pertinentes (ou, du moins, une partie suffisante des connaissances pertinentes) sont capturées par l'ensemble actuel de modèles utilisés ?
- La manière dont les propriétés sont formulées doit être adaptée aux logiques et aux approches de modélisation utilisées dans le domaine d'intérêt. Les langages et les stratégies d'encodage utilisés dans les patrons sont-ils adéquats ?

Afin de répondre à ces deux questions, il est nécessaire de réaliser une étude des spécifications de propriétés dans le domaine des systèmes embarqués (notre domaine d'intérêt dans le contexte actuel). L'objectif de l'étude est de rassembler les propriétés utilisées dans la littérature et rechercher d'éventuels patrons. Dans cette thèse, nous réalisons seulement une partie de cette étude pour définir de nouveaux patrons, car, d'une part, les patrons CDL se basent sur l'étude originale de [63] ainsi que d'autres études qui datent déjà d'un certain nombre d'années, ce qui pose la question de savoir si de nouveaux patrons sont apparus. D'autre part, nous nous intéressons à un domaine d'application particulier qui est celui des systèmes embarqués,

par ailleurs les créateurs de CDL [49] ont déterminé un ensemble de patrons spécifiques à ce domaine. Ces patrons ont été utilisés pour vérifier certains propriétés des systèmes avioniques.

Notre système de patrons reprend les quatre patrons définis dans le CDL classique qui sont : "*Réponse*", "*Précédence*", "*Absence*" et "*Existence*", et défini de nouveaux patrons temporels.

Les patrons de spécification temps réel que nous proposons visant à la vérification de systèmes réactifs avec des contraintes temps réel. Notre objectif principal est d'enrichir le catalogue CDL afin de pouvoir exprimer un nombre élargie de propriétés temporelles. Nos patrons [21] sont conçus pour exprimer des contraintes temporelles générales couramment rencontrées dans l'analyse des systèmes temps réel (telles que le respect des délais, la durée des événements, etc.). Ils sont également conçus pour être simples en termes de clarté. Nous définissons une liste de patrons, des options et des scopes comme montré dans Figure 4.2.

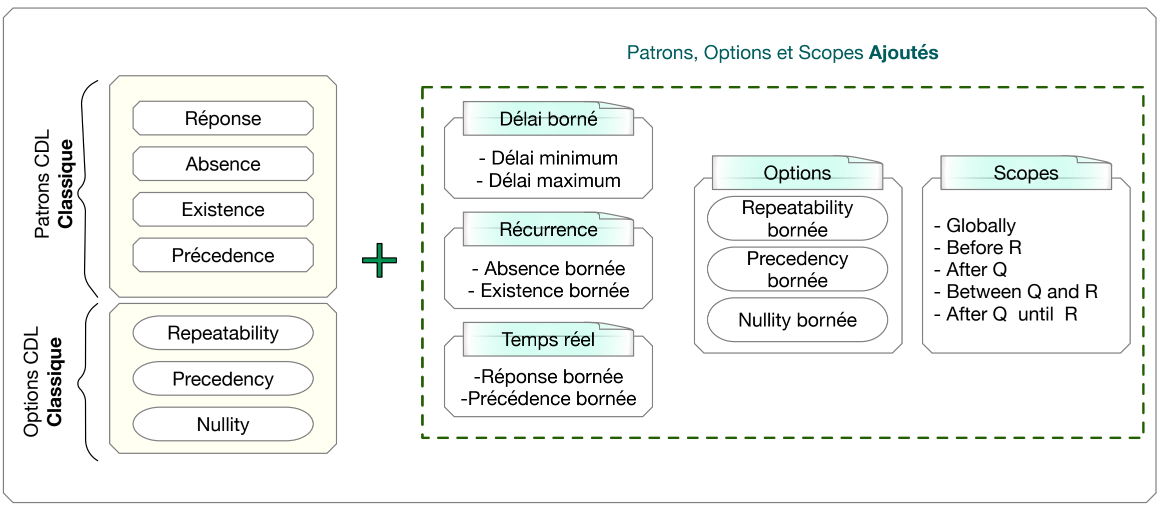


FIGURE 4.2 – Système de patrons ECDL

En ce qui suit, nous allons détaillé les patrons de propriétés ECDL que nous avons proposé, passant ensuite aux options, et terminant par les scopes.

4.3 Patrons de propriétés ECDL

Un patron est défini comme une description ou un modèle expliquant la façon de spécifier une propriété qui apparaît couramment dans la spécification des systèmes embarqués. En général, un patron est constitué de quatre éléments principaux [69] :

- **Nom du Patron :** Mots utilisés pour la description du problème ainsi que de ses résolutions et effets.
- **Problème :** Informations utilisées afin de déterminer quand l'application des patrons doit être envisagée, ainsi qu'une clarification de leurs objectifs et de leur contexte.

- **Solution :** Description des éléments du patron, de leurs relations, des tâches et des collaborations.
- **Conséquences :** Résultats de l'application du patrons, conséquences et compromis.

En effet, il est important de trouver un nom représentatif pour un patron car le nom améliore le vocabulaire de conception, permettant ainsi aux développeurs de communiquer à un niveau d'abstraction plus élevé. Le patron consiste en une description qui permet au lecteur de comprendre le contexte dans lequel le patron peut être appliqué. Section 4.3 montre la classification de nos patrons ECDL par rapport aux patrons de Dwyer et al. [63] ainsi que ceux de [48].

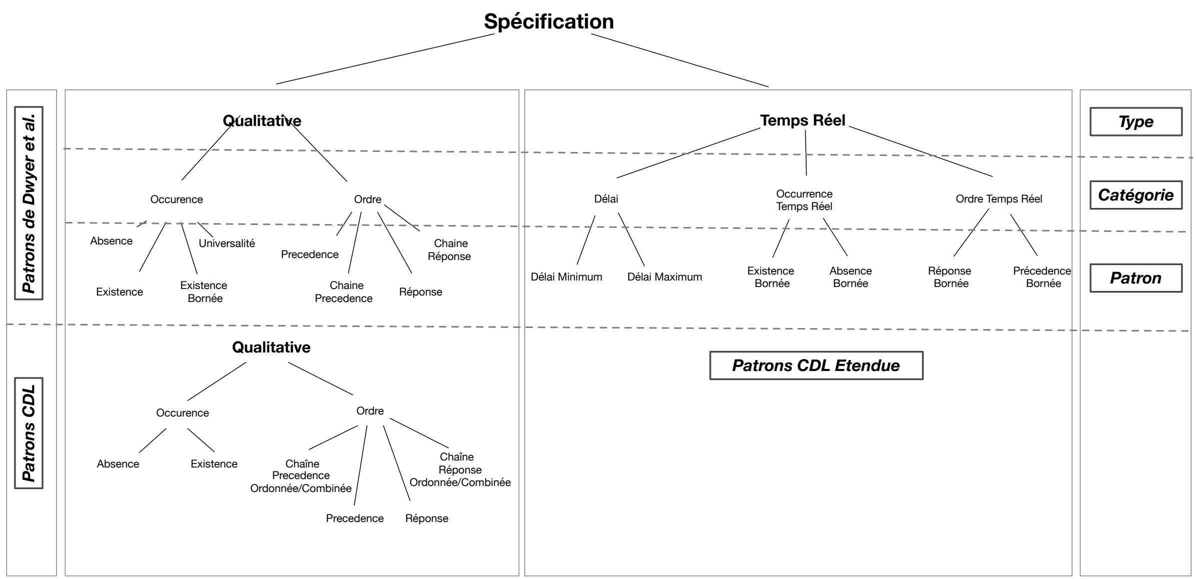


FIGURE 4.3 – Classification des patrons ECDL

4.3.1 Patrons de propriétés qualitatives

4.3.1.1 Patron de réponse

Dans certaines situations, il est possible de vouloir vérifier s'il existe des relations causales entre deux états ou événements. Une possibilité est qu'un état/événement en entraîne un autre. Cette possibilité est prise en compte par les patrons de réponse. Dans les patrons CDL, deux cas de réponses sont possibles, soit la réponse est immédiate ou elle ne l'est pas. Ces deux possibilité sont exprimées en ajoutant le mot clé "***immediately***" dans le cas où la réponse est immédiate, et "***eventually***" dans le cas échéant.

Le patron de Tableau 4.1 exige qu'une réponse soit fournie dans le futur. Une variante de ce patron consiste à exiger que la réponse soit fournie immédiatement après le stimulus. Tandis

que le patron montré dans Tableau 4.2 exige qu'une réponse soit immédiatement fournie dans l'état suivant.

Patron de propriété : *Réponse Éventuelle*
Objectif : Exprimer qu'un événement ou un état P mènera toujours, à un moment donné dans le futur, à un autre événement ou état Q.
Paramètres : P : l'événement/état qui sert de déclencheur. Q : l'événement/état qui constitue la réponse
Formulation LTL : $G(P \to FQ)$ P mènera toujours à Q .
Formulation CDL basique : P Eventually LeadsTo Q *Timebound* *Options*

TABLEAU 4.1 – Formulation de patron de réponse éventuelle

Patron de propriété : *Réponse Immédiate*
Objectif : Exprimer qu'un événement ou un état P mènera toujours, immédiatement, à un autre événement ou état Q.
Paramètres : P : l'événement/état qui sert de déclencheur. Q : l'événement/état qui constitue la réponse
Formulation LTL : $G(P \to XQ)$ P mènera toujours à Q dans l'état suivant.
Formulation CDL basique : P Immediatly LeadsTo Q *Timebound* *Options*

TABLEAU 4.2 – Formulation de patron de réponse immédiate

4.3.2 Patron de précédence

Le patron précédent représente un type de relation causale. Il existe une autre forme de relation causale dans laquelle un état/événement doit toujours précéder un autre état/événement. Le patron de précédence permet de capturer cette relation.

Patron de propriété : *Précédence*
Objectif : Exprimer qu'un événement ou un état P doit se produire avant un autre événement ou état Q. Théoriquement, ce patron est l'opposé du patron de réponse.
Paramètres : P : l'événement/état qui doit se produire en premier. Q : l'événement/état qui doit se produire en second .
Formulation LTL : $\neg Q W P$ Quel que soit le comportement du système, P se produira toujours avant que Q ne se produise.
Formulation CDL basique : P Precedes Q *Timebound* *Options*

Tableau 4.3 – Formulation de patron de précédence

4.3.3 Patron d'absence

Dans de nombreux cas, il est nécessaire de vérifier que des situations indésirables ne peuvent pas se produire. Le patron d'absence peut capturer cette exigence. C'est l'un des patrons les plus courants dans la littérature. L'exemple le plus courant est l'exclusion mutuelle.

Patron de propriété : *Absence*
Objectif : Exprimer qu'un événement ou un état P n'est pas présent tout au long de l'exécution du système.
Paramètres : P : l'événement/état qui ne faut pas qu'il se produire.
Formulation LTL : $G(\neg P)$ L'occurrence de P n'est jamais possible.
Formulation CDL basique : P occurs never *Timebound* *Options*

Tableau 4.4 – Formulation de patron d'absence

4.3.4 Patron d'existence

Dans certaines situations, il ne suffit pas d'exprimer qu'un certain événement ou état est possible, il doit être possible de manière cohérente dans tout le comportement du système.

Ce patron, bien que n'étant pas l'un des plus utilisés, a été utilisé pour exprimer des propriétés pertinentes.

Patron de propriété : *Existence*
Objectif : Exprimer qu'un événement ou un état *P* doit finir par se produire durant toute ou une partie de l'exécution du système.
Paramètres : *P* : l'événement/état qui doit se produire durant l'exécution.
Formulation LTL : *GFP* *P* va toujours se produire dans le futur.
Formulation CDL basique : *P* occurs *Timebound* *Options*

Tableau 4.5 – Formulation de patron d'existence

4.3.5 Patrons de propriétés temporelles

Les patrons de propriétés qualitatives sont conçus pour aider les utilisateurs de systèmes embarqués à décrire l'environnement du système à l'aide de diagrammes d'acteurs et de séquence, ainsi que les propriétés à vérifier [50]. Cependant, les systèmes embarqués sont omniprésents et fréquemment utilisés pour des systèmes critiques dont les fonctionnalités dépendent du temps. Il est donc nécessaire de définir un nouveau ensemble des patrons pour couvrir davantage de propriétés temps réel.

Tableau 4.6 montre une extension de l'ensemble des patrons CDL existants avec des contraintes temps réel encodant des propriétés communes rencontrées lors de la vérification de systèmes temps réel. Ce travail ne prétend pas que le système de patrons de spécification temps réel proposé soit complet. Par conséquent, il se base sur des études antérieures [93, 15, 12]. Pour favoriser l'adoption des patrons de spécification proposés, cet article représente une grammaire structurée basée sur celle définie dans [93].

4.3.6 Options ECDL

Dans le langage CDL original, au lieu de simplement paramétrer le patron en termes d'événements ou d'états, [49] ont proposé de les étendre avec des options alternatives comme dans [121] qui sont explicitement conférées au spécificateur. Le choix parmi ces options devrait aider le spécificateur à considérer les alternatives pertinentes qui sont associées au comportement attendu. Trois options ont été définis dans le catalogue des patrons de spécification CDL (Répétabilité, Précédence, Nullité). Une autre partie du patron est également considérée comme une option, à savoir l'immédiateté (**"Time Clause"**).

Cette dernière détermine si un événement, autre que ceux présentés dans les clauses *Pre* et *Post*, peut se produire immédiatement ou prochainement dans le futur avant un événement de la clause *Post*. L'immédiateté spécifie également le temps d'attente d'occurrence des propriétés

Catégorie	Nom du patron	Description
Délai Borné	Délai minimum	Décrit le temps minimum qu'une formule d'état doit tenir une fois qu'elle devient vraie. (Ex. : Système de démarrage du moteur : "Le système a une période d'arrêt minimale de 120 secondes avant de revenir en mode de démarrage").
	Délai maximum	Capture qu'une formule d'état tient toujours pendant moins d'une durée spécifiée. (Ex. : système de démarrage du moteur "Le système ne peut fonctionner en mode de démarrage du moteur que pendant 10 secondes au maximum à la fois.")
Récurrence	Absence bornée	Indique la durée pendant laquelle une action doit être absente. (Ex. : système ABS "Le contrôleur ABS vérifie l'absence de blocage des roues toutes les 10 millisecondes.")
	Existence Bornée	Décrire une partie de l'exécution d'un système qui doit contenir un événement désigné. (Ex. : Après après au plus 10 secondes de son démarrage, le système de sécurité $AN2S$, doit vérifier le fonctionnement des détecteurs)
Temps Réel	Réponse bornée	Limite le temps maximal qui s'écoule entre le moment où une formule d'état se vérifie et celui où une autre formule d'état devient vraie. (Ex. : Système ABS "A partir d'une entrée directe du client, la détection et la réponse à une décélération rapide doivent se produire dans un délai de 0,015 secondes").
	Précédence Bornée	Spécifie la durée minimale pendant laquelle une formule d'état doit être maintenue une fois qu'une autre formule d'état est satisfaite. (Ex. : Système de détection d'incendies "Si une incendie est envoyée au système d'information du système, une alarme doit de déclencher après 10 secondes.")

Tableau 4.6 – Aperçu du catalogue de patrons de spécification en temps réel

attendues. La nullité détermine si un événement e de la clause *Pre* doit se produire (e doit se produire) ou non (e ne peut jamais se produire). La précédence détermine si un événement e de la clause *Post* peut se produire (e peut se produire avant la première occurrence de (e')) ou non (où e dans Post et e' dans Pre).

Comme mentionné précédemment, dans la version étendue de CDL, de nouveaux options sont proposées qui sont : Precedency bornée, Nullity bornée et Repeatability bornée (Tableau 4.7).

Ces trois nouvelles options sont proposés avec des contraintes temporelles comme suit :

- **Precedency bornée** : Cette option modélise le cas d'un événement qui ne peut se produire que si un autre s'est produit auparavant dans un intervalle de temps donné. Elle représente la version temporisée de l'option de précédence, qui modélise simplement le fait qu'un événement peut ou ne peut pas se produire avant la première occurrence d'un autre. Cette option peut être présentée comme suit : *"si" P "doit se produire au moins une fois, alors" S "doit se produire au plus" t timeUnits "avant la première occurrence de" P*.

  Un exemple simple de cette option peut se présenter comme suit : "Lorsqu'il est armé, le système de détection d'intrusion doit traiter la perte de contact d'un capteur comme une détection positive, une alarme ne doit sonner que si une intrusion s'est produite dans les 5 ms".
- **Nullity bornée** : Cette option représente le fait qu'un événement ne doit jamais se produire pendant l'exécution d'un système donné en suivant les contraintes de temps spécifiées. Cette option est présentée comme suit : *P "ne doit pas se produire" t timeBound.*

  Un exemple simple de cette option peut être l'exigence suivante de l'application de commerce électronique : "Après une authentification erronée, une nouvelle authentification ne peut être effectuée dans les 5 secondes". (scope : après Q).
- **Repeatability limitée** : Cette option se présente comme suit : "si" *S* "tient après" *P* "fait", *P* "est tenu de suivre" S limité dans le temps.

  Voici un exemple de cette option : "Pour confirmer un numéro de téléphone, un SMS avec un code de vérification sera envoyé au numéro spécifié. Si un code erroné est saisi, un autre SMS sera envoyé après 60 secondes".

4.3.7 Définition des scopes

Dans chaque classe, les patrons génériques de [63] peuvent être spécialisés en utilisant l'un des cinq modificateurs de scopes qui limitent la plage de la trace d'exécution sur laquelle le patron doit s'appliquer :

- **Globaly** : le scope par défaut qui ne limite pas l'exécution. Le patron doit tenir sur l'ensemble de la trace temporelle ;
- **Before** R : limite le patron au début d'une plage de temps, jusqu'à la première occurrence de R ;
- **After** Q : limite le patrons aux événements qui suivent le premier R ;
- **Between** Q **and** R : limite le patron aux événements qui se produisent entre un événement Q et l'occurrence suivante d'un événement R ;
- **After** Q **until** R : similaire au scope précédent, sauf que nous n'exigeons pas que R doit nécessairement se produire après un événement Q.

4.4 Grammaire structurée

En plus du catalogue des propriétés des patrons ECDL, nous avons inclus dans notre modèle une "Spécification d'anglais structuré" [21]. Notre modèle de spécification en temps réel contient les champs suivants Tableau 4.6 :

- **Nom du patron et classification** : Le nom du patron servira à identifier l'utilisation du patron et à en décrire sa nature. La classification indique si le patron appartient à la catégorie des *ordres*, *occurrence* ou *temps réel*.
- **Spécification en anglais structuré** : La phrase en anglais structuré capture la propriété. La phrase est donnée dans une version sans scopes ni option ; un scope sera ajouté comme préfixe et les options comme suffixe à la phrase lorsque le patron sera instancié.
- **Intention du patron** : Une courte description des propriétés pour lesquelles le patron est applicable.
- **Réécriture en ECDL** : contient la syntaxe ECDL du patron.

Afin de pouvoir formuler des propriétés et des exigences de qualité en langage naturel basées sur les patrons de spécification CDL, nous avons proposé une grammaire anglaise structurée. La grammaire proposée aide les praticiens à comprendre et à spécifier correctement les propriétés et par conséquent appliquer correctement les techniques de vérification formelle. De plus, cette grammaire facilite la compréhension de la signification d'une propriété sans avoir à analyser la représentation de la logique temporelle. Cette grammaire est similaire dans son esprit à la grammaire anglaise structurée pour les patrons en temps réel [93, 15, 75], ainsi qu'à la grammaire pour la CCTL [67].

Nous définissons le patron de récurrence bornée seulement dans Tableau 4.7. Aucun observateur n'est définit pour ce patron dans cette thèse. Nous définissons seulement les quatre patrons temporelles les plus utilisés dans notre travail.

Contrairement aux grammaires précédentes, dont l'objectif principal est de supporter la spécification de propriétés exprimées en logique temporelle, la grammaire présentée ici

	$P, S, Q, R, t, t_1, t_2 \in \mathbb{R}_{\geq 0}$	States/Events
MainSpec	Property	: :=scope pattern options
Scope	(1) Scope	Globally" |"Before"R | "After"Q | "Between"Q"and"R | "After " Q " until " R
Realtime	(2) realtime (3) durationCategory (4) minDuration (5) maxDuration (6) Reccurence (7) boundedRecurrence (8) boundedExistence (9) boundedAbsence (10) OrderCategory (11) boundedResponse (12) boundedPrecedence	: := it is always the case that " (*durationCategory | periodicCategory | realtimeOrderCategory*) : :="once " P " becomes satisfied, it holds for " (minDurationPattern | maxDurationPattern " : :="at least " t " time unit(s)" : :="less than " t " time unit(s)" : := *P* " holds " (7)|(8)|(9) : := at least once in " t " time unit(s) : := one or more times timeBound : := never in " t " time unit(s) : := "if " P " holds, then " S " holds " (boundedResponsePattern | boundedPrecedencePattern) : := after at most " t " time unit(s) : := before at least " t " time unit(s)
Options	(13) boundedPrecedency (14) BoundedNullity (15) BoundedRepeatability	: := "if " P "has to occur at least once, then" S "has to hold at most" t timeUnits "before the first occurrence of" P : :="P "must not occur" timeBound" : :="if" S "hold after" P "does", P "is required to follow" S timebound
timeBound	(15) noTimeBound (16) upperTimeBound (17) lowerTimeBound (18) timeInterval (19) timeUnits	: :="" : :="within" t timeUnits : :="after at most" t timeUnits : :="between" t1 « and » t2 timeUnits : := any denomination of time (e.g. : seconds, hours, days, ...)

Tableau 4.7 – Grammaire structurée des patrons/options temporels

est destinée à supporter des propriétés particulières faciles à manipuler par des ingénieurs (non-praticiens des logiques temporelles).

La grammaire anglaise structurée est organisée selon notre classification des patrons donnée dans Section 4.3. En général, le processus d'utilisation de notre grammaire pour créer une représentation en langue naturelle est le suivant : Initialement, l'utilisateur choisit le scope de la propriété spécifiée, puis son type (qualitatif ou temps réel). Ensuite, il choisit la catégorie de la propriété spécifiée (durée, ordre périodique ou temps réel pour les propriétés en temps réel, et occurrence ou ordre pour les propriétés qualitatives). La phrase anglaise structurée finale est construite en choisissant le patron de spécification correspondant. Dans la grammaire, "Preceds" et "Leads To" dénotent les mot clés necessaire à la définition d'un patron 'Précédence' et 'Réponse'. Tableau 4.8 montre la grammaire structurée des patrons qualitatifs (CDL classique). Nous précisons que la notion de *scope* dans les patrons qualitatifs est définit comme un prédicat(*strong/weak*). Lorsque le prédicat est faible, le patron peut ne pas le respecter. Par contre, si le prédicat est fort, il faut qu'il soit respecter par le patron de la propriété.

	$P, S, Q, R, t, t_1, t_2 \in \mathbb{R}_{\geq 0}$	States/Events
MainSpec	Property	: :=scope pattern options
Scope	(1) Predicate	: :=strong | weak
pattern	(2) Response	: :="if P holds, then as a response S *Time Clause* holds"
	(3) Precedence	: :="P must always be preceded by S"
	(4) Absence	: :="P must never occur"
	(5) Existence	: :="P must hold"
TimeBound	(6) TimeInterval	: := $[t, t'] \mid [t, t'[\mid]t, t'[\mid]t, t']$
Options	(7) Precedency	: :="S can (or can't) occur before the first occurrence of P"
	(8) Nullity	: :="S may (may never) occur"
	(10) Repeatability	: :="The behavior is repeatable"

Tableau 4.8 – Grammaire structurée des patrons qualitatifs

Par exemple, l'utilisateur souhaite créer une représentation en langage naturel pour une propriété de réponse bornée. Initialement, l'utilisateur dérive la phrase suivante de la grammaire (les règles de grammaire sont données entre parenthèses) : "Globalement, il est toujours le cas que si P tient, alors S tient après au plus t unité(s) de temps". (Grammaire : 1, 2, 3, 10, 11, 17, 19).

Les terminaux non littéraux doivent être instanciés pour la représentation du langage naturel. P et S dénotent des formules booléennes propositionnelles qui décrivent des propriétés d'états et sont utilisées pour capturer des propriétés du système ainsi que pour dénoter des états qui servent de limites pour les scopes. Pour les propriétés en temps réel, c doit également être instancié avec les valeurs entières qui sont utilisées avec les opérateurs contraints dans une formule de logique temporelle en temps réel. Après avoir remplacé P par $(a = 0)$, S par $(b = 1)$ et t par 6, on obtient la représentation finale en langage naturel de la propriété susmentionnée :

"Globalement, il est toujours le cas que si ($a = 0$) se réalise, alors ($b = 1$) se réalise après au plus 6 unité(s) de temps".

4.5 Intérêts des patrons de propriétés ECDL

Dans cette section, une comparaison entre le systémes de patrons CDL étendue (ECDL) et le CDL original est effectuée.

Propriété	ECDL	CDL
P1 : "The system shall simulate calling emergency services by calling a configurable alternate number when running diagnostics every 5 minutes". (Reccurence bornée Tableau 4.7(7))	**Between** diagnostics_initiate **and** diagnostics_terminate **AN** [one occurence of emergency_call_simulation] **at_least** every 5 mn	Exprimer cette propriété avec les patrons CDL classique est un défi, car ils ne supportent ni l'utilisation des scopes temporells (Between Q and R) ni la récurrence bornée.
P2 : "it is always the case that : if the alarm is activated than an intrusion is detected before the alarm activation for at least t-time". (Précédence bornée)	**Globally** **AN** [Each occurrence of intrusion detection] **Preceds** **AN** [one or more occurrence of Alarm activation] **For at least** 30s	Même si le scope temporel évident **Globally** est prise en compte par le CDL, la propriété de précédence bornée n'est toujours pas couverte, ce qui rend très difficile la représentation d'une telle exigence dans la CDL originale.
P3 : "it is always the case that : if fire sensor is triggered than the system will alert the control center after at most t-time." (Bounded response)	**Globally** **AN** [Each occurrence of fire detection] **Leads_to** **AN** [one occurrence of control center alert] **after at most** t-time	**AN** [Exactly one occurrence of fire detection] **Leads_to** [0..t-time] **AN** [one occurrence of control center alert]

Tableau 4.9 – Comparaison de CDL classique et ECDL

Nous démontrons comment utiliser les patrons de spécification en temps réel et comment convertir les exigences informelles des systèmes embarqués en spécifications formelles, sur la base de la grammaire donnée Tableau 4.7. ECDL prend en compte des spécifications en temps réel et les spécifications en langage naturel de propriétés qui ne pouvaient pas être spécifiées en termes de patrons de CDL originaux. Une différence sémantique majeure entre les patrons étendus et les originaux est la manière dont les informations en temps réel sont incluses dans leurs formules. Alors que CDL construit un intervalle avec un opérateur limité dans le temps et désigne les états qui peuvent se produire dans cet intervalle, ECDL donne trois

façons de représenter la limite temporelle (supérieure, inférieure et intervalle, voir Tableau 4.7). Par conséquent, notre système de patrons ECDL est considérée comme plus expressive que l'originale en ayant plus de patrons de propriétés temporelles qui ne sont pas définies dans CDL, par exemple, la réponse bornée et la récurrence bornée.

Pour illustrer l'efficacité de ECDL, nous proposons à titre d'exemple un certain nombre d'exigences sensibles au temps pour le système "AN2S" qui peuvent être bien présentées par ECDL mais pas avec CDL.

La première comparaison dans Tableau 4.9 montre une expressivité significative des patrons ECDL par rapport au CDL. Dans le premier exemple (P1), la propriété de récurrence bornée reste difficile à représenter en CDL en raison du manque de clarté concernant la définition des scopes et de la récurrence temporelle. Quant au deuxième exemple (P2), puisque le scope "Globalement" indique que l'exécution du patron doit se maintenir sur l'ensemble de l'exécution du programme, l'exigence peut être présentée dans les deux langages. Cependant, la propriété de réponse temporelle ne fait pas partie des patrons CDL, ce qui rend difficile l'expression d'une telle exigence. Bien que la dernière exigence (P3) puisse être spécifiée dans les deux langages, la clarté de la grammaire de ECDL améliore la compréhension et l'utilisation du patron par rapport à la CDL. Cependant, un catalogue de patrons est un travail en cours et ECDL ne fait pas exception. Ce travail ne prétend pas que le système de patrons de spécification en temps réel proposé est complet.

4.6 Conclusion

Ce chapitre propose un ensemble étendu de patrons couramment utilisés pour spécifier la correction des systèmes temps réel complexes. Le principal avantage de ce travail est qu'il aide les ingénieurs non experts en méthodes formelles et en logiques temporelles à spécifier les propriétés permettant de vérifier la correction des systèmes. Il fournit une interface en langage naturel sous la forme d'une grammaire anglaise structurée. Comme le système sera plus efficace si les observateurs peuvent être générés automatiquement, nous avons développé un outil d'assistance pour cette tâche Chapitre 6. L'exactitude d'une telle traduction doit être vérifiée, à cette fin, nous avons choisi d'utiliser un outil de preuve de théorème qui est l'assistant de preuve Coq qui semblait être une solution puissante Chapitre 7.

Une autre direction future de cette recherche est l'extension de la bibliothèque proposée de patrons CDL par l'étude d'un nombre pertinent de spécifications du monde réel. Si nécessaire, le système de patrons sera mis à jour à la suite de cette étude.

La conception de patrons de correction pour les extensions probabilistes des systèmes temps réel fait également l'objet de travaux futurs.

CHAPITRE

5

Patrons comme observateurs

« Energy and persistence conquer all things. »

— Benjamin Franklin

Sommaire

5.1 Introduction **66**
5.2 Observateurs **66**
5.3 Patrons observateurs **67**
5.3.1 Non-accessibilité 67
5.3.2 Patron de précédence 68
5.3.3 Patron de réponse 72
5.3.4 Existence bornée 75
5.3.5 Précédence bornée 76
5.3.6 Réponse bornée 79
5.4 Conclusion **83**

5.1 Introduction

Dans ce chapitre, nous proposons pour chaque patron une syntaxe aussi lisible que possible par l'homme, afin que des ingénieurs non experts en méthodes formelles puissent les utiliser. De plus, nous montrons comment les traduire en propriétés d'accessibilité en utilisant des observateurs simples, c'est-à-dire des sous-systèmes supplémentaires qui observent certaines actions du système et peuvent également utiliser le temps. Alors qu'une classe de patrons doit être traduite en observateurs "must-reach" (c'est-à-dire pour lesquels un bon état doit être atteint à chaque exécution "***succes***"), tous les autres patrons peuvent être traduits en observateurs de non-accessibilité (c'est-à-dire que la propriété est correcte si un mauvais état donné "***reject***" n'est jamais accessible). Par conséquent, leur vérification en pratique évite l'utilisation d'algorithmes de vérification complexes ou d'outils dédiés, et les développeurs d'outils peuvent les mettre en œuvre à peu de frais. Afin de montrer l'applicabilité de notre approche ; ces observateurs peuvent alors être utilisés par OBP comme par des outils qui ne gèrent que l'analyse d'accessibilité. Les contributions de ce chapitre sont résumées ci-dessous :

1. Nous proposons une syntaxe abstraite pour chaque patron.
2. Nous traduisons chaque patron en un observateur.

5.2 Observateurs

Nous proposons ici une définition des observateurs dans les automates temporisés. Les observateurs sont des sous-systèmes standards, avec certaines conditions et exceptions. Un observateur ne doit pas avoir d'effet sur le système, et ne doit pas empêcher un comportement de se produire. En particulier, il ne doit pas bloquer le temps, ni empêcher les actions de se produire, ni créer des blocages qui ne se produiraient pas autrement.

Par conséquent, les observateurs doivent être complets : dans le cas des automates temporisés, toutes les actions déclarées par l'observateur doivent être autorisées dans n'importe quel état.

Dans ce qui suit, nous distinguons le modèle original (c'est-à-dire le modèle à vérifier) du modèle global (c'est-à-dire le modèle original en plus de l'observateur).

Dans le formalisme des automates temporisés, un observateur est un automate temporisé standard (voir la définition Chapitre 3 avec certaines restrictions) :

- l'observateur utilise (au maximum) une horloge locale x_{obs} (le cas avec plus d'une horloge locale pourrait être possible, mais n'est pas utilisé dans ce travail), et aucune horloge partagée ;
- l'observateur peut avoir un état spécial mauvais (noté par "***reject***"), qui n'a aucune transition sortante ;

- l'observateur peut contenir un ensemble de bons états (notés "***success***").

Le modèle global est défini comme suit : $A_1 \parallel ... \parallel A_n \parallel A_{obs}$, où $A_1 ... A_n$ sont les automates temporisés modélisant le modèle original, et A_{obs} est l'automate observateur. La composition parallèle garantit que les actions partagées par l'observateur et le modèle original seront synchronisées.

5.3 Patrons observateurs

Dans cette section, nous présentons la bibliothèque de patrons ECDL[21], que nous traduisons en observateurs, de sorte que des outils simples sans capacités complexes de model-checking puissent les vérifier. Dans ce qui suit, nous faisons l'hypothèse que l'outil est capable de vérifier deux types de propriétés. La première propriété est la propriété de non-accessibilité, c'est-à-dire qu'un état "*reject*" donné n'est pas atteignable, dans aucune des exécutions possibles. Nous supposons la syntaxe abstraite suivante inspiré de la proposition de [12] pour la commande de vérification :

$$assert\ unreachable(Reject)$$

La deuxième propriété est une propriété d'accessibilité qui exige que chaque exécution se termine dans un bon état ("*Success*"). (Si le chemin d'exécution est fini, alors la notion de "fin" fait référence au dernier état ; sinon, à partir d'un certain point, il boucle avec un cycle ; nous exigeons que tous ses états soient "*Success*"). Nous supposons la syntaxe abstraite suivante :

$$assert\ alwaysEndsWith(Success)$$

Notons que les deux propriétés font référence à des propriétés qui doivent être vraies (ou fausses) pour toutes les exécutions. Dans ce qui suit, les définitions des états bons ou mauvais dans les automates temporisés ne feront référence qu'aux états des automates ; pour les observateurs, elles feront donc référence à leur(s) bon(s) ou mauvais état(s) (*Reject, Success*). Dans la suite de cette section, nous proposons 8 patrons et 6 options organisés en 2 classes. Chaque patron sera présenté avec une simple syntaxe et une description dans deux langues : Français (***FR*** en abrégé) et Anglais (***EN*** en abrégé). Avant de présenter les 8 patrons principaux de ECDL, il est important de présenter la propriété de non-accessibilité.

5.3.1 Non-accessibilité

La propriété de non-accessibilité d'un certain mauvais état est de loin la propriété la plus courante utilisée pour caractériser la correction des systèmes temps réel. Cette propriété est en quelque sorte dégénérée puisque, pour les outils de vérification capables de vérifier

nativement la non-accessibilité d'un mauvais état, cette propriété ne nécessite pas l'utilisation d'un observateur. Néanmoins, nous la présentons ici car c'est la propriété la plus courante. Étant donné une définition d'état mauvais "***Reject***", le patron de non-accessibilité peut être décrit comme suit :

Syntaxe Abstraite : assert unreachable(Reject).
Description : ***FR*** : "L'état "*Reject*" n'est jamais accessible". ***EN*** : "The state "*Reject*" never happens".

Tableau 5.1 – Patron de Non-Accessibilité

La littérature est pleine de exemples de propriétés de non-accessibilité. Parmi les études de cas les plus courantes, la correction du protocole d'exclusion mutuelle de *Fischer* (une version temporisée est présentée par exemple dans [11]). L'exclusion mutuelle est généralement considérée comme une propriété de non-accessibilité : le mauvais état est défini comme un état où plus d'un processus se trouve dans la section critique. De même, dans le problème du passage des trains (voir un exemple simple dans [11]), le mauvais état correspond à un état où le train traverse la route bien que la barrière soit encore ouverte.

5.3.2 Patron de précédence

Cette classe de patrons modélise le cas d'une action qui ne peut se produire que si une autre action s'est déjà produite auparavant. Un exemple typique est le cas où l'on veut éviter les fausses alarmes dans le système *AN2S* à titre d'exemple : une alarme ne doit se déclencher que si une action donnée (par exemple, détection de feu ou intrusion) s'est déjà produite auparavant. (Notons que cette propriété ne signifie pas que l'intrusion ou le feu conduira toujours à une alarme; cela fera l'objet des patron "Réponse" dans Section 5.3.6). Nous considérons deux patrons dans cette classe, chacun avec trois variants différents. La vérification de leur exactitude se réduit toujours à une propriété de non-accessibilité.

Ces propriétés de précédence qualitatifs sont toutes non temporelles, pour cette raison, les observateurs correspondants ne contient aucune notion de temps.

5.3.2.1 Précédence AN

Le variant AN signifie que seulement une action doit se produire. Nous considérons seulement le variant de la première action qui doit se produire (celle qui précède) car peu importe les actions qui suivent, le résultat reste le même. Trois versions différentes de ce patrons peuvent être déduites en se basant sur l'option de répétabilité ("True", "False") et l'expression qui précise le nombre d'occurrences de la première action (*Each, One or more*) .

1. ***Version Acyclique*** : Dans ce patron, nous vérifions donc que si une action a_2 (ou une liste d'action "AllOrdered/AllCombined" $a_2 ... a_n$) se produit, alors l'action a_1 s'est produite (au moins une fois) avant la première occurrence de a_2 (ou la première occurrence de première action si la liste est ordonnée (AllOrdered) ou la première occurrence de toutes les actions sinon (AllCombined)). La description de ce patron est comme suit :

Syntaxe Abstraite : Si a_2 Alors a_1 s'est produit avant.
Description : ***FR*** : "Si a_2 se produit au moins une fois, alors a_1 s'est produit avant la première occurrence de a_2". ***EN*** : "if a_2 happens at least once, then a_1 has happened before the first occurrence of a_2".

Tableau 5.2 – Patron de Précédence AN

Rappelons que ce patron ne nécessite pas que a_2 se produise du tout, même si a_1 s'est produit. Nous donnons l'observateur correspondant sur Figure 5.1(a). L'observateur est simple : si a_2 se produit en premier, l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. Si a_1 se produit, l'observateur entre dans un autre emplacement (s_1), et y reste pour toujours. Ce patron peut être vérifier par un outil de model-checking utilisant la commande suivante :

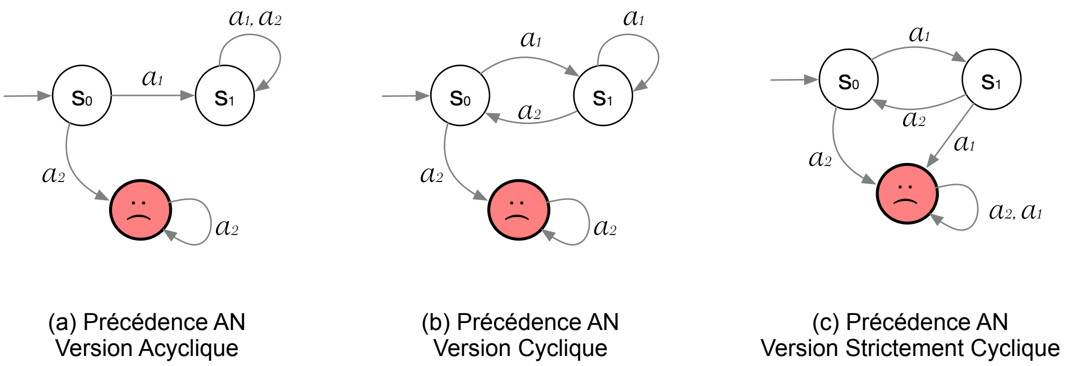
$$assert\ unreachable(state[observer] = reject),$$

Où nous supposons que "$state[observer]$" désigne l'état actuel de l'automate observateur. En d'autres termes, le système satisfait cette propriété si le mauvais état de l'observateur n'est jamais atteint.

2. ***Version cyclique*** : Dans cette version, nous vérifions que si une action a_2 se produit, alors l'action a_1 s'est produite (au moins une fois) depuis la dernière occurrence de a_2, et ainsi de suite de manière cyclique. Encore une fois, notez que cette version n'exige pas que a_2 se produise du tout, et ne garantit pas que a_2 se produira un nombre infini de fois. En outre, a_1 peut se produire plusieurs fois (au moins une fois) entre deux occurrences de a_2. Cela peut être contrôler avec le mot clé "One or more" ou bien "Each".

Syntaxe Abstraite : Chaque fois que a_2 se produit, a_1 s'est produit avant.
Description : ***FR*** : "Chaque fois que a_2 se produit, signifie que a_1 s'est produit auparavant, depuis la dernière occurrence de a_2 (s'il y en a une)". ***EN*** : "Every time a2 happens, then a_1 has happened before, since the last occurrence of a_2 (if any)".

Tableau 5.3 – Précédence AN version cyclique



(a) Précédence AN
Version Acyclique

(b) Précédence AN
Version Cyclique

(c) Précédence AN
Version Strictement Cyclique

FIGURE 5.1 – Automate Observateur pour le patron de “Précédence” AN

Nous montrons l'observateur correspondant à cette version dans Figure 5.1(b). Si a_2 se produit en premier, l'observateur entre dans son mauvais emplacement et y reste. Ensuite, tant qu'au moins une occurrence de a_1 se produit entre deux occurrences de a_2, l'observateur n'entre pas dans le mauvais état ("reject").

3. ***Version strictement cyclique*** : Dans ce patron, nous vérifions que si une action a_2 se produit, alors l'action a_1 s'est produite exactement une fois depuis la dernière occurrence de a_2, et ainsi de suite de manière cyclique. Comme la version précédente, cette version ne nécessite pas que a_2 se produise du tout, et ne garantit pas que a_2 se produira un nombre infini de fois. En d'autres termes, ce patron exige que a_1 et a_2 alternent, en partant de a_1.

Syntaxe Abstraite : Chaque fois que a_2 se produit, a_1 s'est produit avant.
Description : ***FR*** : "Chaque fois que a_2 se produit, signifie que a_1 s'est produit auparavant, exactement une fois depuis la dernière occurrence de a_2 (s'il y en a une)". ***EN*** : "Every time a2 happens, then a_1 has happened before, exactly once since the last occurrence of a_2 (if any)".

TABLEAU 5.4 – Précédence AN version strictement cyclique

L'observateur correspondant est illustré par Figure 5.1(c). Si a_2 se produit en premier, l'observateur entre dans son mauvais emplacement et y reste. Ensuite, a_1 et a_2 alternent ; sinon, l'observateur entre dans l'état de "reject".

5.3.2.2 Précédence AllOrdered version acyclique

Le variant AllOrdered signifie qu'une chaîne d'actions peuvent se produire suivant leurs ordre dans la chaîne. Identique au premier patron, nous considérons la chaîne d'action qui se produit en premier.

Soit : `AllOrdered` a_1, a_2 `Occur before (`a_3`)` un exemple de patron de précedence version AllOrdered. Dans ce patron, nous vérifions donc que si une action a_3 se produit, alors la dernière action a_2 de la chaîne d'actions ordonnées a_1, a_2 s'est produite (au moins une fois) avant la première occurrence de a_3. La description de ce patron est :

Syntaxe Abstraite : Si a_3 Alors toute la chaîne AllOrdered a_1, a_2 s'est produit avant.
Description : ***FR*** : "Si a_3 se produit au moins une fois, alors a_2 s'est produit avant la première occurrence de a_3". ***EN*** : "if a_3 happens at least once, then a_2 has happened before the first occurrence of a_3".

Tableau 5.5 – Patron de précédence AllOrdered version acyclique

Nous donnons l'observateur correspondant sur Figure B.2(a). Si a_3 se produit en premier (avant la dernière action de la chaîne ordonnée, dans notre exemple a_2), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. Dés que a_2 se produit, l'observateur entre dans un autre emplacement (s_2), et a_3 peut se produire.

5.3.2.3 Précédence AllCombined version acyclique

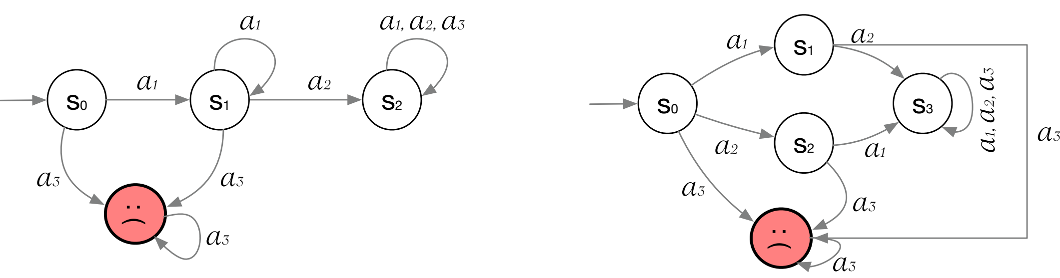
Le mot clé *AllCombined* signifie qu'une chaîne d'actions peuvent se produire suivant toutes les combinaisons possibles de l'ordre des actions.

Soit : `AllCombined` a_1, a_2 `Occur before (`a_3`)` un exemple de patron de précedence version AllCombined avec trois actions. Dans ce patron, nous vérifions donc que si une action a_3 se produit, alors au moins l'une des chaîne $\{a_1,\ a_2\}$, $\{a_2,\ a_1\}$ s'est produite (au moins une fois) avant la première occurrence de a_3. La description de ce patron est :

Syntaxe Abstraite : Si a_3 Alors $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ s'est produite avant.
Description : ***FR*** : "Si a_3 se produit au moins une fois, alors $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ s'est produit avant la première occurrence de a_3". ***EN*** : "if a_3 happens at least once, then either $\{a_1,\ a_2\}$ or $\{a_2,\ a_1\}$ has happened before the first occurrence of a_3".

Tableau 5.6 – Patron de précédence AllCombined version acyclique

L'observateur correspondant à ce patron est montré sur Figure B.2(b). Si a_3 se produit en premier (avant l'occurrence de toute la chaîne des actions $\{a_1, a_2\}$), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours.



(a) Précedence AllOrdered version Acyclique

(b) Précedence AllCombined version Acyclique

Figure 5.2 – Automates observateurs pour le patron “Précédence” AllOrdered et AllCombined

5.3.3 Patron de réponse

Cette classe de propriétés considère le cas d’une action toujours suivie d’une autre. Cette classe de propriétés est souvent appelée ***"liveness"*** ; cependant, ce mot a une sémantique différente, et même les experts en model-checking ne sont pas tous d’accord sur sa signification [3]. Cette classe de patrons est la seule dans ce travail à ne pas être basée sur la non-accessibilité ; à la place, on doit vérifier que chaque exécution se termine dans un bon état "*success*". Encore une fois, puisque ces propriétés sont toutes non temporelles, les observateurs n’utiliseront aucune fonction de temps.

5.3.3.1 Réponse AN :

1. **Version acyclique** : La notion acyclique signifie que l’option *Repeatability* de ce patron est définie comme "*False*". Dans ce patron, nous vérifions que si une action a_1 se produit, alors l’action a_2 finit par se produire.

Syntaxe Abstraite : Si a_1 alors a_2.
Description : ***FR*** : "Si a_1 se produit, alors a_2 finit par se produire". ***EN*** : "if a_1 happens, then a_2 eventually happens".

Tableau 5.7 – Patron de Réponse AN version Acyclique

Notons que ce patron ne nécessite pas que a_1 se produise. De plus, a_1 peut se produire plusieurs fois avant que a_2 ne se produise. Ceci est réalisé en utilisant le mot clé "*One or more*" avant l’action a_1. Nous montrons l’observateur d’automate correspondant en Figure 5.3(a). L’observateur commence à un bon endroit ; si a_1 ne se produit jamais, il reste dans ce bon endroit. Lorsque a_1 se produit, l’observateur entre dans un emplacement intermédiaire qui

3. Voir : `https://cs.nyu.edu/acsys/beyond-safety/liveness.htm`

n'est pas bon s_1 ; ce n'est que lorsque a_2 se produit que l'observateur entre dans le deuxième bon emplacement, et il y restera pour toujours.

2. **Version cyclique** : Dans cette version, l'option *Repeatability* est définie comme "True", cela signifie que le comportement est répété. En d'autres termes, nous vérifions qu'à chaque fois qu'une action a_1 se produit, alors l'action a_2 finit par se produire. L'observateur correspondant est illustré par Figure 5.3(b).

Syntaxe Abstraite : À chaque fois a_1 alors a_2 finit par se produire.
Description : ***FR*** : "Chaque fois que a_1 se produit, alors a_2 finit par se produire". ***EN*** : "Every time a_1 happens, then a_2 eventually happens".

TABLEAU 5.8 – Patron de réponse AN version cyclique

3. **Version strictement cyclique** : Dans cette version, l'option répétabilité est définie comme "True", en plus, nous précisons que a_1 doit se produire seulement une seule fois en utilisant l'expression $Exactly\ one\ occurence\ of\ a_1$. Dans ce patron, nous vérifions que chaque fois qu'une action a_1 se produit, l'action a_2 finit par se produire.

Syntaxe Abstraite : chaque fois a_1, alors a_2 finit par se produire une fois avant la prochaine a_1.
Description : ***FR*** : "Chaque fois que a_1 se produit, alors a_2 se produit dans le futur exactement une fois avant la prochaine occurrence de a_1". ***EN*** : "Every time a_1 happens, then a_2 eventually happens exactly once before the next occurrence of a_1".

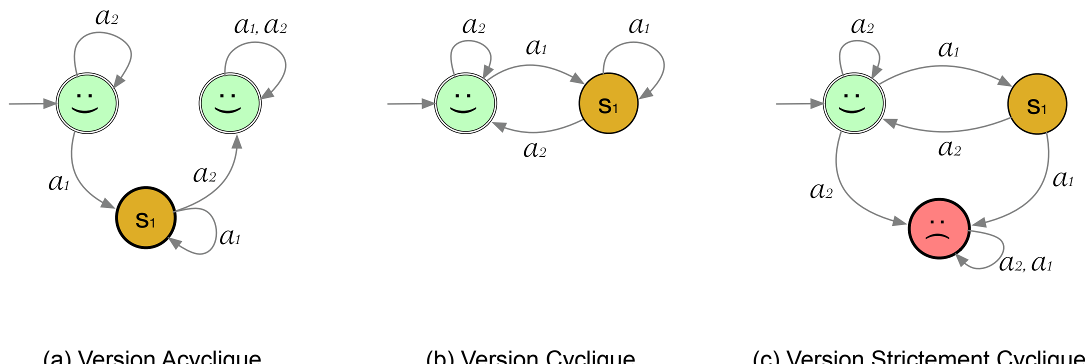
TABLEAU 5.9 – Patron de réponse AN version strictement cyclique

Dans cette version (strictement cyclique), a_1 et a_2 alternent, en commençant par a_1 et, chaque fois que a_1 apparaît, a_2 doit également apparaître. Nous montrons l'observateur correspondant dans Figure 5.3(c). Cet observateur utilise à la fois les mauvais et les bons états ("reject", "success"). Cependant, il suffit de vérifier que le système se termine dans un état de "success".

5.3.3.2 Réponse AllOrdered version acyclique

Soit : `AllOrdered` a_1, a_2 `Leads to (`a_3`)` un exemple de patron de réponse AllOrdered. Dans ce patron, nous vérifions donc que si la chaîne d'actions a_1, a_2 se produit, alors en réponse, l'action a_3 se produit. La description de ce patron est donnée par Tableau 5.10.

Nous donnons l'observateur correspondant sur Figure 5.4(a). Si a_2 se produit en premier (avant a_1), l'observateur entre dans un mauvais état. Si l'ordre est respecté, et a_2 se produit après



(a) Version Acyclique (b) Version Cyclique (c) Version Strictement Cyclique

FIGURE 5.3 – Automate observateur pour le patron de "Réponse" AN

Syntaxe Abstraite : Si a_1, a_2 alors a_3 finit par se produire.
Description : ***FR*** : "Si a_1 se produit suivie par a_2, alors a_3 finit par se produire". ***EN*** : "if a_1 happens followed by a_2, then a_3 eventually happens".

TABLEAU 5.10 – Patron de réponse AllOrdered version Acyclique

a_1, l'observateur entre dans un état intermédiaire qui n'est pas bon s_2 en attendant l'arrivée du a_3. Dés que a_3 arrive, l'observateur entre dans un état bon "succes".

5.3.3.3 Réponse AllCombined version acyclique

Soit : `AllCombined` a_1, a_2 `Leads to (`a_3`)` un exemple de patron de réponse AllCombined avec trois actions. Dans ce patron, nous vérifions donc que si au moins une des chaines $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ se produit, alors a_3 finit par se produire. La description de ce patron est :

Syntaxe Abstraite : Si au moins $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ alors a_3.
Description : ***FR*** : "Si au moins $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ se produit au moins une fois, alors a_3 se produira". ***EN*** : "if at least one of the orderd lists $\{a_1,\ a_2\}$ or $\{a_2,\ a_1\}$ happens at least once, then a_3 eventually happens".

TABLEAU 5.11 – Patron de réponse AllCombined version Acyclique

L'observateur correspondant à ce patron est montré sur Figure 5.4(b). Si au moins une des chaines d'actions attendues se produit, l'observateur entre dans un état intermédiaire pas bon s_3. Après l'arrivée du a_3, l'observateur entre dans un état de "success".

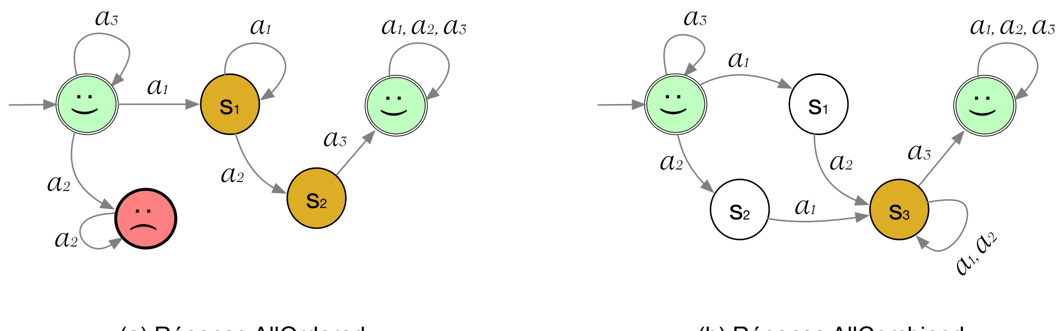


FIGURE 5.4 – Automates Observateurs pour le patron “Réponse” AllOrdered et AllCombined

5.3.4 Existence bornée

Ce patron modélise le cas d'une action qui doit se produire en respectant un délai donné après le démarrage du système. Dans cette section, nous présentons le patron avec $lowerTimeBound$ qui signifie que la condition de temps a respecter est : pas plus tard que d unité de temps ("*after at most d*").

Syntaxe Abstraite : a au plus tard d.
Description : ***FR*** : "a se produira après au plus tard d unités de temps après le démarrage du système". ***EN*** : "a will happen after at most d units of time after the system start".

TABLEAU 5.12 – Patron existence bornée version cyclique

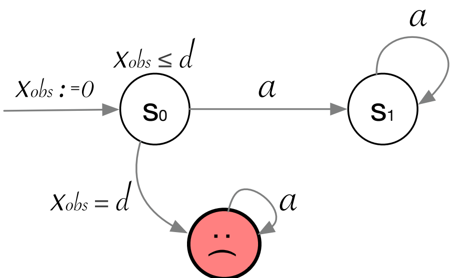


FIGURE 5.5 – Automate observateur pour le patron “Existence Bornée”

Nous illustrons l'observateur temporisé correspondant dans Figure 5.5. L'horloge de l'observateur x_{obs} est initialement fixée à 0. Ensuite, si a se produit avant d unités de temps (modélisé par l'invariant $x_{obs} \leq d$ de s_0), il entre dans s_1 où il restera pour toujours. Mais si a ne se produit pas avant le temps requis (d unités de temps), l'observateur entre dans le mauvais état ("reject").

5.3.5 Précédence bornée

Cette classe de patrons modélise le cas d'une action qui ne peut se produire que si une autre s'est produite dans un intervalle de temps donné auparavant. Il s'agit d'une extension temporelle de la classe de patrons "Précédence" que nous avons défini en *ECDL*. Pour reprendre l'exemple des alarmes de *AN2S*, une alarme ne doit sonner que si une intrusion a eu lieu dans les 5 secondes précédentes. (Encore une fois, cette propriété ne signifie pas que l'intrusion entraînera toujours une alarme dans un intervalle de temps donné ; ce sera le sujet des patrons "Réponse bornée" dans Section 5.3.6).

5.3.5.1 Précédence bornée AN

1. **Version acyclique** : Dans cette version, nous vérifions que si une action a_2 se produit, alors l'action a_1 s'est produite au moins une fois avant la première occurrence de a_2 dans les d dernières unités de temps.

Syntaxe Abstraite : Si a_2 alors a_1 s'est produit au plus d unités de temps avant.
Description : ***FR*** : "si a_2 se produit au moins une fois, alors a_1 s'est produit au maximum d unités de temps avant la première occurrence de a_2". ***EN*** : "If a_2 happens at least once, then a_1 has happened at most d units of time before the first occurrence of a_2".

Tableau 5.13 – Patron de précédence bornée AN version acyclique

   Nous donnons l'observateur temporisé correspondant sur Figure 5.6(a). La transition de s_0 à $reject$ modélise le fait que a_2 ne peut pas se produire en premier. Ensuite, lorsque a_1 se produit, l'horloge x_{obs} est initialisée. Le reste de l'automate est similaire au patron "Existence bornée" (Figure 5.5).

2. **Version cyclique** : Dans cette version, nous vérifions qu'à chaque fois qu'une action a_2 se produit, l'action a_1 s'est produite (au moins une fois) avant la dernière occurrence de a_2.

   Nous noterons que ce patron ne nécessite pas que a_2 se produise, même si a_1 le fait. L'observateur de l'automate temporisé correspondant est présenté à Figure 5.6(b). La transition de s_0 à $reject$ modélise le fait que a2 ne peut pas se produire en premier. Ensuite, chaque fois que a1 se produit, l'horloge x_{obs} est initialisée. Si a_2 se produit au plus d unités de temps plus tard (invariant $x_{obs} \leq d$) alors le système continue. Sinon ($x_{obs} \geq d$), l'observateur entre dans le mauvais état "reject".

3. **Version strictement cyclique** : Dans ce patron, nous vérifions qu'à chaque fois qu'une action a_2 se produit, l'action a_1 s'est produite exactement une fois avant la dernière occurrence de a_2 (Dans ECDL nous utilions le mot clé "*exactly one occurence of*"), et au maximum d unités de temps depuis cette dernière occurrence.

Syntaxe Abstraite :
A chaque fois a_2 alors a_1 s'est produit au plus d unités de temps avant.
Description : ***FR*** : "Chaque fois que a_2 se produit, alors a_1 s'est produit au maximum d d'unités de temps auparavant, et au moins une fois depuis la dernière occurrence de a_2 (si elle existe)". ***EN*** : "Every time a_2 happens, then a_1 has happened at most d units of time before, and at least once since the latest occurrence of a_2 (if any)".

Tableau 5.14 – Patron de précédence bornée AN version cyclique

Syntaxe Abstraite :
A chaque fois a_2 alors a_1 s'est produit exactement une fois au plus d unités de temps.
Description : ***FR*** : "Chaque fois que a_2 se produit, alors a_1 s'est produit au maximum d d'unités de temps auparavant, et exactement une fois depuis la dernière occurrence de a_2 (si elle existe)". ***EN*** : "Every time a_2 happens, then a_1 has happened at most d units of time before, and exactly once since the latest occurrence of a_2 (if any)".

Tableau 5.15 – Patron de précédence bornée AN version strictement cyclique

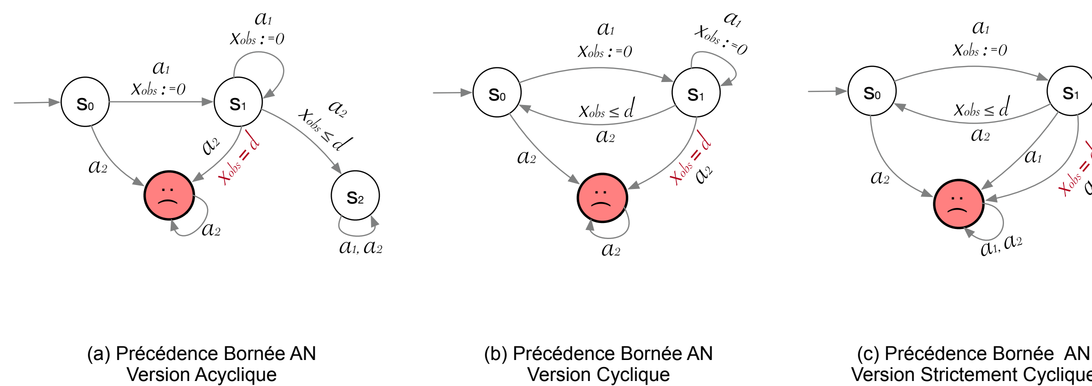


Figure 5.6 – Automates observateurs pour le patron “Précédence Bornée AN”

En d'autres termes, a_1 et a_2 alternent, en commençant par a_1, et avec au maximum d unités de temps entre a_1 et a_2. Nous montrons l'observateur correspondant dans Figure 5.6(c). Ce patron est identique à celui de Figure 5.6(b), à la différence que a_1 ne peut pas se produire deux fois dans une rangée, ce qui explique la transition (étiquetée avec a_1) de s_1 à $reject$.

5.3.5.2 Précédence bornée AllOrdered version acyclique

Soit : `AllOrdered` a_1, a_2 `Occur before (`a_3`) within` d `time units`, un exemple de patron de précedence bornée version AllOrdered. Dans ce patron, nous vérifions donc que si une action a_3 se produit, alors la dernière action a_2 de la chaîne d'actions ordonnées a_1, a_2 s'est produit (au moins une fois) avant la première occurrence de a_3 dans les d dernières unités de temps . La description de ce patron est :

Syntaxe Abstraite : Si a_3 Alors toute la chaîne AllOrdered a_1, a_2 s'est produit au plus d unités de temps avant.
Description : ***FR*** : "Si a_3 se produit au moins une fois, alors a_2 s'est produite au maximum d unités de temps avant la première occurrence de a_3". ***EN*** : "if a_3 happens at least once, then a_2 has happened at most d units of time before the first occurrence of a_3".

Tableau 5.16 – Patron de précédence bornée AllOrdered version acyclique

Nous donnons l'observateur correspondant sur Figure 5.7(a). Si a_3 se produit en premier (avant la dernière action de la chaîne ordonnée, dans notre exemple a_2), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. Dés que a_2 se produit, l'horloge de l'observateur s'initialise et a_3 peut se produire dans le temps exigé ($x_{obs} \leq d$) ainsi l'observateur passe à l'état suivant s_2. Sinon, ($x_{obs} \geq d$), l'observateur entre dans son mauvais état et y reste pour toujours.

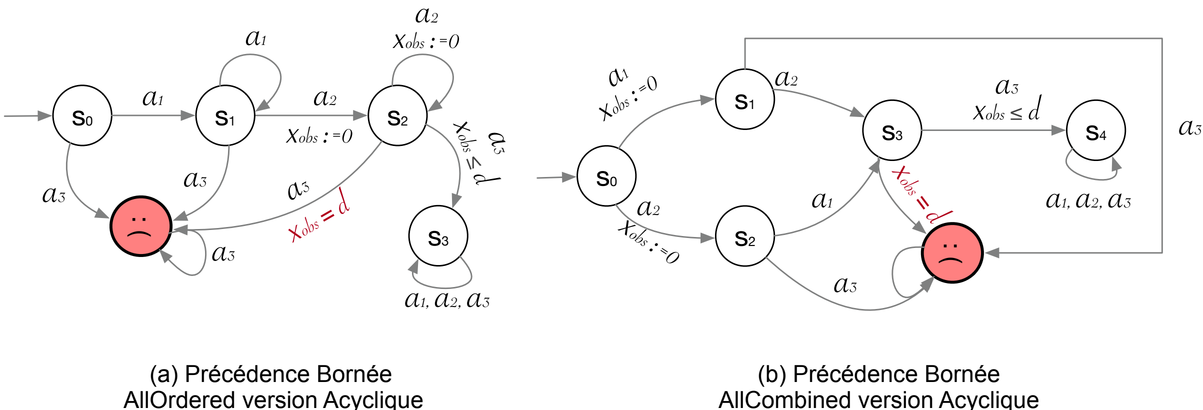
5.3.5.3 Précédence bornée AllCombined version acyclique

Soit : `AllCombined` a_1, a_2 `Occur before (`a_3`) within` d `time units`, un exemple de patron de précédence bornée version AllCombined avec trois actions. Dans ce patron, nous vérifions donc que si une action a_3 se produit, alors au moins une des chaînes d'actions $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produite (au moins une fois) avant la première occurrence de a_3 dans les d dernières unités de temps.

L'observateur correspondant à ce patron est montré sur Figure 5.7(b). Si a_3 se produit en premier (avant l'une des chaînes $\{a_1, a_2\}$ ou $\{a_2, a_1\}$), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. x_{obs} est réinitialisé à chaque fois que l'une des actions a_1 ou

Syntaxe Abstraite : Si a_3 Alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produite avant.
Description : ***FR*** : "Si a_3 se produit au moins une fois, alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit avant la première occurrence de a_3 au plus d unités de temps". ***EN*** : "if a_3 happens at least once, then either $\{a_1, a_2\}$ or $\{a_2, a_1\}$ has happened at most d units of time before the first occurrence of a_3".

Tableau 5.17 – Patron de précédence bornée AllCombined version acyclique



(a) Précédence Bornée AllOrdered version Acyclique

(b) Précédence Bornée AllCombined version Acyclique

Figure 5.7 – Automates observateurs pour le patron “Précédence bornée” AllOrdered et AllCombined

a_2 se produit pour la premiére fois. Si a_3 arrive avant d unités de temps, l'observateur passe à l'état suivant s_4. Par contre, si le temps est écoulé avant l'arrivé de a_3 ($x_{obs} \geq d$), l'observateur passe au mauvais état de "reject".

5.3.6 Réponse bornée

Cette classe de propriétés considère le cas d'une action suivie d'une autre dans un certain intervalle de temps. Elle peut être considérée comme une extension temporelle du patron de "Réponse". Là encore, le nom de ce patron fait l'objet de débats. Il peut exister en littérature sous le nom "***Vivacité borné*** ou ***Bounded Liveness*** en anglais" [24].

5.3.6.1 Réponse AN

1. **Version acyclique** : Cette version signifie que l'option *Repeatability* de ce patron est définie comme "*False*".

   Dans ce patron, nous modélisons que si une action a_1 se produit, alors l'action a_2 se produira dans d unités de temps. Ce patron ne nécessite pas que a_1 se produise. De plus, a_1 peut se

produire plusieurs fois avant que a_2 ne se produise. Ceci est réalisé en utilisant le mot clé "*One or more*" avant l'action a_1. Nous montrons l'observateur d'automate correspondant en Figure 5.8(a).

Syntaxe Abstraite : Si a_1 alors a_2 dans un délai de d unités de temps.
Description : ***FR*** : "Si a_1 se produit, alors a_2 finit par se produire dans un délai de d unités de temps". ***EN*** : "if a_1 happens, then a_2 will eventually happens within d units of time".

TABLEAU 5.18 – Patron de Réponse bornée AN version Acyclique

Au début, a_2 peut se produire à tout moment. Puis, une fois que a_1 se produit, l'horloge x_{obs} est initialisée. Si a_2 se produit dans un délai de d unités de temps (garanti par l'invariant $x_{obs} \leq d$), alors le système entre dans un état suivant s_2, et a_1 et a_2 peuvent tous deux se produire à tout moment. Sinon, après d unités de temps, le système entre dans le mauvais état "*reject*", et y reste.

2. **Version cyclique** : Dans cette version, l'option *repeatability* est définie comme "***True***", ce qui signifie que le comportement est répétable. En d'autres termes, nous vérifions qu'à chaque fois une action a_1 se produit, alors l'action a_2 finit par se produire dans d unités de temps. L'observateur correspondant est illustré par Figure 5.8(b).

Syntaxe Abstraite : À chaque fois a_1 alors a_2 dans d unités de temps.
Description : ***FR*** : "Chaque fois que a_1 se produit, alors a_2 finit par se produire dans un délai de d unités de temps, et avant la prochaine occurrence de a_1 (s'il y en a une)". ***EN*** : "Every time a_1 happens, then a_2 will eventually happen within d units of time, and before the next occurrence of a_1 (if any)".

TABLEAU 5.19 – Patron de Réponse Bornée AN version cyclique

L'observateur correspondant est similaire à celui de la version acyclique Figure 5.8(a) sauf que la transition portant l'étiquette a_2 revient vers s_0 au lieu de s_2.

3. **Version strictement cyclique** : Dans cette version, l'option répétabilité est définie comme "True", en plus, nous précisons que a_1 doit se produire seulement une seule fois en utilisant l'expression $exactly\ one\ occurence\ of$ a_1. Dans ce patron, nous vérifions que chaque fois qu'une action a_1 se produit, l'action a_2 finit par se produire .

Dans cette version, a_1 et a_2 alternent, en commençant par a_1 et, chaque fois que a_1 apparaît, a_2

Syntaxe Abstraite : chaque fois a_1, alors a_2 finit par se produire une fois avant la prochaine a_1 dans d unités de temps.
Description : ***FR*** : "Chaque fois que a_1 se produit, alors a_2 se produit dans d unités de temps, exactement une fois, avant la prochaine occurrence de a_1" . ***EN*** : "Every time a_1 happens, then a_2 within d units of time, exactly once before the next occurrence of a_1".

TABLEAU 5.20 – Patron de Réponse bornée AN version strictement cyclique

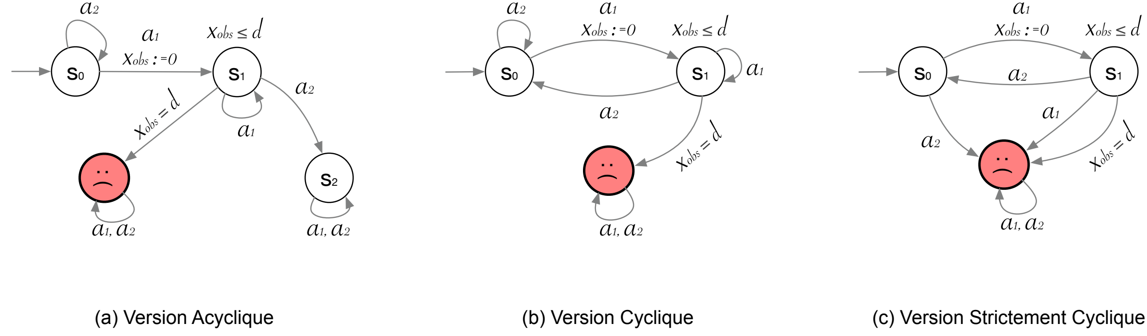


FIGURE 5.8 – Automate observateur pour le patron de “Réponse Bornée” AN

doit également se produire dans un délai de d unités de temps. Nous montrons l'observateur correspondant dans Figure 5.8(c).

L'observateur correspondant est très proche du version cyclique Figure 5.8(b) avec l'ajout de deux transitions de s_1 vers $reject$ et de s_0 vers $reject$ portant l'étiquette a_1.

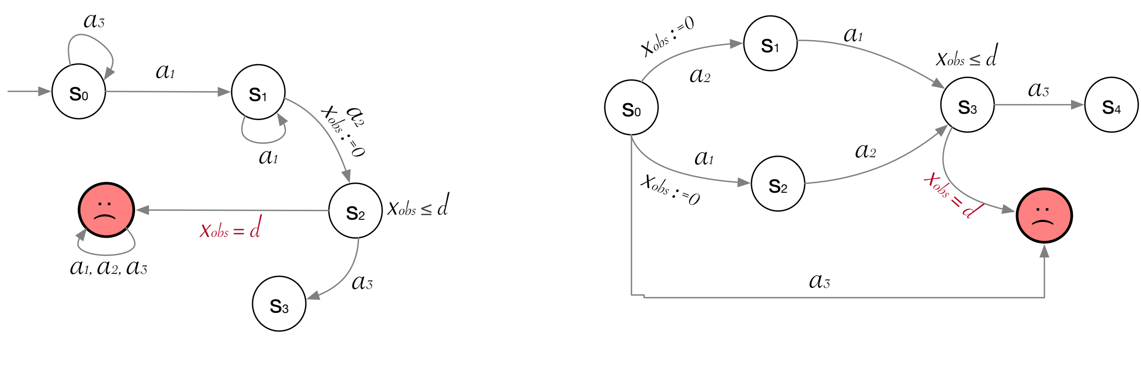
5.3.6.2 Réponse AllOrdered version acyclique

Soit : `AllOrdered` a_1, a_2 `Leads to` (a_3) `within` d `time units`, un exemple de patron de réponse bornée AllOrdered. Dans ce patron, nous vérifions donc que si la chaîne d'actions a_1, a_2 se produit, alors en réponse, l'action a_3 se produit dans d unités de temps.

Syntaxe Abstraite : Si a_1, a_2 alors a_3 dans d unités de temps.
Description : ***FR*** : "Si a_1 se produit suivie par a_2, alors a_3 se produit dans d unités de temps". ***EN*** : "if a_1 happens followed by a_2, then a_3 eventually happens within d time units".

TABLEAU 5.21 – Patron de Réponse bornée AllOrdered version Acyclique

Nous donnons l'observateur correspondant sur Figure 5.9(a). Si l'ordre est respecté, et a_2 se produit après a_1, l'horloge s'initialise et l'observateur commence le compte à rebours pour l'arrivée du a_3. Si a_3 arrive avant que la fin du temps prévu, l'observateur passe à l'état suivant. Dans le cas échéant, l'observateur passe à son mauvais état ("*reject*") et y reste pour toujours.



(a) Réponse Bornée AllOrdered

(b) Réponse Bornée AllCombined

FIGURE 5.9 – Automates Observateurs pour le patron "Réponse" AllOrdered et AllCombined

5.3.6.3 Réponse AllCombined version acyclique

Soit : `AllCombined` a_1, a_2 `Leads to (`a_3`) within` d `time units`, un exemple de patron de réponse AllCombined avec trois actions.

Dans ce patron, nous vérifions donc que si au moins une des chaînes $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ se produit, alors a_3 se produit dans un délai de d unités de temps. L'observateur correspondant à ce patron est montré sur Figure 5.9(b). La description de ce patron est :

Syntaxe Abstraite : Si au moins $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ alors a_3 dans d unités de temps.
Description : ***FR*** : "Si au moins $\{a_1,\ a_2\}$ ou $\{a_2,\ a_1\}$ se produit au moins une fois, alors a_3 se produit dans d unités de temps". ***EN*** : "if at least one of $\{a_1,\ a_2\}$ or $\{a_2,\ a_1\}$ happens at least once, then a_3 eventually happens within d time units".

TABLEAU 5.22 – Patron de Réponse Bornée AllCombined version Acyclique

Remarque. *Les versions cycliques et strictement cycliques des automates observateurs correspondants au patrons AllOrdered et AllCombined sont présentées en Annexe B.*

5.4 Conclusion

Nous proposons ici un ensemble de patrons couramment utilisés pour spécifier la correction des systèmes temps réel complexes. Les principaux avantages sont les suivants. Premièrement, l'ingénieur non-expert en méthodes formelles peut facilement spécifier la correction du système à partir d'une bibliothèque de propriétés communes exprimées dans un langage intuitif. Deuxièmement, le développeur d'outils peut facilement vérifier ces propriétés en se basant sur l'algorithme de (non-)accessibilité. Ce travail est une extension de langage CDL ainsi que des patrons proposé en [12].

Jusqu'à présent, seuls les patrons exprimant des propriétés qui doivent être vraies (ou fausses) pour toutes les exécutions (scope *Globally*) ont été pris en compte dans ce chapitre, car ils sont de loin les plus courants dans l'exemple industriel que nous avons considéré. L'extension de notre ensemble de patrons peut se faire suivant notre approche compositionnelle proposé en Chapitre 6.

Contrairement aux patrons de [12], nos patrons peuvent être composés sans problème théorique grâce à notre approche compositionnelle [20].

CHAPITRE

6

Transformation des patrons

« Knowledge is power. »

— Francis Bacon

Sommaire

6.1 Introduction 86

6.2 Formalisation des patrons ECDL sans options et scopes 86

6.2.1 Détermination des états 87

6.2.2 Génération des transitions 89

6.2.3 Règles de combinaison de clauses *Pre* et *Post* 89

6.3 Le framework ECDL 92

6.4 Approche compositionnelle 94

6.4.1 Exemple de motivation 94

6.4.2 Formalisation des patrons ECDL 95

6.5 Composition de patrons et options 98

6.5.1 Règles de composition de patrons et options 98

6.5.2 L'opération de composition 99

6.6 Composition des patrons et scopes 103

6.7 Conclusion 104

6.1 Introduction

Un patron CDL s'écrit comme une combinaison de propriété et options. Il est transformé en un automate observateur visant à être vérifier à l'aide de l'outil OBP, qui génère des schémas d'exécution qui révèlent le chemin d'état rejeté, ce qui indique que la propriété vérifiée n'est pas valide.

La transformation est réalisée respectant un algorithme implémenté en langage Java [51], il contient des fonctions de transformation pour chaque combinaison patron/options. Cette transformation a deux problèmes majeurs :

- Explosion combinatoire : les transformations deviennent plus compliquées lorsque nous avons une combinaison de plusieurs propriétés et options à exprimer.
- Validation de transformations : Les fonctions de transformations n'ont pas été valider.

Afin de remédier à ces problème, dans ECDL nous avons proposé une approche compositionnelle en reprenant l'idée développée par [126] qui proposent d'exprimer la sémantique des patrons par une composition d'automates de Büchi décrivant séparément les patrons et les scopes. Nous présentons dans ce chapitre les règles de composition entre les patrons de base et les options.

6.2 Formalisation des patrons ECDL sans options et scopes

Afin de simplifier le processus de transformation, nous proposons de décomposer le patron en trois parties principales qui sont : les scopes, les options et le corps du patrons comme mentionner en Figure 6.7. Dans cette section, nous proposons de traduire le patron (seulement le corps sans options ni scopes) par partie et ensuite composer les parties transformées en automates observateurs ensemble. Pour cette raison, nous proposons une partition de corps de patron en trois parties principales comme montré dans Figure 6.1.

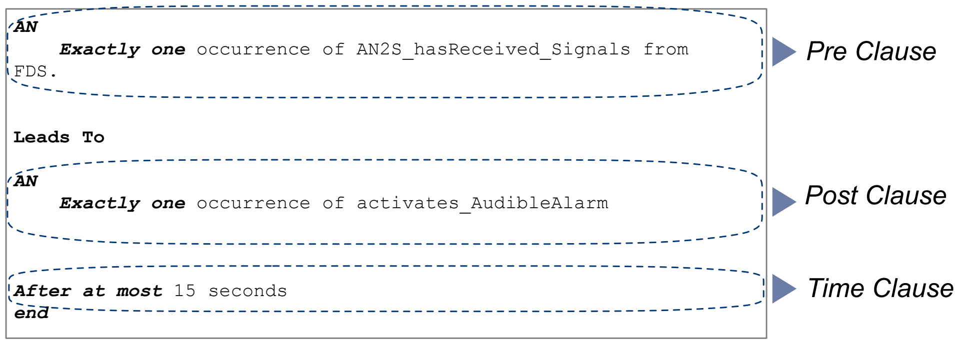


Figure 6.1 – Éléments d'un patron de réponse ECDL (exemple P1)

Pour notre transformation, nous avons choisi le patron de propriété de réponse bornée. Pour accomplir son transformation, nous devons traduire chacune des parties ***Pre Clause*** et ***Post Clause*** en un système de transition auquel nos règles de composition (définies dans Section 6.5) seront appliquées pour obtenir un automate correspondant à un patron sans options.

Ce processus de transformation se compose de trois étapes importantes qui sont :

1. Définition des états de l'observateur : l'ensemble des états qui constituent l'observateur résultant est obtenu en appliquant l'union des états des parties *Pre Clause* et *Post Clause* obtenus selon Algorithme 6.1.
2. Définition de l'ensemble des transitions reliant les états de chaque partie (Algorithme 6.2).
3. Combinaison des automates correspondants aux parties Pre et Post : consiste en l'ajout d'une transition que nous appelons "*Compositionnelle*" qui relie les deux parties ensemble pour obtenir l'observateur final (sans options ou scopes). De plus, les conditions relatives à l'horloge définies par la partie *"Time clause"* sont ajoutées.

6.2.1 Détermination des états

Dans cette section, nous allons détaillé l'étape de détermination des états qui constituent les systèmes de transitions correspondants à la Pre et Post Clause.

- $Pr = (Init_{Pr}, S_{Pr}, F_{Pr}, E_{Pr}, T_{Pr})$ représente le système d'état correspondant au *Pre Clause.*, et,
- $Ps = (Init_{Ps}, S_{Ps}, F_{Ps}, E_{Ps}, T_{Ps})$ est le système représentant *Post Clause,* d'où :

  $Init_{Pr}$: est l'état initial de la clause Pre, et, $Init_{Ps}$ représente l'état initial de la clause $Post$, et définit l'état final de la clause Pre.

  Nous utilisons une variable nommée *index* qui peut être substituée par l'une des valeurs (Pr, Ps) représentant respectivement les Clauses *Pre* et *Post.*

  S_{index} : Ensemble finis d'états de la partie $index$.

  E_{index} : l'ensemble des événements qui seront déclenchés dans la clause $index$.

  f_{index} : représente l'état final de la clause $index$, c'est-à-dire l'état final à atteindre dans cette partie qui pourra être différent de l'état final de l'observateur (rejet/succès).

  $T_{index} : S_{index} \times E_{index} \times S_{index}$ est une fonction de transition.

> **Remarque.** *Pour les patrons réponse, et précédence avec leurs différents variants, nous notons que l'état final de la clause* Pre *représente l'état initial de la clause* $Post$*.* *(*$Init_{Ps} = f_{Pr}$*) et vice versa (*$f_{Ps} = Init_{Pr}$*).*

Algorithme 6.1 détaille le processus de détermination des états. Ce processus se base sur le type de l'occurrence des évènements nommé "*arityType*" et qui a les valeurs (AN, AllOrdered,

AllCombined). Pour le type "*AN*" seulement deux états de base sont nécessaires quelque soit le nombre d'évènements contrairement aux types *AllOrdered* et *AllCombined* dont le nombre d'états dépend de nombre d'évènements.

Algorithme 6.1 Détermination des états

Inputs Ev, Ev_{ps} : *Event list*
$ExpOccLoc$: $ExpressionOccurenceLocation$
$arityType$: $ExpressionOccurenceType$

1 : **Initialisation** : $ExpressionOccLoc := \{Pre, Post\}$, $arityType := \{AN, AllOrdered, AllCombined\}$
2 : **function** ENUMERATESTATES_PR($Ev, arityType$) ▷ Cette fonction génère tous les états de la Pre Clause.
3 : $S_{Pr} \leftarrow \emptyset$
4 : **si** $arityType = AN$ **alors**
5 : $j \leftarrow 2$
6 : **sinon si** $arityType = AllOrdered$ **alors**
7 : $j \leftarrow l + 1$ ▷ l : est la longueur de la liste des évènements de la Pre Clause
8 : **sinon**
9 : $j \leftarrow (l * (l - 1) + 2)$
10 : **fin si**
11 : **pour** $k \leftarrow 0$, $k < j$ **faire**
12 : $S_{Pr} \leftarrow S_{Pr} \cup \{S_k\}$
13 : **fin pour**
14 : **retourner** S_{Pr}
15 : **fin function**

16 : **function** ENUMERATESTATES_PS($Ev, arityType$) ▷ Cette fonction génère tous les états de la Post Clause.
17 : $S_{Ps} \leftarrow \emptyset$
18 : **si** $arityType = An$ **alors**
19 : $j \leftarrow 1$
20 : **sinon si** $arityType = AllOrdered$ **alors**
21 : $j \leftarrow l$ ▷ l : est la longueur de la liste des évènements de la Post Clause
22 : **sinon**
23 : $j \leftarrow (l * (l - 1) + 1)$
24 : **fin si**
25 : **pour** $P \leftarrow K$, $(j + k$ **faire**
26 : $S_{ps} \leftarrow S_{ps} \cup \{S_p\}$
27 : **fin pour**
28 : **retourner** S_{ps}
29 : **fin function**

6.2.2 Génération des transitions

L'ensemble des transitions reliants les différents états des parties Pre et Post sont générées selon Algorithme 6.2. Cet algorithme fait appel aux fonctions *EnumerateStates_Pr* et *EnumerateStates_Ps* de Algorithme 6.1 pour récupérer l'ensemble des états.

Algorithme 6.2 Génération des transitions

Input *Evt_List_pre, Evt_List_post : Event list*
Input S_{pre}, S_{Post} *: States list*
1 : **Call** EnumerateStates_index ▹ La fonction EnumerateStates_index de Algorithme 6.1 est appelée pour avoir les états S_{index}
2 : **Initialisation** : $S_{pre} := S_{pr}, S_{Post} := S_{ps}$
3 : **function** TRANSITIONDEF_PRE(Evt_List_pre, S_{pre}) ▹ Cette fonction est utilisée pour générer les transitions de la Pre Clause
4 : **pour** $j \leftarrow 0$, $j \leq S_{pre}_length$ **faire**
5 : $List_T_pre \leftarrow List_T_pre \cup \langle S_j, Lb_j, Next_pr(S_j)\rangle$
6 : **fin pour**
7 : **retourner** *List_T_pre*
8 : **fin function**

---

9 : **function** TRANSITIONDEF_POST(Evt_List_post, S_{post}) ▹ Cette fonction est utilisée pour générer les transitions de la Post Clause
10 : **pour** $j \leftarrow 0$, $j \leq length(S_{post})$ **faire**
11 : $List_T_post \leftarrow List_T_pre \cup \langle S_j, Lb_j, Next_ps(S_j)\rangle$
12 : **fin pour**
13 : **retourner** *List_T_post*
14 : **fin function**

6.2.3 Règles de combinaison de clauses *Pre* et *Post*

Nous avons défini les règles suivantes afin de réaliser la génération d'un automate de patron ECDL sans option/scopes. Cette étape sert à composer les deux parties *Pre Clause* et *Post Clause*. Initialement, les deux sous-observateurs correspondant à ces parties sont définis comme montré dans Section 6.2.1 ($Pr = (Init_{Pr}, S_{Pr}, F_{Pr}, E_{Pr}, T_{Pr})$ et $Ps = (Init_{Ps}, S_{Ps}, F_{Ps}, E_{Ps}, T_{Ps})$).

Chaque patron ECDL noté Pt est représenté par un automate observateur comme suit : $Pt = (Init_{Pt}, S_{Pt}, F_{Pt}, E_{Pt}, T_{Pt})$. Cet observateur est le résultat de combinaison des automates des Clauses Pre et Post suivant les règles ci-dessous :

- **Règle 1**. $Init_{Pt} = Init_{Pr}$
- **Règle 2.** $S_{Pt} = S_{Pr} \cup S_{Ps}$

- **Règle 3.** $E_{Pt} = E_{Pr} \cup E_{Ps}$

- **Règle 4.** $PatternTransitions = \begin{cases} \dfrac{s \xrightarrow{e} s' \in T_{Pr}}{s \xrightarrow{e} s' \in T_{Pt}} (*) \\ \dfrac{s \xrightarrow{e} s' \in T_{Ps}}{s \xrightarrow{e} s' \in T_{Pt}} \\ \dfrac{X_{obs} \sim delay = true}{s \xrightarrow{X_{obs} \sim delay} reject \in T_{Pt}} [s = f_{pr}] (**) \end{cases}$

- **Règle 5.** $f_{Pt} = reject$

Les règles précédentes représentent les étapes de construction que nous avons suivies pour obtenir un automate observateur correspondant à un patron CDL sans options ni scopes.

> **Remarque.**
>
> (*) $\frac{s \xrightarrow{e} s' \in T_{Pr}}{s \xrightarrow{e} s' \in T_{Pt}}$ *Cette Règle indique que : si une transition $s \xrightarrow{e} s'$ appartienne à l'ensemble de transition de la clause Pre implique que cette transition doit aussi appartenir à l'ensemble de transitions de patron T_{Pt} résultant de la composition.*
>
> (**) *Cette transition représente la propriété de temps (Time Clause), le symbole $\sim$ peut avoir une valeur $(\leq, <, \geq, >, =)$ et $delay$ représente le temps à respecter.*

L'algorithme de construction qui génère un observateur correspondant à un patron sans options est donné par Algorithme 6.3. Comme nous l'indiquons dans les commentaires, chaque

Algorithme 6.3 Combinaison de clause Pre et Post

Inputs $Pr = (Init_{Pr}, S_{Pr}, F_{Pr}, E_{Pr}, T_{Pr})$
$Ps = (Init_{Ps}, S_{Ps}, F_{Ps}, E_{Ps}, T_{Ps})$
$Ev = Event\ list$

```
1 : function COMBINE_PATTERN_PARTS(pre, post)
2 :     Init ← Init_pr                          ▷ Cela est représentée par Règle 1
3 :     S ← S_pr ∪ S_ps                         ▷ Obtenu par Règle 2
4 :     F ← F_ps                                ▷ Règle 5
5 :     E ← E_pr ∪ E_ps ∪ (ck > delay)          ▷ Représente la Règle 3
6 :     T ← patternTransition(T_pr, T_ps)       ▷ Règle 4
7 :     retourner P = ⟨Init, S, F, E, T⟩
8 : fin function
```

instruction correspond à une règle spécifique (parmi l'ensemble des règles **1** à **5**).

La Figure 6.2 montre les différents automates observateurs correspondant aux patrons de réponse bornée ***version cyclique*** (AN-AN, AllOrderd-AN,...) obtenus en appliquant les règles de construction (de 1 à 5). Dans Figure 6.2, la première ligne représente un patron de réponse avec *AN* comme action, une réponse donnée respectivement avec l'un des opérateurs (*AN/AllOrdered/Allcombined*). La deuxième ligne correspond au patron de réponse avec une action précédé d'un opérateur *AllOrdered* indiquant que les événements de cette partie doivent être exécuter selon leurs ordre d'arrivée. La réponse peut être précéder par l'un des opérateurs (*AN/AllOrdered/Allcombined*). La dernière ligne représente le patron de Réponse avec AllCombined comme action et l'une des réponses (AN/ AllOrdered/Allcombined).

Nous illustrons l'application des règles de construction précédentes (de 1 à 5), en les appliquant au patron de propriété *P*1 montré en Figure 6.1. Figure 6.3 montre les trois parties du patron de réponse correspondant à la propriété *P*1 précédemment traitée et les observateurs correspondants.

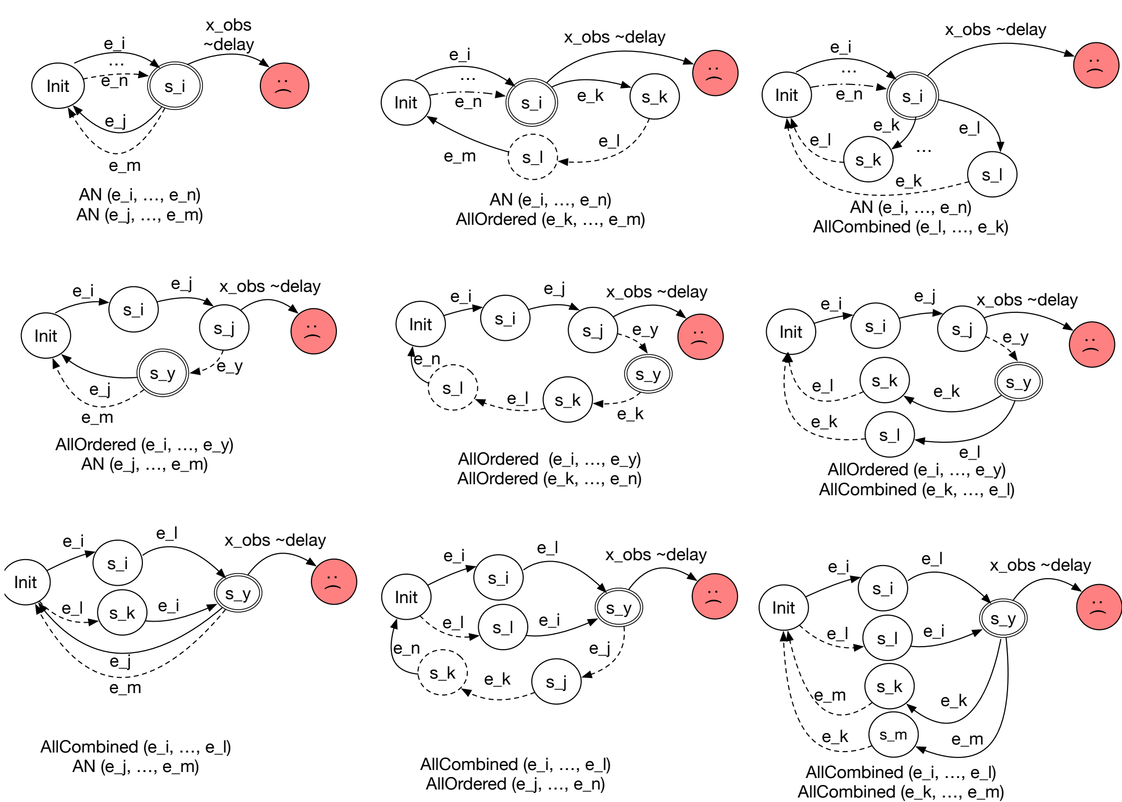


FIGURE 6.2 – Observateur de patron de réponse bornée

Nous notons que les règles de transformation montrées dans ce chapitre ne sont pas complets,

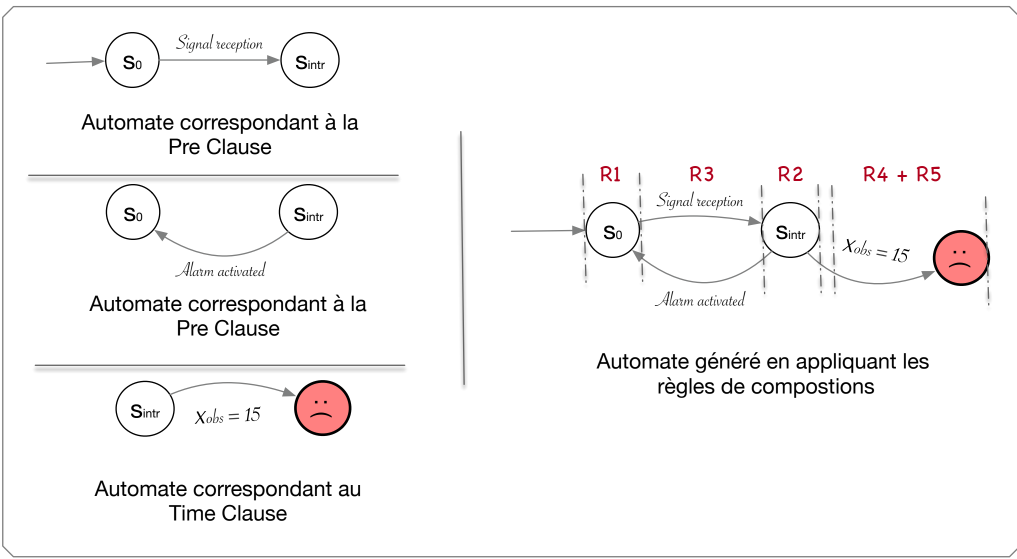


FIGURE 6.3 – Observateur sans options

nous avons seulement montré une version simplifié d'un seul patron (Réponse bornée). Toutes les règles nécessaires à la transformations des autres patrons sont définies avec le même principe et implémentées à l'aide de langage de programmation *Kotlin* [4].

6.3 Le framework ECDL

Comme nous l'avons déjà évoqué, l'expression formelle des propriétés temporelle peut être une tâche complexe. Bien que les patrons ECDL décrits ci-dessus puissent être utiles pour faire face à cette complexité, le processus manuel de sélection et d'instanciation d'un patron est susceptible de produire des erreurs qui peuvent être difficiles à détecter. C'est particulièrement le cas lorsque des formules complexes sont en jeu.

Pour remédier à ce problème, nous avons développé un outil d'aide à l'expression des patrons ECDL. Non seulement cet outil permet d'écrire des patrons suivant la syntaxe et la grammaire structurée que nous avons proposé, mais aussi transforme les patrons en automates observateurs qui peuvent être vérifier par la suite par des outils de model-checking.

L'outil comprend toutes les informations contenues dans les patrons ECDL, telles que le type des occurrences (AN, ALLOrdred, ALLCombined), ainsi que les mots clés (ExactlyOne, ...) et offre une auto complétion. Cela permet à l'utilisateur de parcourir les options afin de sélectionner celui qui convient le mieux à la propriété en question sans erreurs syntaxiques. En outre, l'outil transforme le patron entré en un automate observateur correspondant suivant nos règles de transformations implémentées. L'observateur est affiché en utilisant le langage DOT en deux formats comme montré dans Figure 6.4 à droite, format écrite et format graphe.

4. https ://kotlinlang.org/

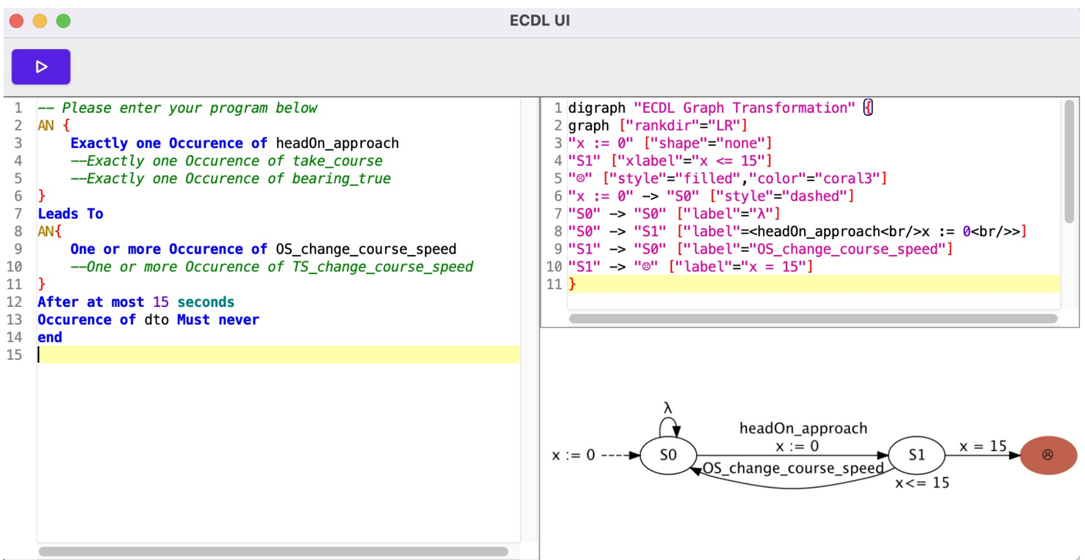


Figure 6.4 – L'outil de transformation ECDL

L'interface utilisateur de cet outil est créé en utilisant un framework appelé *Jetpack Compose Desktop* [5]. Ce framework propose une approche déclarative et réactive de la création d'interfaces utilisateurs de bureau avec *Kotlin.* Aucun XML ou langage de template n'est nécessaire.

Nous avons exécuté un exemple de patron de réponse ($P1$) dans l'outil et nous avons eu le résultat suivant.

```
digraph "ECDL Graph Transformation" {
graph ["rankdir"="LR"]
"x  := 0" ["shape"="none"]
"S1" ["xlabel"="x <= 15"]
" :(" ["style"="filled","color"="coral3"]
"x  := 0" -> "S0" ["style"="dashed"]
"S0" -> "S0" ["label"="λ"]
"S0" -> "S1" ["label"=<headOn_approach<br/>x  := 0<br/>>]
"S1" -> "S0" ["label"="OS_change_course_speed"]
"S1" -> " :(" ["label"="x = 15"]
}
```

L'avantage de ce format écrit est de pouvoir modifier l'automate seulement en modifiant ce format. On peut facilement changer la couleur des états, le format des transitions, leurs couleurs, ect. Le graphe correspondant est montré dans Figure 6.5.

Dans les sections précédentes de ce chapitre, nous avons discuté la transformation d'un patron sans options ou scopes. Afin d'ajouter les options et les scopes, nous allons présenté en se qui suit une approche compositionnelle permettant de composer le résultat de transformation

5. Jetpack Compose Desktop : https ://www.jetbrains.com/fr-fr/lp/compose-mpp/

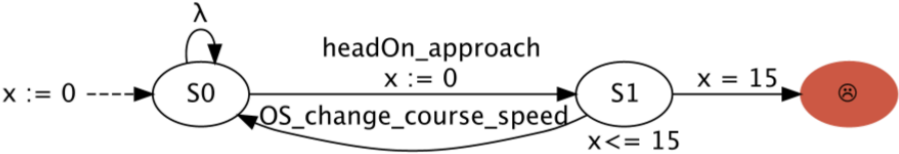


Figure 6.5 – Automate correspondant à P1 généré par l'outil ECDL

de patron présenté précédemment avec multiple options et/ou scopes.

6.4 Approche compositionnelle

Dans cette section, nous proposons une extension complémentaire des patrons ECDL qui permet de simplifier la sémantique de leurs expressions. Cette proposition a pour but aussi de :

1. Simplifier la transformation des patrons ECDL en automates observateurs.
2. Composition d'un patron de propriété ECDL avec différentes options d'une manière constructif relativement simple.
3. Composition des patrons ECDL avec différents Scopes s'inspirant de l'approche de [126].
4. Composition des différents patrons en utilisant les opérateurs logiques pour exprimer des propriétés plus complexes [134].

Afin de résoudre les problèmes d'explosion combinatoire relatifs à la complexité de certaines propriétés, nous proposons une sémantique compositionnelle des patrons *ECDL* sous forme de systèmes de transition représentant des automates observateurs. Les concepts informels de cette sémantique ont fait l'objet de notre travail dans [20].

6.4.1 Exemple de motivation

La Figure 6.6 montre un exemple de patron de réponse bornée ECDL qui est composé de quatre événements dans la partie Pre Clause, avec l'ordre *AllCombined* et quatre autres dans la partie Post Clause, qui sont triés selon l'option *AllOrdered.* L'évènement $r1$ ne peut pas se produire avant la première occurrence de l'évènement $e1$, cette propriété est précisée avec l'option *Precedency.*

Une transformation de ce patron ECDL en un observateur, génère un graphe d'états composé de 17 états et plus de 30 transitions. Figure 6.6 montre l'observateur correspondant à la propriété. Les états colorés représentent les états de la deuxième partie du patron, Tandis que le reste des

états (non colorés) sont ceux de la première partie du patron. L'état coloré en bleu s_{13} est l'état intermédiaire qui relie les deux parties de la propriété (Pre et Post).

```
ALL Combined{
    Exactly one Occurence of e1
    Exactly one Occurence of e2
    Exactly one Occurence of e3
    Exactly one Occurence of e4
}                                        Pre Clause

Leads To

ALL Ordered{
    Exactly one Occurence of r1
    Exactly one Occurence of r2
    Exactly one Occurence of r3
    Exactly one Occurence of r4
}                                        Post Clause

After at most 5 seconds
One of r1 can't occur before the first
occurence of e1
```

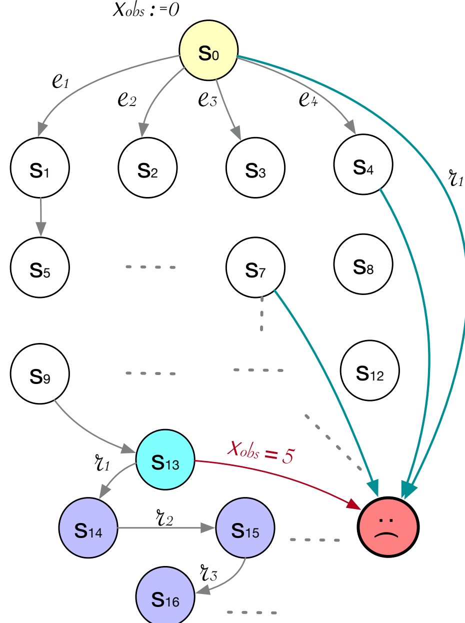


FIGURE 6.6 – Patron de **réponse** ECDL et l'observateur correspondant

La complexité que l'on remarque ici est la difficulté d'exprimer la sémantique de ce patron, qui modélise une propriété qui n'est pas très compliquée. Une relation de corrélation positive relie les nombre d'évènement et la complexité de sémantique du patron.

> *"Plus le nombre d'événements augmente, plus l'expression sémantique du patron ECDL devient complexe".*

Le but de notre approche compositionnelle est, d'une part, de nous permettre d'être capable de construire des observateurs complexes à partir de patrons élémentaires et de propriétés optionnelles en utilisant des opérateurs de composition et, d'autre part, de dériver les algorithmes de génération d'automates observateurs corrects pour les outils de model-checking et plus spécialement l'outil OBP.

6.4.2 Formalisation des patrons ECDL

Afin de réussir la composition des patrons de propriétés avec différents variants (options, scopes, patrons), nous allons d'abord procéder à la décomposition du patron en différents parties. Notre proposition est légèrement différente de notre décomposition que nous avons proposé en [20]. Cette différence est dû à la nouvelle structure de patron que nous proposons afin de mieux présenter les patrons de propriétés temporelles proposés en ECDL.

Exemple illustratif

Afin de mieux comprendre cette approche compositionnelle, nous proposons d'illustrer la composition avec un exemple de propriété simple (E1) qui repose sur un patron de réponse. Il représente une exigence pour un système de sécurité (appelé "***Akhbar Neb Security System***" ou ***AN2S*** en abrégé [123]. Le document écrit par [123] décrit les exigences pour l'***AN2S***, qui fournit un contrôle centralisé de diverses zones de sécurité équipées de capteurs et d'actionneurs pour la détection et la suppression des incendies et des intrusions.

En général, le système affiche des informations sur l'état des zones contrôlées et permet aux utilisateurs de coopérer avec les capteurs et les actionneurs des zones. Le système reçoit des signaux de détection d'incendie pour une zone spécifique et active les alarmes incendie et les systèmes d'extinction d'incendie.

L'une des exigences de ce système peut être représentée comme indiqué dans Tableau 6.1.

```
"If the an2s system receives signals from fire detection
sensors (FDS for short)" in the zone then :

"It shall activate an audible alarm" and, "signal exit
direction indicators",

"within 15 seconds".
```

TABLEAU 6.1 – Exigence E1

Le patron ECDL correspondant à cette exigence *E*1 est le patron de *Réponse* bornée avec le scope "**After Q**". Quatre options peuvent être drivées de cette propriété comme suit :

1. ***Nullity*** : Après la détection d'un feu dans une zone, AN2S doit impérativement recevoir un signal pour pouvoir déclencher les action nécessaires.Ce qui signifie que la négation de cette propriété ne doit jamais se produire. Cela se traduit comme suit :

   "*if fire", **Not** (AN2S_hasReceived_Signals) **might never** occurs.*

2. ***Precedency_1*** : Une alarme audible doit s'activer suite à la réception de signal de détection de feu par le système AN2S.

   En aucun cas cette alarme est autorisé à se déclencher avant le signal de AN2S. Cette option est traduite comme suit :

   "*One of activates_AudibleAlarm cannot occur beforeAN2S_hasReceived_Signals*"

3. ***Precedency_2*** : Les indicateurs qui indiquent les directions des sorties de secours (**Exit**) doivent s'activer suite à la réception de signal de détection de feu par le système AN2S.

En aucun cas ces indicateurs ne sont autorisé à se déclencher avant le signal de AN2S. Cette option est traduite comme suit :

"*One of Signals_Exist_Direction cannot occur before AN2S_hasReceived_Signals*"

4. ***Repeatability*** : Le comportement est répétable.

En utilisant la grammaire structurée proposée dans Chapitre 4, nous obtenons une traduction de la propriété de langage naturelle en un patron ECDL comme montre Figure 6.7.

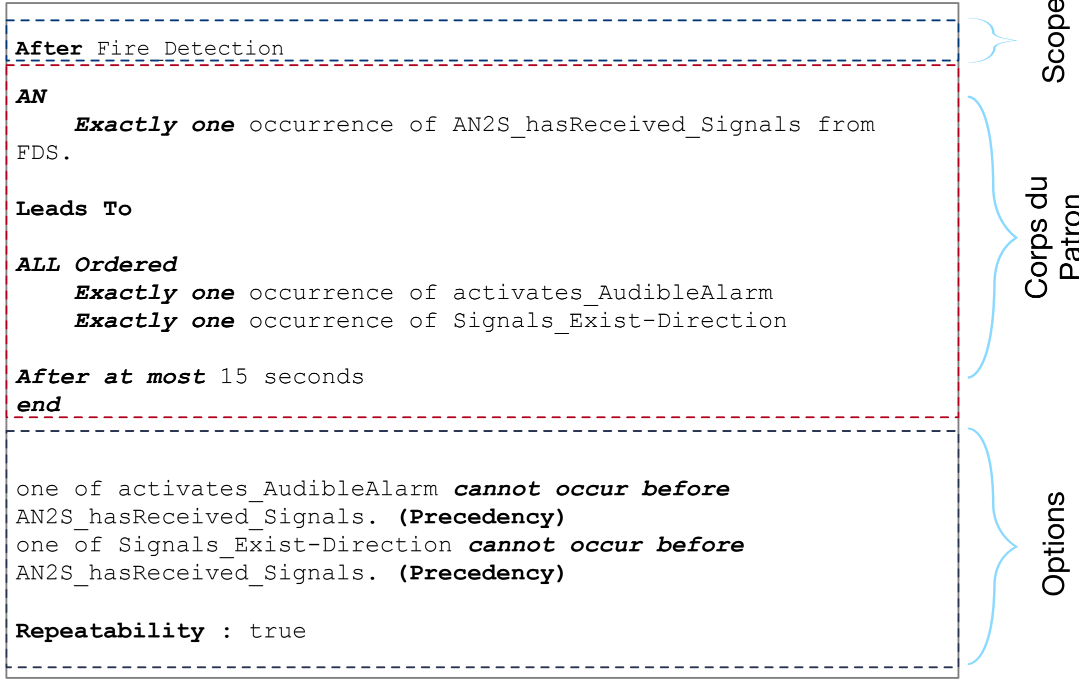


FIGURE 6.7 – Patron de réponse correspondant à **E1**

Un patron ECDL est composé principalement de trois parties principales comme indiqué dans Figure 6.7 :

- ***Scope*** : dans cet exemple, le scope est ***After Q*** ce qui signifie que la propriété est valide après l'occurrence d'un évènement. Dans notre cas, c'est la *détection de feu.*
- ***Patron*** : la propriété représenté par l'exemple de Figure 6.7 est un patron de Réponse bornée. En réponse à la réception du signal par le système AN2S, l'activation d'alarme audible ainsi que les signales de direction de sorties de secours doivent se produire dans un délai de 15 secondes.
- ***Options*** : cette partie regroupe les propriétés optionnelles qui sont dérivées de la propriété principale. Le patron de propriété $E1$ possède quatre options comme montré dans Figure 6.7.

Dans les sections précédentes, nous avons montré les algorithmes et les règles nécessaires à la transformation d'un patron sans options ou scope en utilisant une méthode de composition.

Dans la suite de ce chapitre, nous allons présenter les règles nécessaires à la composition de patron (obtenu précédemment) avec les options ainsi que les scopes pour compléter le processus de transformation.

6.5 Composition de patrons et options

En plus de la sémantique naturelle des patrons et des scopes, Dywer et al. [63, 62] ont fourni une sémantique formelle en les traduisant en différents logiques temporelles, associant ainsi chaque combinaison *patron/scope* à une formule temporelle correspondante. Comme il y avait 10 patrons et 5 scopes, ils ont dû traduire environ 50 combinaisons [7]. Dans [37], Castillos et al. ont proposé une approche basée sur la composition d'automates de Büchi [128] en définissant la sémantique de patron et scope par un automate chacun. Ensuite, ils suggèrent une opération de composition afin que la sémantique des propriétés soit définie par la composition des deux automates. Ils motivent leurs propositions par le fait que dans [7] la traduction de plus de 20 patrons et 20 scopes nécessite environ 400 formules temporelles. En outre, Dwyer et al. [63] ont déterminé de manière informelle des patrons génériques (par exemple, une première chaîne d'événements précédant une seconde chaîne d'événements) qu'ils ne parviennent pas à traduire en formules équivalentes de logique temporelle générique. En conséquence, ils n'ont traduit qu'un nombre limité de cas évidents (par exemple, des chaînes avec seulement 1 ou 2 événements). L'extensibilité et la généralité sont les principales limites de la sémantique de Dwyer et al [7].

De même, le système de patron CDL classique souffre d'une ambiguïté sémantique et d'une complexité de transformation de patrons en observateurs. Pour ces raisons, et en s'inspirant de la proposition de [37, 126], nous proposons une approche compositionnelle permettant de composer chaque patron avec les propriétés optionnelles *"options"* qui sera ensuite composer avec le scope correspondant.

6.5.1 Règles de composition de patrons et options

6.5.1.1 Spécification des options comme observateurs

Figure 6.2 montre les différents automates observateurs correspondants aux patrons de réponse (ANAN, AllOrderdAN,...) obtenus par transformation en appliquant les règles de construction (de 1 à 5) présentées dans Section 6.2.3.

Les options sont également décrits par des automates comme illustre Figure 6.8 pour un patron de type Réponse. Figure 6.8.a illustre un automate représentant l'option Precedency dans le cas de la variante *"may occur before"* (variante notée Precedency = true). Cette option spécifie que l'évènement *r* qui représente la réponse peut se produire avant la première occurrence de l'action *a*. Figure 6.8.b illustre un automate représentant l'option Precedency dans le cas

de la variante "*cannot occur before*" (variante notée Precedency = false). Cette option spécifie l'évènement r ne peut pas se produire avant la première occurrence de a. L'arrivée de cet événement avant a déclenche une transition vers l'état de rejet indiquant une erreur.

Figure 6.8.c illustre un automate représentant l'option *Repeatability*. Dans le cas de *Repeatability = true*), la transition comportant la réponse r revient vers l'état initial permettant au comportement de l'observateur de se répéter. Dans le cas échéant, *Repeatability = false*, la transition va directement à l'état de *succès* et y reste (Figure 6.8.d).

Figure 6.8.e illustre un automate représentant l'option *Nullity* dans le cas de la variante ***may never occur*** (variante notée *Nullity = true*). Cela est indiqué par une transition vers l'état de rejet avec l'évènement a. Pour l'instant, cet évènement est unique mais cette spécification pourrait être étendue dans une version ultérieure à une liste d'évènements.Figure 6.8.f illustre un automate représentant l'option Nullity dans le cas de la variante *may/must occur* (variante notée Nullity = false).

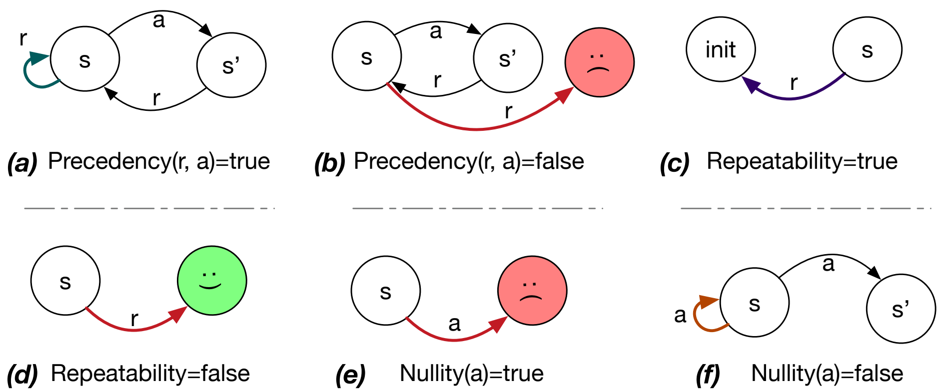


FIGURE 6.8 – Options sous forme d'automate

Dans chaque automate représentant une option (Figure 6.8), nous illustrons une transition particulière en gras représentée par une flèche colorée. Cette transition particulière, suivant les règles de composition décrites en Section 6.5.2, va permettre la composition des deux automates Pt et Op correspondant respectivement au *patron* et *option*.

6.5.2 L'opération de composition

Dans cette section, nous définissons formellement les opérations de composition de patron et options. Le principe de composition est basé sur plusieurs compositions élémentaires. Ces opérations de composition sont définies en ajoutant la transition de composition au patron (précédemment transformé) en fonction du type d'option considéré : *(Nullity, Repeatability ou Precedency)*.

- *Soit* $Pt = (Init_{Pt}, S_{Pt}, F_{Pt}, E_{Pt}, T_{Pt})$ l'observateur correspondant au patron Pt et,

$Op = (Init_{Op}, S_{Op}, F_{Op}, E_{Op}, \{T_{Op}\} \cup \{C_T\})$ est l'automate observateur représentant une option Op.

- Une option Op est caractérisé par un état initial noté : $Init_{Op}$, un ensemble fini des états nommé S_{Op}, ainsi qu'un ensemble fini d'états finaux F_{Op} qui sont $Reject$ ou $Success$, aussi un ensemble de labels désigné par E_{Op} et un ensemble de transitions $\{T_{Op}\} \cup \{C_T\}$, dont C_T est la transition spéciale nommée "*Transition de composition*".

Le pré-traitement le plus important avant de procéder à la composition, est la substitution de l'état source et de l'état cible d'une transition de composition. Nous devons les remplacer par les états s_i, s_j dans Pt en suivant les règles représentées en Section 6.5.2.1.

6.5.2.1 Les règles de substitutions

Afin de maintenir une composition correcte, la transition de composition doit être placée à l'endroit approprié. C'est pourquoi un ensemble de règles régissant la mise en œuvre de cette transition doit être suivi pour chaque option. Le facteur clé de cette opération de substitution est la sélection des états fixes d'une transition de composition ($Source(C_T)$ et $Target(C_T)$) représentant ses états source et cible. Dans la suite de cette section, nous allons introduire le concept d'état avant-final nommé BF tel que $BF = Pre(f, e)$ avec $f \in F$. Il s'agit de l'état qui est positionné avant l'état final dans la clause $Post$. Il est important d'identifier cet état de manière significative en raison de son rôle qui offre la possibilité de suivre un chemin de retour vers l'état initial (lorsque la répétabilité est vraie).

R1 : $\forall\ Repeatability : Source(C_T) = BF$

R2 : $Target(C_T) = \begin{cases} init_{pt} & if\ Repeatability = true. \\ Success & otherwise. \end{cases}$

R3 : $\forall\ Precedency(r, a) : Source(C_T) = Source(a)$

R4 : $Target(C_T) = \begin{cases} reject & if (Precedency = false). \\ Source(a) & otherwise. \end{cases}$

R5 : $\forall\ Nullity(a) : Source(C_T) = Source(a)$

R6 : $Target(C_T) = \begin{cases} reject & if (Nullity = false). \\ Source(a) & otherwise. \end{cases}$

Les règles $R1$ et $R2$ sont utilisées pour positionner la transition de composition de l'option $Repeatability$. Nous avons accordé à la transition de composition l'état BF comme source quel que soit la valeur de *repeatability vrai* ou $faux$ ($R1$). Afin de revenir à l'état initial de notre observateur, ce qui est requis par la valeur de répétabilité vraie (possibilité de répéter le comportement depuis le début de l'observateur), il est évident que la cible de C_T de la transition de composition sera l'état initial $init_{pt}$ comme le montre la Règle $R2$ dans sa première ligne.

Sinon, si la répétabilité est fausse, la cible sera l'état *success*. En utilisant les règles $R3$ et $R4$, il est possible de positionner la transition de composition liée à l'option *Precedency*.

Remarque. *$Precedency(r, a)$ définit si un événement de réponse r peut (ou non) se produire avant la première occurrence de a.*

La règle $R3$ attribue l'état source de l'événement a comme source quelle que soit la valeur de Precedency ($vrai$, $faux$) à la transition compositionnelle. Tandis que, la règle $R4$ définit la cible de la transition compositionnelle de Precedency. Nous passons ensuite à la dernière option, qui est $Nullity$. Nous rappelons que $Nullity(a)$ nécessite l'absence (ou non) de l'événement a. Quelle que soit sa valeur, la transition compositionnelle C_T de l'option $Nullity$ aura pour état $Source(a)$ comme le montre $R5$. Cependant, pour sa cible, cela varie en fonction de sa valeur ($Reject$ s'il elle est $faux$, ou $source(a)$ sinon) comme le montre $R6$.

6.5.2.2 Les règles de composition

Dans le but de réaliser la composition, il ne reste plus qu'à insérer la transition de composition à la place qui lui convient dans le patron, en marquant ses états (source et cible) avec les étiquettes appropriées.

Soit $Pt = (Init_{Pt}, S_{Pt}, F_{Pt}, E_{Pt}, T_{Pt})$ un automate de patron, et $Op = (Init_{Op}, S_{Op}, F_{Op}, E_{Op}, \{T_{Op}\} \cup \{C_T\})$ un automate d'option, où C_T est la transition de composition de Op. La composition de Op et Pt est un automate observateur où : $Pt \oplus Op = (Init, S, F, E, T)$ (nommé Pa) tel que :

- $S = S_{Pt} \cup S_{op}$
- $Init = Init_{Pt}$
- $F = F_{Pt} \cup F_{Op}$
- $T \subseteq S \times E \times S$ est la relation de transition qui est définie par les règles suivantes :
  1. **Transitions d'option** :

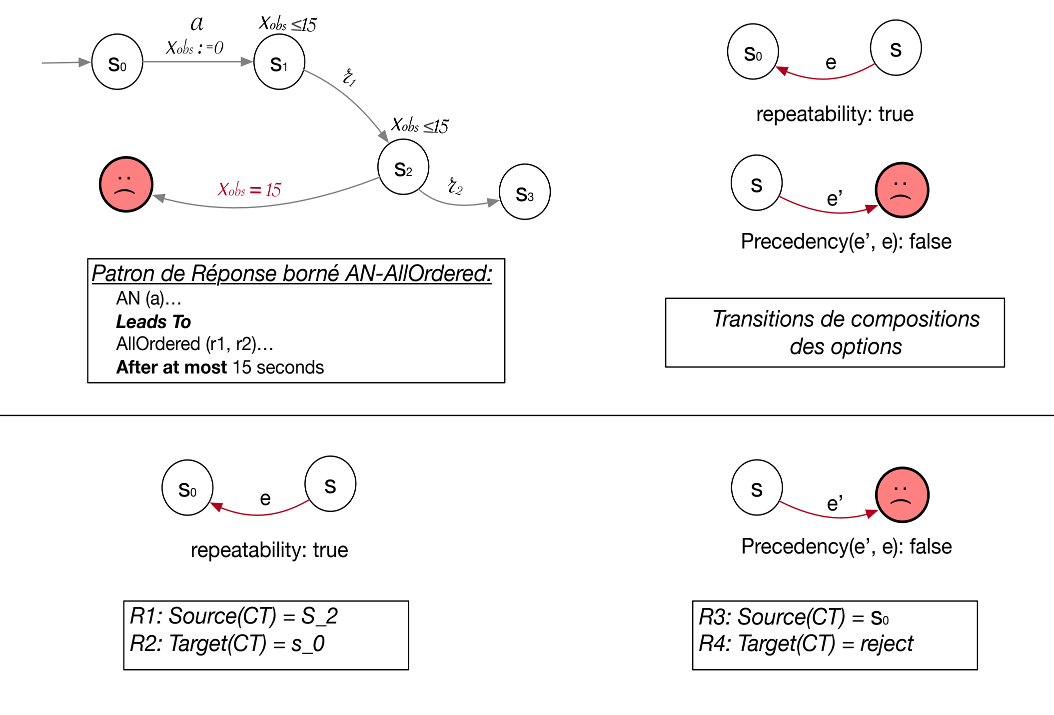
$$- \; Repeatability \begin{cases} \dfrac{s' \xrightarrow{l} s \in T_{op}, s' \xrightarrow{a} k \in T_{pt}}{s' \xrightarrow{a} s \in T_{pa}} \\ \text{(where } s' = BF \text{ and } k \in S_{pt}) \\ \dfrac{s \xrightarrow{l} Success \in T_{op}, s \xrightarrow{a} k \in T_{pt}}{s \xrightarrow{a} Success \in T_{pa}}. \end{cases}$$

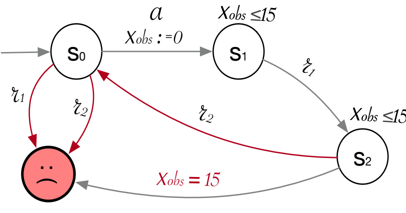
$$- \; Precedency \begin{cases} \dfrac{s \xrightarrow{a} s', s' \xrightarrow{r} s \in T_{pt}, s \xrightarrow{l} reject \in T_{op}}{s \xrightarrow{r} reject \in T_{pa}} \\ \dfrac{s \xrightarrow{a} s', s' \xrightarrow{r} s \in T_{pt}, s \xrightarrow{l} s \in T_{op}}{s \xrightarrow{r} s \in T_{pa}}. \end{cases}$$

– *Nullity*
$$\begin{cases} \dfrac{S\xrightarrow{a}S' \in T_{pt}, S\xrightarrow{l}S \in T_{op}}{S\xrightarrow{a}S \in T_{pa}} \\ \dfrac{S\xrightarrow{a}S' \in T_{pt}, S\xrightarrow{l}reject \in T_{op}}{S\xrightarrow{a}reject \in T_{pa}} \end{cases}.$$

2. **Transitions de Patron** : $\dfrac{s\xrightarrow{e}s' \in T_{Pt}}{s\xrightarrow{e}s' \in T_{Pa}}$

Figure 6.9 montre un exemple de composition utilisant les règles décrites précédemment.





Règles de Composition

FIGURE 6.9 – Application des règles de composition sur l'exemple de Figure 6.7

6.6 Composition des patrons et scopes

Le système original de spécification de propriété de patron est formé d'un patron Pt et d'un scope S. Comme mentionné précédemment, un patron spécifie ce qui doit se produire tandis que la scope spécifie quand le patron doit se vérifier. Soit O l'observateur correspondant à une propriété Pt donnée. En CDL classique, O peut observer si la propriété Pt tient globalement, c'est-à-dire pendant toute l'exécution du modèle.

Cette section montre comment O est modifié pour être capable de vérifier une propriété sur un scope donnée autre que la scope global. Les scopes proposés correspondent aux scopes définies par Dwyer et al. [63]. L'idée clé pour gérer les scopes est d'exprimer chacun d'eux comme un automate. Une approche de composition proposée par [37, 126] est adoptée dans ce travail pour obtenir un nouvel observateur O' en composant le patron de propriété à vérifier (Pt) et le scope (S).

Figure 6.10 illustre les automates de scope proposés par [126]. Les carrés sont utilisés pour représenter les états de composition. Les doubles carrés représentent les états de composition acceptables. Les règles de composition qu'ils proposent peuvent être utilisées pour composer un observateur avec une scope donné en substituant l'état cs par l'observateur suivant les règles de composition.

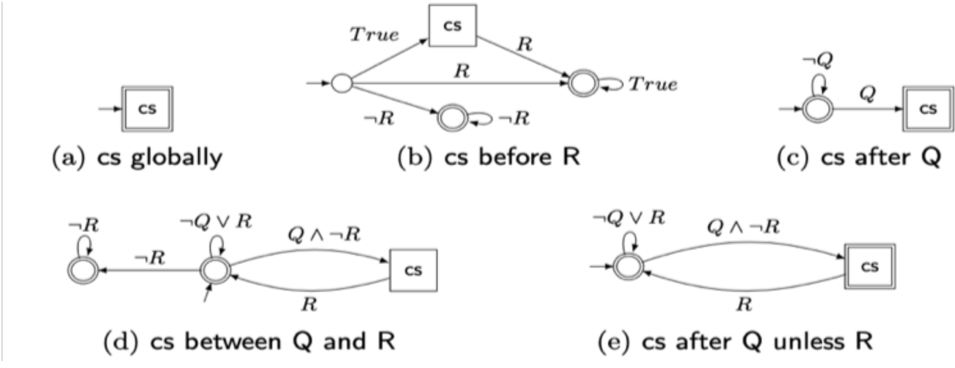
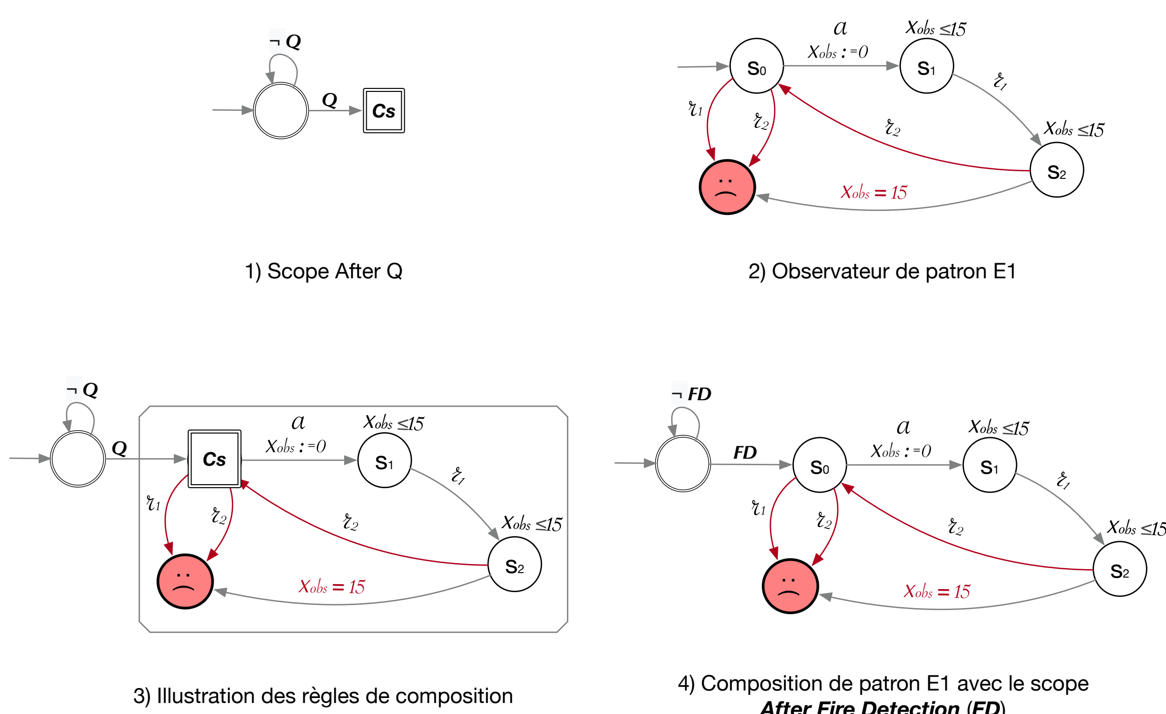


FIGURE 6.10 – Scopes automates [126]

L'ensemble d'états résultant de O' est l'union des ensembles d'états de O sans l'état de composition cs. L'état initial est l'état initial de O si le cs est initial, sinon c'est l'état initial de la scope. Les transitions résultantes sont définies par les règles montrées dans [126].

Figure 6.11 illustre l'exemple précédent avec le scope ***After Q***. Comme le montre , le cs est remplacé par l'observateur Figure 6.11.3. Les règles de composition sont appliquées Figure 6.11.4.



1) Scope After Q

2) Observateur de patron E1

3) Illustration des règles de composition

4) Composition de patron E1 avec le scope ***After Fire Detection*** (***FD***)

FIGURE 6.11 – Composition de patron de réponse ECDL avec scope "*After Q*"

6.7 Conclusion

La vérification des propriétés consiste en une analyse d'accessibilité des états de rejet sur le graphe d'exploration résultant de la composition du modèle à valider et des observateurs. Les automates observateurs sont générés en transformant les patrons de propriétés ECDL. La construction d'un observateur, transformant un patron de propriétés sans options, a été réalisée et peut être réutilisée pour définir une autre opération de composition avec plus de deux propriétés, afin de traiter des patrons génériques.

Le principe de construction d'un observateur par composition peut aussi s'étendre à la composition de plusieurs observateurs afin de pouvoir exprimer des propriétés plus complexes basées sur les propriétés élémentaires. Compte tenu du type d'exigences que nous avons à traiter dans certaines applications industrielles, notre objectif est d'expérimenter ces extensions pour mener des actions de validation formelle sur les modèles industriels.

Dans ce chapitre, nous formalisons la composition d'éléments de base constitutifs des patrons de propriétés avec des éléments optionnels qui viennent enrichir les patrons. Ce travail nous permet de disposer d'un cadre formel de construction de propriétés plus complexes à partir d'éléments de base et pouvoir valider formellement les transformations. Par la suite, nous pourrons donc continuer à concevoir des extensions ECDL en rajoutant des options

supplémentaires.

Compte tenu du type d'exigences auxquelles nous sommes confrontés dans certaines applications industrielles, notre objectif est de tester ces extensions afin de mener des actions de validation formelle sur des modèles industriels dans Chapitre 8.

Des règles de composition, implémentées en Coq pour les patrons de type Réponse, permettent de générer des systèmes de transitions bisimilaires aux systèmes de transitions de référence (observateurs), décrivant la sémantique des patrons. L'implémentation reste à être finalisé pour les autres patrons de type (Précedence bornée, Absence...). Notre technique de preuve est basée sur la démonstration d'une équivalence de bi-simulation entre les automates sources (obtenus par OBP) et cibles, qui sont générés par notre approche compositionnelle Chapitre 7.

CHAPITRE

7

Validations des transformations

« Research is to see what everybody else has seen, and to think what nobody else has thought. »

— Albert Szent-Gyorgyi

Sommaire

7.1 Introduction **108**
7.2 Vérification formelle - "Preuve Bisimulation" **108**
7.3 Description de Coq **109**
7.4 Sémantique opérationnelle des automates observateurs **110**
7.4.1 Automates 111
7.4.2 Implémentation en Coq 114
7.5 Sémantique opérationnelle des patrons ECDL **116**
7.5.1 Type de patron 117
7.5.2 Type d'occurrence des évènements 119
7.5.3 Nombre d'occurrence des évènements 120
7.5.4 Implémentation en Coq 120
7.6 Construction de patrons **121**
7.7 Règles de traduction **121**
7.7.1 Implémentation en Coq 122
7.7.2 Théorème à prouver 123
7.8 Conclusion **127**

7.1 Introduction

Étant donné la difficulté de vérifier des programmes relativement simples, la plupart des outils essaient de simplifier ce processus et de le rendre aussi automatisé que possible, ce qui peut étendre considérablement leur base de code fiable. Cependant, le principal enjeu est de savoir comment analyser et vérifier ces modèles à l'aide d'outils.

La validation de traduction consiste à transformer un programme et à le valider a posteriori afin de détecter une modification de sa sémantique. Cette approche peut être utilisée dans un compilateur vérifié, à condition de prouver formellement que la validation est correcte.

Nous présentons dans ce chapitre, les règles de transformations et de compositions des patrons ECDL et de leurs preuves Coq de correction.

7.2 Vérification formelle - "Preuve Bisimulation"

Puisque ECDL est un langage formel et que nos règles de transformation permettent de transformer des spécifications écrites sous formes de patrons en observateurs, il est nécessaire de s'assurer que la spécification obtenue après la réécriture des propriétés en patrons ECDL a la même signification que la spécification originale écrite en langage naturel. Pour s'assurer que le sens de la spécification ne change pas, il est nécessaire de vérifier que tout patron ECDL sur lequel une règle de transformation a été appliquée est équivalent à l'observateur généré au sens de la bisimulation [36]. S'il existe deux observateurs O_1 et O_2, on dit que O_1 et O_2 sont équivalents au sens de la *bisimulation* si O_1 simule O_2 et de même O_2 simule O_1.

Par conséquent, on dit que O_2 simule O_1 s'il existe une relation entre les états de O_2 et ceux de O_1 telle que : pour un déclenchement d'une transition d'un état de O_2, il est possible de déclencher cette transition d'un état équivalent de O_1 et la transition arrive dans un état équivalent de O_1.

Pour montrer la bisimulation, il suffit donc de montrer la bisimulation de l'observateur sur lequel s'applique la règle et de montrer que les états initiaux et finaux sont équivalents. La définition formelle de la simulation est la suivante :

Définition 7.2.1. *Soient O_1 et O_2 deux observateurs, on dit que O_2 simule O_1 ssi il est possible de trouver une relation $\mathcal{R}$ entre les états de O_1 et les états de O_2 telle que pour toute transition t, on a :*

$$\forall (s, s' \in S_{O_1}, s_1 \in S_{O_2}).(s \rightarrow_{O_1} s' \wedge \mathcal{R}(s, s_1)) \implies \exists s'_1 \in S_{O_2}.(s_1 \rightarrow_{O_2} s'_1 \wedge \mathcal{R}(s', s'_1)) \quad (7.1)$$

Et

$$\forall s_0.(s_0 = init_{O_1}) \implies \exists s_0'.(s_0' = init_{O_2}) \wedge \mathcal{R}(s_0, s'_0) \quad (7.2)$$

$$\forall s.(s \rightarrow_{O_1} reject) \implies \exists s'.(s' \rightarrow_{O_2} reject) \land \mathcal{R}(s, s') \tag{7.3}$$

$$\forall s.(s \rightarrow_{O_1} success) \implies \exists s'.(s' \rightarrow_{O_2} success) \land \mathcal{R}(s, s') \tag{7.4}$$

Les preuves de la bisimulation sont très longues et techniques. Dans ce chapitre, nous exposons donc uniquement les grandes lignes de ces preuves, en précisant les principaux critères justifiant notre conviction de la validité de ces règles.

Pour réaliser la preuve de la bisimulation et définir les sémantiques des patrons et observateurs ainsi que les fonction de transformations et compositions, nous avons choisi d'utiliser l'assistant de preuve Coq.

7.3 Description de Coq

Coq est un assistant de preuve basé sur la théorie des types. C'est un environnement pour l'édition et la vérification de théories mathématiques exprimées en logique d'ordre supérieur. Coq fournit un langage, appelé le calcul des constructions inductives "*Calculus of Inductive Constructions*", pour représenter les objets et les propriétés. L'ingrédient principal de ce calcul est un lambda-calcul typé d'ordre supérieur : les fonctions, les types ou les propositions sont des objets de première classe et il est par conséquent permis de quantifier sur les variables des fonctions, des types ou des propositions.

Nous précisons que nous allons pas détaillé la syntaxe de Coq ce qui est déjà présenté dans le manuel de référence en [82].

Dans le reste du ce chapitre, nous allons utilisés des notations et constructions que nous allons expliquer en ce qui suit.

Sets : Les types de données sont représentés par des objets dont le type est une constante spéciale appelée *Set*. Par exemple, le type *nat* des nombres naturels est lui-même un objet de type *Set*. Si T et U sont des objets de type *Set*, alors $T \rightarrow U$ est un nouvel objet de type *Set* qui représente le type des fonctions de T à U. De nouveaux types peuvent être introduits par des déclarations inductives :

Inductive *name* : **Set** ≔ $c_1 : T_1|...|c_n : T_n$

Cette commande ajoute un nouveau nom de type concret dans l'environnement, ainsi que les constructeurs de ce type nommés $c_1, ..., c_n$. Chaque c_i a le type T_i. Un constructeur constant sera déclaré comme c : *name*, un constructeur *unaire* attendant un argument de type α sera déclaré comme $c : \alpha \rightarrow$ *name*. Le constructeur peut accepter des arguments récursifs de type *name*. Par exemple la définition des listes de type α est définie comme suit :

Inductive $list_\alpha$: **Set** ≔ `nil` : $list_\alpha$ | `cons` : $\alpha \rightarrow list_\alpha \rightarrow list_\alpha$.

Si t est un objet de type inductif T, il est possible de construire un objet par *pattern-matching* sur la structure des valeurs du type T. Lorsque T est défini inductivement, il est possible de définir une fonction récursive dépendant de x de type T dès lors que les appels récursifs dans la définition de x sont effectués sur des objets structurellement plus petits que x.

Propositions : Les propositions logiques sont représentées par des objets de type spécial nommé $Prop$. Si T et U sont des objets de type $Prop$, alors $T \rightarrow U$ (également écrit $T \implies U$ dans ce document) est un nouvel objet de type $Prop$ qui représente la propriété "T *implique* U". Le prologue de Coq contient la définition des connecteurs logiques habituels tels que la conjonction, la disjonction ou encore l'égalité. La quantification universelle d'une propriété P dépendant d'une variable x de type A s'écrit $(x : A)P$.

Abstraction et application : Un type ou un objet t peut être librement abstrait par rapport à un type ou une variable objet x de type τ. Le terme abstrait s'écrit $[x : \tau]t$ ou simplement $[x]t$ lorsque τ peut être déduit du contexte. Un terme t représentant une construction fonctionnelle peut être appliqué à un terme u du type approprié, l'application s'écrit (tu).

Prédicats : Un prédicat P sur un type α est un terme de type $\alpha \rightarrow Prop$, c'est-à-dire une fonction qui, étant donné un élément a de type α, produit une propriété (Pa) de type $Prop$. Il est possible de définir une proposition soit comme une fonction définie par cas et récurrence sur un paramètre défini inductivement, soit de manière inductive. Un prédicat défini inductivement ressemble à la définition d'un programme *Prolog* avec des clauses. Par exemple, étant donné une relation binaire $\mathcal{R}$ de type $\alpha \rightarrow \alpha \rightarrow Prop$ et un prédicat P de type $\alpha \rightarrow Prop$, il est possible de définir inductivement le prédicat P^* qui est vrai pour tout y de type α tel qu'il existe un chemin de x qui a satisfait P à y avec deux éléments consécutifs satisfaisant $\mathcal{R}$.

Déclarations : Les Déclarations de la forme :

```
Definition name  := t.
```

sont utilisées pour introduire *name* comme une abréviation de terme t.

7.4 Sémantique opérationnelle des automates observateurs

La sémantique opérationnelle des automates observateurs est définie à l'aide d'un système de transition étiqueté. Un système de transition étiqueté est un sous ensemble de $State \times Event \times \mathcal{X} \times State$, où $State$ est l'ensemble des états d'un observateurs et $Event$ est l'ensemble des événements possibles, quant à $\mathcal{X}$ représente les prédicats de l'horloge d'un observateur. On écrit $s \xrightarrow{\sigma} s'$ qui signifie qu'un événement σ permet de passer de l'état s à l'état s'. On écrit $s \xrightarrow{\varphi} s'$ qui signifie qu'un prédicat φ est vérifié et cela permet de passer de l'état s à l'état s'. Une conjonction ou disjonction d'un évènement σ et d'un prédicat de l'horloge φ est totalement possible comme montré dans plusieurs observateurs dans ce travail.

7.4.1 Automates

Un automate observateur en ECDL est un automate exprimant la propriété à vérifié et réagit en fonction des événements, des changements des états et des modifications des valeurs des variables du système observé. L'automate d'un observateur ECDL est plus prioritaire que le système observé et il est déterministe. Chaque état d'un observateur peut être : Un état dit normale (ou d'observation), Un état dit rejet (propriété non valide) ou un état dit succès(propriété valide).

Un automate est décrit par un nom *name* correspondant au type de patron (Réponse, ...), un ensemble d'événement Σ, un ensemble S_{enum} de noms correspondant aux états de l'automate, un ensemble T de transitions, un ensemble SF des états finaux ainsi qu'un nom d'état initial $init$.

Un état ou une configuration se compose de l'emplacement actuel. La variabilité des emplacements est déclenchée par la survenance d'un événement et éventuellement la comparaison de la valeur de l'horloge avec des délais que le système doit le respecter. Le type de la transition (entrante ou sortante) avec son état source et cible, détermine s'il y'a une initialisation ou désactivation de l'horloge. Cependant, nous pouvons définir deux types de transitions entre états : simple, boucle (une transition qui a le même état source et cible). L'ensemble des transitions $sortantes$ des états finaux de "reject" doit être vide.

Dans le reste de document, on va symbolisé l'occurrence des événement comme suit : e : indique que e se produit seulement une fois. $e+$: indique que e se produit une ou plusieurs fois. $e-$: indique que e ne peut jamais arrivé. $e_1, ...e_n$: indique que ces événements se produisent suivant un ordre séquentiel (dans le cas de l'opérateur *AllOrdered*). $e_1|...|e_n$: indique que ces événements se produisent en parallèle (dans le cas de l'opérateur *AllCombined*).

Les fonctions : $src(e) = s \Rightarrow src(e) = Pre(s, e)$ et $trg(e) = s' \Rightarrow s' = Post(s', e)$, indique que s représente l'état source de e tandis que s' est son cible.

Définition 7.4.1 (Types de Transitions). *On dit que t une transition simple alors : $src(t) \neq trg(t)$, on dit que t est une transition boucle si elle boucle un état sur lui même. Donc, $src(t) = trg(t)$*

7.4.1.1 Sémantique d'occurrence des événements

Soit s, s' deux états de l'automate observateur, les transitons déclenchées par un évènement e avec ces différents types d'occurrences entre ces deux états suit les règles suivantes :

- $iff\ e : s \xrightarrow{e} s'$
- $iff\ e+ : s \xrightarrow{e} s' \wedge s' \xrightarrow{e} s'$

- soit $s_{final=reject}$ est un état de rejet, $iff\ e- : s \xrightarrow{e} s'\ s' = s_{final=reject}$
- *Séquence (AllOrdered) version acyclique* : Soit $s_i \in S\ \ avec\ \ i \in [1, n]\ iff\ e_1, e_2, ...e_n$: $init \xrightarrow{e_1} s_1 \wedge s_1 \xrightarrow{e_2} s_2 ... \wedge S_{n-1} \xrightarrow{e_n} s_n$
- *Séquence (AllOrdered) version cyclique* : Soit $s_i \in S\ \ avec\ \ i \in [1, n]\ iff\ e_1, e_2, ...e_n$: $init \xrightarrow{e_1} s_1 \wedge s_1 \xrightarrow{e_2} s_2 ... \wedge S_{n-1} \xrightarrow{e_n} s_n$ avec $Trg(e_{n-1}) = Src(e_n)$
- *AllCombined version acyclique* $iff\ e_1|e_2|...|e_n$: $init \xrightarrow{e_1} s_1 \wedge init \xrightarrow{e_2} s_1 ... \wedge s_0 \xrightarrow{e_n s_1}$
- *AllCombined version cyclique* soit e_m un évènement de réponse ou précédence, $iff\ e_1|e_2|...|e_n$: $init \xrightarrow{e_1} s_1 \wedge init \xrightarrow{e_2} s_1 ... \wedge s_0 \xrightarrow{e_n s_1}$ où $Src(e_m) = Trg(e_n) \wedge Trg(e_m) = init)$

Afin d'exprimer les notations ci-dessus par des règles d'inférences, nous avons utilisé une définition simple de l'observateur sous forme d'un graphe (V, T, Src, Trg, Lab) tel que :

- V : l'ensemble de nœuds qui associe à chaque état un nœud,
- T : l'ensemble des arêtes qui représente les transitions. $T_{boucle} \in T$ représente le sous ensemble de transitions boucles, $T_{simple} \in T$ est le sous ensemble de transitions simples $(T_{simple} \cup T_{boucle})$.
- $Src : T \longrightarrow V$: cette fonction associe à chaque arête $t \in T$ un état source $s \in V$ noté $(s, t)\ src(t) = s$.
- $Trg : T \longrightarrow V$: cette fonction associe à chaque arête $t \in T$ un état cible $s' \in V$ noté $(t, s')\ Trg(t) = s'$.
- $Lab : T \longrightarrow \Sigma \cup \mathcal{X}$: une fonction qui étiquette chaque arête $t \in T$ avec une labelle de l'ensemble $\sum$.
- $\Sigma, \mathcal{X}$: Ensemble finis d'alphabet (évènement), ensemble de prédicats sur l'horloge, respectivement. On note un label comme suit : $t[l]$ (l'arête t possède l comme label) $lbl(t) = l$.

Remarque. *Une arête qui possède la source s définis par la fonction $src : t \longrightarrow s$ et une destination s' définit par $trg : t \longrightarrow s$ ainsi qu'une étiquette l ($lbl : t \longrightarrow l$), peut s'écrire comme suit : $t(s, l, s')$, cette notation sera utilisée pour exprimer les règles d'inférence.*

Les règles d'inférence qui définissent la sémantique opérationnelle des automates sont les suivantes :

La première règle concerne l'étiquetage des transitions, chaque transition dans l'automate observateur est associés avec un label que ce soit un évènement et/ou un prédicat sur l'horloge. La conjonction et disjonction des étiquettes est toute à fait possible, mais la règle ci-dessous montre le cas d'un étiquetage simple (sans conjonction/disjonction).

$$\frac{\{t \in T\}\ l \in \Sigma \cup \mathcal{X}}{\rho \vdash lbl\,(t) : t \longrightarrow l = t[l]}\, obs_1$$

Soit e_i un évènement qui possède l'un des préfixes *Exactly one* ou *Each* qui signifie que cet évènement peut se produire une seule fois, cet évènement déclenche une transition simple entre deux états s, s' : $simple(s, e, s') => src(e) = s, trg(e) = s'$

$$\frac{t \in T \ s, s' \in V \ e \in Event}{\rho \vdash (e) \longrightarrow (s, t[e], s')} obs_2$$

Soit e_i un évènement qui possède l'opérateur *OneOrMore* qui signifie que cet évènement peut se produire une ou plusieurs fois (e^+), cet évènement se représente par deux transitions, la première est simple entre deux états s, s', et la deuxième boucle sur l'état s'. Les deux transitions sont défini comme suit : $loop(s, e, s') = src(e) = s, trg(e) = s' \wedge src(e) = s', trg(e) = s' => t(s, e, s') \wedge t(s', e, s')$

$$\frac{t \in T \ s, s' \in V \ e \in Event}{\rho \vdash (e+) \longrightarrow loop(s, t[e], s')} obs_3$$

Soit e_i un évènement qui ne doit pas se produire (en général définie par l'option nullity), cet évènement déclenche une transition simple entre un état v et l'état de $reject$: $simple(s, e, reject) => src(e) = s, trg(e) = reject$

$$\frac{t \in T \ s, s' \in V \ e \in Event}{\rho \vdash (e-) \longrightarrow (s, t[e], s'), s' = reject} obs_4$$

L'occurrence de deux évènements de type séquence (AllOrdered) est décrite par la règle suivante :

$$\frac{s, s', s'' \in V \ e, e' \in Event}{\rho \vdash (e; e') \longrightarrow (s, t[e], s') \wedge (s', t[e], s'')} obs_5$$

L'occurrence de deux évènements de type (AllCombined) est décrite par la règle suivante :

$$\frac{s, s', s'', s''' \in V e, e' \in Event}{\rho \vdash (e|e') \longrightarrow (s, t[e], s') \wedge (s', t[e'], s''') \wedge (s, t[e'], s'') \wedge (s'', t[e], s''')} obs_6$$

L'initialisation de l'horloge de l'observateur s'effectue en affectant la valeur 0 à la variable représentant l'horloge, la même chose pour la dés-activation sauf que cette fois ci en affectant la valeur -1.

$$\frac{\varphi\in\mathcal{X},\ s'\in V}{(s, t\,[\varphi := 0]\,, s')}obs_7$$

$$\frac{\varphi\in\mathcal{X}\ s, s'\in V}{(s, t\,[\varphi := -1]\,, s')}obs_8$$

Une propriétés de temps est vérifiée si la condition sur l'horloge à vérifier est vraie, sinon, la propriété n'est pas valide et par conséquent déclenche une transition vers le mauvais état "reject". La règle d'inférence correspondante à cette propriété est définie comme suit :

$$\frac{\varphi\in\mathcal{X}\ s, s'\in V\ \ \approx\ \in I}{\rho\vdash(\varphi\approx D \nvDash True)\longrightarrow(s, t\,[\varphi \not\approx D]\,, s')\,, s' = reject}obs_9$$

I : est l'ensemble des invariants de comparaisons $(<, \leq, =, \geq, >)$*, et D : délai à comparer avec la valeur actuelle de l'horloge*

La sémantique opérationnelle des automates observateurs ECDL contient neuf règles, qui sont données dans cette section (ci-dessus). La règle obs_1 décrit comment étiqueter une transition. Les règles obs_2, obs_3 et obs_4 décrivent comment exécuter différentes transitions tout dépend de type d'occurrence des évènements qui les déclenchent : obs_2 décrit la sémantique pour exécuter une transition d'un évènement qui se produit une seule fois, obs_3 décrit le cas où la transition dépend d'un évènement qui peut se produire plusieurs fois, dans ce cas deux transitions sont créées, la première est simple et la seconde est boucle. La règle obs_4 décrit la sémantique pour exécuter une transition partant à un état de "reject" car l'évènement ne doit pas arriver. Les règles obs_5 et obs_6 décrit le cas où les évènements se produisent d'une manière séquentielle ou parallèle. Enfin, les règles obs_7, obs_8 et obs_9 décrivent la sémantique des transitions temporelles, c'est-à-dire le cas d'une transition qui peut être exécuter en désactivant, initialisant ou évaluant la valeur de l'horloge.

7.4.2 Implémentation en Coq

L'implémentation des patrons observateurs ECDL décrit dans la section précédente nécessite de décrire le langage des automates observateurs, les structures modélisant l'état courant d'un

automate, et les règles d'inférence. Les patrons observateurs et les états sont décrits en utilisant les types inductifs de Coq, les règles d'inférence sont décrites en utilisant les prédicats inductifs.

Tout d'abord, un type transition permet de décrire l'ensemble des transitions. Il est défini de la manière suivante :

```
Inductive transition  :=
|Simple  : src_state -> trg_state -> label -> guard -> bool -> transition
|Loop  : src_state -> trg_state -> label -> guard -> bool -> transition
|Verif  : src_state -> trg_state -> label -> guard -> bool -> transition
```

Cette définition doit être lue de la manière suivante : le type transition peut être construit à partir de trois constructeurs(*Simple*, *Loop*, *Verif*). Chaque constructeur est une fonction qui permet de construire une transition à partir de paramètres. Le constructeur *Simple* par exemple permet de construire une transition à partir de cinq paramètres : un identifiant correspondant à l'état source, un identifiant correspondant à l'état cible, une étiquette de transition, une garde et une valeur booléenne indiquant si la transition est finale. Dans notre cas, les états sont identifiés par une chaîne de caractère. Ici, src_state et trg_state correspondent donc à un renommage du type String de Coq. Pour le constructeur *Loop*, la valeur booléenne indique si la transition est boucle en vérifiant si l'état source de cette transition est aussi son état cible ($src_state = trg_state$). Tandis que pour le constructeur $Verif$, sa valeur booléenne indique si la transition se termine dans un état de vérification ("success" ou "reject").

De même, le type des automates observateurs est défini de la manière suivante :

```
Inductive observer :=
|Automata :list_state -> list_transitions -> list_state -> list_state -> state -> observer .
```

Le constructeur Automata permet de construire un automate observateur à partir d'une liste des états de l'automate, d'une liste de transitions, de deux listes décrivant les noms des états finaux (success et reject) et du nom de l'état initial. La liste des transitions hérite donc sa définition du type inductive transition. L'ensemble des états de l'observateur est décrit par le type inductif suivant :

```
 Inductive state  :=
|Automata_state  : state -> name_state -> state
```

Le constructeur Automata_state permet de construire un état de l'automate.

Les fonctions *init* et $final$ sont encodées grâce à des prédicats inductifs de Coq, qui sont définis grâce à un ensemble de constructeurs. Ces constructeurs permettent de définir un

```
Inductive isStateFinal  : state -> observer -> Prop
```

ensemble de règles pour construire des éléments qui satisfont le prédicats (à la manière des règles d'inférence). La fonction final par exemple est encodée par un prédicat de type suivant :

Ce prédicat inductif est définie de telle manière que pour tout état s de type $state$ et tout observateur O de type observateur, $isStateFinal\ s\ O$ est vrai $ssi\ final(s)$ est vrai dans l'observateur O.

De la même manière, la fonction *init* est codée par le prédicat inductif suivant :

```
Inductive isInitialState : state -> observer -> Prop
```

Enfin, un prédicat inductif *step* permet d'encoder les règles d'inférence de la sémantique des automates. Ce prédicat est défini avec le type suivant :

```
Inductive step  : state -> observer -> label -> state -> Prop
```

Soient deux états $s, s' \in state$, un observateur $O \in observer$ et une étiquette de transition $l \in label$, le prédicat inductif $(step\ s\ O\ l\ s')$ est vrai $ssi\ s \xrightarrow{l}_O s'$ cela signifie qu'il est possible de passer d'un état s à l'état s' par la transition l dans l'automate O.

7.5 Sémantique opérationnelle des patrons ECDL

Dans cette section, nous allons définir un ensemble de règles de langage de patrons de propriétés ECDL nécessaire pour les traduire en automates observateurs. Dans cet ensemble, un patron de propriété est représenté par un ensemble d'expressions et d'opérateurs. Ces opérateurs ont un nom, une position et un contenue qui est l'expression. Les noms des opérateurs positionnés au début de (*PreClause, PostClause*) appartient à l'ensemble (*AN, AllOrdered, AllCombined*) et définie le type d'occurrence de l'ensemble des événements possibles. L'expression qui suit ces opérateurs, contient un mot clé qui appartient à l'ensemble (*Exactly one, Each, One or more*) qui définie le nombre d'occurrence des évènements. Un autre opérateur très important qui nous permis de connaître le type du patron (Réponse, Précedence, Absence, ...) se positionne au milieu des deux parties *Pre* et *Post* (si elle existe, sinon après la première et seule partie dans le cas de patron d'existence et absence). Son nom appartient à l'ensemble suivant (*LeadsTo, Precedes, Exists, Absent*). La garde a les mêmes valeurs de vérité en observateur et en propriétés de patrons. Il n'est donc pas nécessaire de décrire la manière de les évaluer pour prouver l'équivalence sémantique.

Pour la traduction des observateurs, les variables nécessaires sont des variables indiquant le type et le nombre d'occurrence des évènements, ainsi que la variable qui indique le type de patron. Ces variables sont la base de définition des patrons sans options ni propriétés temporelles. Afin de définir la notion du temps et les options, d'autres variables sont à considérer. La valeurs de ces variables (ou opérateurs) sont dans des ensembles *Constants* des littéraux dans la grammaire (détails en Annexe A). La sémantique opérationnelle pour un patron de propriétés ECDL est décrite comme un système de transitions étiquetées qui associe à chaque opérateur ou variable son équivalent en LTS.

7.5.1 Type de patron

Le type de patron de propriétés ECDL peut être déterminer en se basant sur des mots clés nommés "Pattern Indicators" (*PI*) placés après la partie *PreClause* et avant *PostClause* dans le cas de patron de réponse ou précédence. PI $\in \{LeadsTo, Preceds, Exists, Absent\}$. Figure 7.1 montre quatre différents patrons identifiés se basant sur les PI, en plus, de *timeBound* pour indiquer les patrons temporels (réponse bornée, ...).

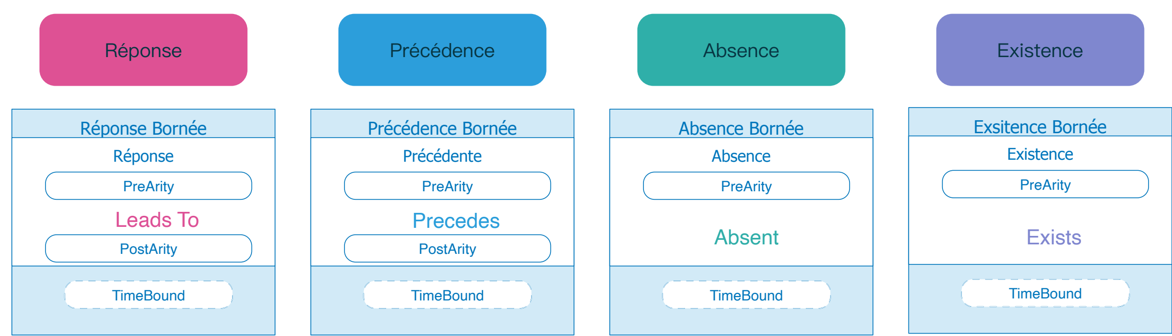


Figure 7.1 – Identification d'un type de patron en fonction de certains opérateurs

L'ensemble des substitutions qui permet d'associer à chaque patron un LTS est décrit d'une manière inductive par les propositions suivantes :

- ***LeadsTo*** : Selon ce mot clé le patron correspondant est un patron de réponse, cela peut se présenter par un LTS composé d'un état initial q_0, un état final q_f, et deux sous systèmes de transitons SLTS. Chaque SLTS représente une partie comme montré dans Figure 7.2(a) et est constitué d'un nombre d'état et de transitions.
- ***Preceds*** : Ce mot clé indique que le patron correspondant est un patron de précédence. Comme pour le patron de réponse, il peut être présenter par un LTS composé d'un état initial q_0, un état final, et deux SLTS(Figure 7.2 (a)).
- ***Exists*** : Le patron correspondant à cet opérateur est un patron d'existence. Contrairement aux patrons précédent, ce patron possède un seul sous système SLTS correspondant à la seule partie *PreClause* (Figure 7.2 (b)).

– ***Absent*** : Selon ce mot clé le patron correspondant est un patron d'absence. Ce patron suit le même raisonnement que pour le patron d'existence, son LTS correspondant donc se compose d'un état initial q_0, un état final q_f, et un SLTS (Figure 7.2 (b)).

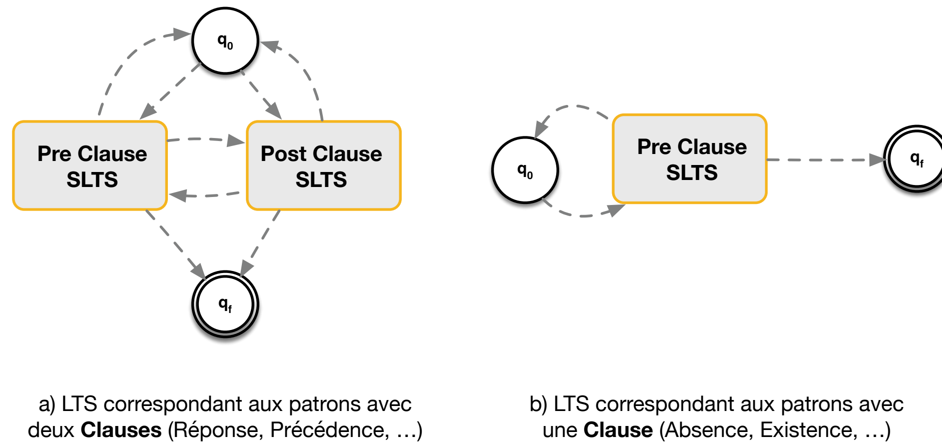


FIGURE 7.2 – LTS correspondants aux patrons ECDL

La sémantique de ces types de patrons est décrite par un système de déduction qui est inclus dans $Pattern \times PI$. Soit $p \in Pattern$ et $i \in PI$, $i \vdash p$ signifie que le patron p permet de satisfaire l'indicateur i. Les règles associées sont les suivantes :

$$\frac{i(x) \vdash p(y)}{i \vdash x \quad p \vdash p(y_{|x})}(pi_p) \tag{7.5}$$

pi_p est la règle pour savoir quel type de patron $p(y)$ correspond à l'indicateur $i(x)$. $x \in LeadsTo, Preceds, Exists, Absentety \in Response, Precedence, Absence, Existence$. La notation $y_{|x}$ indique la valeur de y qui est dépendante de la valeur de x. Les deux règles suivantes montre la substitution des variable x et y par deux valeurs différentes.

$$\frac{y_{|x} \vdash Response}{i(x) \vdash x \longmapsto LeasTo}(pi_{res}) \quad \frac{y_{|x} \vdash Precedence}{i(x) \vdash x \longmapsto Preceds}(pi_{prec}) \tag{7.6}$$

Comme nous avons mentionné plus haut, chaque type de patron peut s'écrire sous forme d'un LTS. La sémantique des LTS correspondants aux différents types de patrons est décrite par un système de déduction qui est inclus dans $LTS_{pt} \times Pattern$. Soit $p \in Pattern$ et $g \in LTS_{pt}$, $g \vdash p$ signifie que le patron p permet de satisfaire le LTS g. Un système de transition étiqueté (LTS) est un uplet $(Q, A, \longrightarrow, q_0, q_f)$ avec Q un ensemble d'états, A un ensemble d'actions et $\longrightarrow \subseteq Q \times A \times Q$ la relation de transition, ou ensemble de transitions.

- Soit $\mathbb{T}^r$ le SLTS correspond à la partie *PreClause* dans un patron, tandis que $\mathbb{T}^s$ est SLTS de la partie *PostClause* (pour un patron réponse ou précédence seulement). Les règles associées aux LTS des patrons sont les suivantes :

$$\frac{p(y) \vdash Response}{p(y) \vdash \langle q_0, A^r \cup A^s, \mathbb{T}^r \bowtie \mathbb{T}^s, q_f \rangle}(lts_{res}) \quad \frac{p(y) \vdash Precedence}{p(y) \vdash \langle q_0, A^r \cup A^s, \mathbb{T}^r \bowtie \mathbb{T}^s, q_f \rangle}(lts_{prec}) \tag{7.7}$$

$$\frac{p(y) \vdash Absence}{p(y) \vdash \langle q_0, A^r, \mathbb{T}^r, q_f \rangle}(lts_{abs}) \quad \frac{p(y) \vdash Existence}{p(y) \vdash \langle q_0, A^r, \mathbb{T}^r, q_f \rangle}(lts_{exist}) \tag{7.8}$$

Comme indiqué dans les règles ci-dessus, chaque type de patron à un système de transition correspondant. Les patrons de réponse et précédence sont définit par le système $\langle q_0, A^r \cup A^s, \mathbb{T}^r \bowtie \mathbb{T}^s, q_f \rangle$ dont A^r et A^s sont les sous ensembles des actions exécuter en partie *Pre* et *Post* respectivement. Quant aux autres types, ils possèdent seulement un ensemble d'action noté A^r.

7.5.2 Type d'occurrence des évènements

Dans cette section, nous allons définir les règles sémantiques relatives aux types d'occurrence des évènements. Chaque type (AN, AllOrdered, AllCombined) indique l'ordre et la méthode dont les évènements doivent se produire. Pour des fins de simplification, nous n'allons pas spécifier la partie d'occurrence des évènements (*Post* ou *Pre*), à la place nous allons définir d'une manière général en utilisant le mot clé (*Clause*).

- **AN :** Soit $AN(a, a')$, un type AN indique que les évènements doivent se produire entre deux états en parallèle. La règle correspondante est la suivante :

$$\frac{OccType(a, a') \vdash AN(a, a')}{\begin{array}{c} q \xrightarrow{a} q' \\ q \xrightarrow{a'} q' \end{array}} \tag{7.9}$$

- **ALLCombined :** La sémantique de l'opérateur AllCombined est définie sur deux évènements comme suit :

$$\frac{OccType(a, a') \vdash AllCombined(a, a')}{\begin{array}{c} q \xrightarrow{a} q' \;\; q \xrightarrow{a'} q'' \\ q' \xrightarrow{a'} q''' \;\; q' \xrightarrow{a} q''' \end{array}} \tag{7.10}$$

- **ALLOrdered :** La règle sémantique relative à ce type d'occurrence est définie sur deux évènements en guise d'illustration. Soit $a, a' \in A^r$ deux évènements ordonnés.

$$\frac{OccType(a, a') \vdash AllOrdered(a, a')}{q \xrightarrow{a} q' \;\; q' \xrightarrow{a'} q''} \tag{7.11}$$

7.5.3 Nombre d'occurrence des évènements

Dans cette section, nous allons définir les règles sémantiques relatives aux nombre d'occurrence des évènements. Chaque mot clé (ExactlyOne, OneOrMore) indique le combien de fois les évènement doivent se produire.

- **ExactlyOne :** Soit ***ExactlyOne*** occurrence of (a), indique que l'évènement a doit se produire seulement une fois entre deux états. La règle correspondante est la suivante :

$$\frac{OccNum(a) \vdash ExactlyOne(a)}{q \xrightarrow{a} q'} \tag{7.12}$$

- **OneOrMore :** cela indique que l'évènement a peut se produire une ou plusieurs fois, qui signifie que deux transitions sont déclenchées par cet évènement comme suit :

$$\frac{OccNum(a) \vdash OneOrMore(a)}{q \xrightarrow{a} q' \quad q' \xrightarrow{a} q'} \tag{7.13}$$

7.5.4 Implémentation en Coq

Comme pour les observateurs, les patrons de propriétés ECDL sont traduit en utilisant des types inductifs et leurs sémantique est traduite en utilisant des prédicats inductifs. Les parties *Clause* d'un patron est définie par le type inductive *PatternClause*. Les Types d'occurrences

```
Inductive PatternClause := |Pre_Clause  : PatternClause |Post_Clause  : PatternClause.
```

sont définies comme suit :

```
Inductive OccurenceType :=
|AllOrdered  : PatternClause -> OccurenceType  -> OccurenceType
|AllCombined  : PatternClause -> OccurenceType  -> OccurenceType
|AN  : PatternClause -> OccurenceType  -> OccurenceType
```

Afin de définir les états correspondantes aux types d'occurrence, nous avons utilisé la notion de fonctions récursives en Coq *"Fixpoint"*. Cette fonction utilise une autre fonction *StateOrdered* qui prend en paramètres le type d'occurrence, la partie concerné du patron (*PatternClause*) et la liste des évènements, et retourne une liste d'états comme suit :

```
Fixpoint EventOccurenceStates(E :OccurenceType)(evt_list :list Event)(PA : PatternClause) :list State :=
match E with
|AllOrdered => ConsState E evt_list PA
|AllCombined => ConsState E evt_list PA
|AN =>ConsState E evt_list PA
end.
```

7.6 Construction de patrons

Pour accomplir cette transformation, nous devons transformer chacune des parties PreClause et PostClause en un système de transition auquel nos règles de composition (définies dans Chapitre 4) seront appliquées pour obtenir un LTS correspondant à un patron sans options. L'ensemble des états qui constituent l'observateur résultant est obtenu en appliquant l'union des états des parties PreClause et PostClause.

Les transitions sont comme les états générés en utilisant des fonctions récursives comme suit :

```
Fixpoint EventOccurenceTransitions(E :OccurenceType)(evt_list :list Event)(PA : PatternClause)
 :list Transition := match E with
|AllOrdered => ConsTrans E StateOrdered evt_list PA
|AllCombined => ConsTrans E StateOrdered evt_list PA
|AN =>ConsTrans E evt_list PA
end.
```

7.7 Règles de traduction

Les règles de traduction des patrons ECDL vers les automates observateurs ont été données dans Chapitre 4. Le principe de la traduction est le suivant : le type d'un patron ECDL dont le nom est "name" (est représenté par un observateur "name" (un patron de "réponse" est représenté par un observateur "réponse"). Une opération de nomination est créée pour chaque étiquette de transition. L'ensemble des étiquettes de transition d'un automate observateur est alors l'union de l'ensemble des étiquettes représentées par les événements et les contraintes de temps. Dans un automate, plusieurs transitions peuvent avoir la même étiquette. Le type de patron correspondant spécifie alors toutes les transitions possibles pour cette étiquette. La traduction des automates est inductive. Les automates correspondants aux différents types de patron considéré sont traduits. Le résultat donne alors les manières d'effectuer une transition dans un sous-automate de l'automate considéré. Ensuite, une pré-condition (contraintes temporelles) et une post-condition (options) sont définis pour chaque transition de l'automate considéré. Une

fois que tous les sous-automates et toutes les transitions sont traduites, les pré-conditions et post-conditions sont rassemblées par étiquette (et parfois transitions supplémentaires).

7.7.1 Implémentation en Coq

Les fonctions de traduction sont implémentées par des fonctions récursives du langage Coq. L'idée de la traduction d'un patron est de traduire l'ensemble des sous-patrons (Pre Clause, Post Clause, ...) ECDL courants puis de rassembler ces traductions. La fonction de traduction principale est une fonction dont l'expression (simplifiée pour des raisons de compréhension) est la suivante :

```
Definition translatePattern(PreC :Occurence_description)(evt_list_Pre :list Event)
(PostC :Occurence_description)(evt_list_Post :list Event)(States :PatternStates) :Observer :=
   {|St :=(PreClause States)++ (PostClause States) ;
                Initial :=Head((PreClause States)++
                        (PostClause States)) ;
                PreSubObsever :=translate PreC Pre evt_list_Pre States ;
                PostSubObserver :=translate PostC Post evt_list_Post States ;
                BeFinal :=BeforLast ((PreClause States)++
                        (PostClause States)) ;
                Final :=Last ((PreClause States)++
                        (PostClause States))
|} .
```

Pour faciliter la lecture, l'expression de la fonction de traduction donnée ci-dessus est une simplification de la véritable fonction de traduction.

La fonction *translatePattern* permet de traduire un patron en un observateur. Seulement les patrons de type réponse sont pris en considération pour le moment. la variable *Pre/PostSubObsever* contient la liste des opérations qui peuvent être exécutées dans un des sous-automates pre et post contenant la liste des opérations correspondante à la traduction des transitions de l'automate. Tandis que l'ensemble des états est défini par la variable *St* en distinguant deux états spéciaux définis par $BeFinal$ et $Final$. La variable *Pre/PostSubObsever* est calculée à partir de la fonction $translate$. Cette fonction traduit chaque clause en se basant sur son type "Pre" ou "Post" (comme montré ci-dessous).

Les fonction "*translate*" traduit l'ensemble des transitions selon le type de l'occurrence de la clause (AllOrdered, AllCombined, AN). Pour chaque transition, une étiquette est accordée. Elle prend en argument le type (Pre/Post) et la description (AN, ...) de la clause ainsi que la liste des évènements, et retourne une liste de transitions.

```
Definition translate(Clause :Occurence_description)(clauseType : ClauseType)(le :list Event)
(states :PatternStates) :list Transition := match Clause with
    |AllOrdered=>ConsTransOrd le
    (match clauseType with |Pre => PreClause states |Post =>PostClause states end)
    |AN=>ConsTransAnPre le states
    |AllCombined=>AllCombinedTrs le
    (match clauseType with |Pre => PreClause states |Post=>PostClause states end)
end.
```

7.7.2 Théorème à prouver

Théorème

Nous cherchons à montrer que si la traduction d'un patron produit un automate observateur, alors le patron traduit simule l'observateur obtenu par la traduction. Un patron de propriété ECDL simule un automate observateur (Figure 7.3) s'il existe une relation $\mathcal{R}$ entre les états du patron et les états de l'observateur. Cette relation est définie d'une façon que s'il est possible d'exécuter une transition de l'observateur à partir d'un état s, et qu'à partir de s, cette transition aboutit à un état $s\prime$, alors à partir de tout état p de patron en relation avec l'état s selon la relation $\mathcal{R}$ il est possible d'exécuter une transition de même étiquette et cette transition aboutit à un état $p\prime$ en relation avec $s\prime$ selon la relation $\mathcal{R}$. Plus formellement, le théorème à montrer est donc le théorème suivant :

Théorème 1 (*Traduction*)

Pour tout patron pt
S'il existe un automate observateur Ob T_{pt} telle que T_{pt} est la traduction de pt, alors, Pour toute étiquette δ :

$$\forall (s, s' \in State_{Ob})(p \in State_{pt}).\mathcal{R}(p, s) \wedge s \overset{\delta}{\longrightarrow}_{Ob} s'$$
$$\exists (p' \in State_{pt}).p \overset{\delta}{\longrightarrow}_{pt} p' \wedge \mathcal{R}(s', p')$$

Pour prouver ce théorème, il est nécessaire de définir la relation entre les états d'un observateur et les états de patrons ECDL. On peut dire qu'un état d'un patron est équivalent à un état d'un observateur de même type si la valeur associée à la transition entrante et sortante de cet état, sa position (Pre Clause, Post Clause, ...) en se basant sur la fonction de détermination sont équivalents à ceux de l'état courant. Plus formellement, soit un automate observateur $Ob = \langle S, s_{init}, \Sigma, F, Type, X \rangle$ dans un état $s = \langle t_{in}, t_{out}, t_l, pos \rangle$ tel que : t_{in}, t_{out} représente la/les transition(s) entrante(s), respectivement, sortante(s) de l'état s, t_l est la valeur de la transition représentée par l'étiquette l et enfin pos indique la position de l'état (*init*, *final*, *inter*, ou s'il

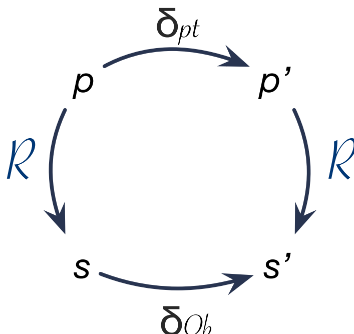


FIGURE 7.3 – Relation de simulation

appartient à la Pre Clause, ou Post Clause).

- On dit qu'un état $p \in State_{pt}$ est équivalent à l'état s selon $\mathcal{R}$ s'il existe un état s' équivalent à l'état p' tel que :
  $pos(p) \sim pos(p') \equiv pos(s) \sim pos(s')$,
  Nombre(t_{in}(s)) = Nombre(t_{in}(p)) et Nombre(t_{out}(s)) = Nombre(t_{out}(p)),
  soit t_l une transition qui relie s, s' avec la valeur l écrite sous forme $t_l\langle s, l, s'\rangle$, $\exists t_l'\langle p, l', p'\rangle$ tel que $l = l'$.

Preuve

Le langage de Coq (appelé Gallina) est un langage mathématique de haut niveau. Vu comme un langage de programmation, Gallina est un langage fonctionnel typé de manière dépendante, tandis que vu comme un système logique, Gallina est une théorie des types intuitionniste d'ordre supérieur qui permet de formuler directement des théorèmes.

Théorème 1 se traduit donc directement dans ce langage. La relation entre les états des automates observateur et les états des patrons de propriétés est modélisée par un prédicat inductif du type suivant :

```
Inductive stateAreEquivalent  : StatePt  -> StateOb ->  PatternType  ->  Prop
```

L'objectif de cette section est de présenter un aperçu sur le schéma général de la preuve du théorème de correction de la traduction. La lecture d'une preuve Coq est peu évidente et ne présente pas beaucoup d'intérêts. Nous présentons donc dans cette section le schéma d'induction adopté pour cette preuve, les principaux lemmes requis et enfin le schéma permettant de démontrer le théorème principal. Pour des raisons de simplification de lecture, les théorèmes sont présentés dans un langage plus naturel que Coq, mais qui conserve les mots clés de Coq.

Plusieurs lemmes sont nécessaires pour montrer le théorème de correction de la traduction.

- ***Lemmes*** : On montre d'abord que les fonctions de détermination des états et plus précisément *initDesignation* et *finalDesignation* qui permettent de désigner les états spéciaux (initial, final) sont correctes. Pour montrer que la fonction *finalDesignation* est correcte, il faut montrer que pour tout état p d'un patron pt et pour tout état s de l'observateur correspondant Ob équivalent à l'état p, si le prédicat obtenu par la fonction *finalDesignation* est vrai dans l'état p, alors l'état s est final pour l'automate Ob. Ce lemme se traduit plus formellement de la manière suivante :

  Lemme 7.7.1 (finalDesignation)**.**
  Pour tout observateur Ob,
  Soit $pred = finalDesignation\ Ob$,
  Pour tout p du patron pt et pour tout état s de Ob,

  $$p \vdash pred \wedge \mathcal{R}(p,s) \implies final(s).$$

  Suivant la même logique, nous devons montrer que la fonction $stateDetermination$ qui génère les états d'un observateur et attribut à chaque automate son état initial suivant la règle R_1 (Chapitre 5) est correcte. Pour cette raison, il est nécessaire de montrer si la substitution générée par R_1 à partir d'un patron pt permet d'atteindre un état p' à partir de l'état p ; alors il existe un état s' équivalent à l'état p' et que cet état s' est l'état initial de Ob, soit le lemme suivant :

  Lemme 7.7.2 (initDesignation)**.**
  Pour tout état p, p' du patron pt tel que :

  $$p \xrightarrow{l} p' \wedge p = init(pt),$$
  $$\exists (s' \in State_{Ob}).\mathcal{R}(p',s') \wedge s \xrightarrow{l} s' \implies s' = init(Ob)$$

  A partir de ces deux lemmes, nous pouvons montrer que la détermination des états spéciaux est correcte, puis par induction que la fonction qui traduit une liste des états *translateAllStates* est correcte. De la même manière la fonction de traduction des transitions peut être prouver.

  En se basant sur les preuves précédentes, nous pouvons montrer que la fonction de génération des transitions est correcte. La fonction *TransitionDef* génère une liste de transitions à partir d'une liste des états et une liste des évènements d'un patron. Afin de prouver que cette fonction est correcte, il convient de montrer que pour toute transition t exécutée dans un patron pt, il est possible d'exécuter une transition dans l'automate observateur résultant de la traduction portant le même label de t soit le lemme suivant :

  Lemme 7.7.3 (TransitionDef)**.**
  Pour tout patron pt,
  Soit $trs = TransitionDef(States, Events)$,

Pour tout état p, p' *du patron* pt,
Pour tout état s *de l'observateur* Ob *résultant de la traduction,*

$$p \xrightarrow{l}_{trs} p' \wedge \mathcal{R}(p, s) \implies \exists(s' \in State_{Ob}).\mathcal{R}(p', s') \wedge s \xrightarrow{l}_{Ob} s'$$

Pour terminer, nous allons vérifier que la fonction *CombinePatternParts* est correcte. En effet, afin de montrer que celle-ci est correcte, nous montrons que s'il est possible d'exécuter une transition dans l'observateur obtenu par traduction de patron, c'est qu'il était possible d'exécuter cette transition soit dans la Pre clause , soit dans la Post clause du patron, soit le lemme suivant :

Lemme 7.7.4 (CombinePatternParts)**.**
Pour toute clause de patron p_1 *et* p_2,
Soit $pt = CombinePatternParts\ p_1\ p_2$,
Pour tout état s *et* s' *d'un observateur* Ob *obtenu par traduction de* pt,

$$s \xrightarrow{l} s' \wedge \mathcal{R}(s, p) \wedge \mathcal{R}(s', p') \implies p \xrightarrow{l}_{p_1} p' \vee p \xrightarrow{l}_{p_2} p'$$

- **Preuve de théorème** : grâce aux lemmes développés plus haut, nous pouvons montrer le théorème de correction de traduction. En effet, ce théorème se définit sous forme d'une proposition dont l'idée est la suivante : s'il est possible d'exécuter une transition dans le patron ECDL, il est également possible d'exécuter une transition portant le même label dans l'observateur généré par la traduction. De manière plus formelle, cette proposition s'écrit comme suit :

  Proposition 7.7.5 (Hypothèse de traduction)**.**
  Soit un patron pt,
  Soit $trans$ *le résultat de traduction de* pt *en un observateur* Ob,
  Pour tout état s, s' *de l'observateur* Ob,
  Pour tout état p, p' *du patron* pt,
  Nous avons :

$$s \xrightarrow{l}_{trans} s' \wedge \mathcal{R}(s, p) \implies \exists(p' \in State_{pt}).\mathcal{R}(s', p') \wedge p \xrightarrow{l}_{pt} p'$$

  Il s'agit de prouver que s'il est possible de traduire un patron en un automate observateur et qu'il est possible d'exécuter une transition dans cet observateur, il est possible d'exécuter une transition avec le même label dans le patron de départ. Soit l_{trans} la liste des transitions résultant de la traduction d'un patron pt en un automate Ob. Nous savons

que l_{trans} est obtenue en fusionnant la liste des transitions lst_1 obtenue en traduisant la clause Pre de patron pt avec la liste lst_2 obtenue en traduisant la clause Post de pt. En utilisant *Lemme 7.7.4* (CombinePatternParts), nous savons que s'il est possible d'exécuter une transition dans l_trans, c'est parce qu'il est possible d'exécuter cette transition soit dans lst_1 soit dans lst_2. En utilisant *Lemme 7.7.3* (TransitionDef), nous montrons qu'il est possible d'exécuter une transition avec le même label dans pt.

7.8 Conclusion

Au cours des sections précédentes, nous avons présenté la sémantique des automates observateurs et des patrons ECDL. En plus, nous avons brièvement résumé les principaux lemmes et le schéma général de la preuve effectuée. Cette preuve a été réalisée à l'aide de l'assistant de preuve Coq. La traduction en Coq de la sémantique et la preuve globale de tous ces théorèmes et lemmes représentent plus de 2000 lignes de Coq.

Cependant, ce chapitre ne présente qu'une preuve partielle de notre outil de traduction. En effet, nous ne présentons que la preuve de l'exactitude de la traduction des automates de type Réponse. Pour valider complètement l'outil de traduction pour les patrons ECDL, il faudrait alors étendre la traduction et sa preuve à tous les autres patrons ECDL.

Quatrième partie

Expérimentations, conclusion et perspectives

Chapitre

8

Exemple industriel

« Knowledge is power. »

— Francis Bacon

Sommaire

8.1 Introduction **132**

8.2 Vérification formelle des véhicules autonomes **132**

8.2.1 Concept 132

8.2.2 Vérification des bateaux autonomes 133

8.3 Les règles de COLREG **134**

8.3.1 Généralités 134

8.3.2 Catégories des règles COLREG 134

8.4 Prévention de collision **135**

8.4.1 Machine à états 136

8.4.2 Expérimentations sur la machine d'état 141

8.4.3 Patrons ECDL pour les règles de COLREG 142

8.5 Discussions **148**

8.1 Introduction

Dans ce chapitre nous allons présenter un exemple d'application de nos propositions dans cette thèse. Durant l'année 2020 j'ai eu l'occasion d'effectuer des services de consultation pour une startup américaine qui développe des technologies autonomes avancées pour le secteur maritime notamment la prévention des collisions et d'autre services. J'ai contribué au développement d'une ligne de produits de systèmes de contrôle et de navigation autonomes pour les bateaux autonomes. Pour ces raisons, notre choix de l'exemple industriel s'est porté sur la vérification des systèmes (algorithmes) des prévention de collision des bateaux autonomes.

8.2 Vérification formelle des véhicules autonomes

8.2.1 Concept

La conception de systèmes de contrôle embarqués fiables présente les mêmes difficultés que la conception de systèmes de contrôle et de systèmes informatiques distribués. Les anomalies de conception de ces systèmes peuvent provenir d'interactions imprévues entre les sous-systèmes de calcul, de communication et de contrôle.

Les véhicules autonomes (AV), tels que les voitures sans conducteur, les robots, les bateaux et les équipements de construction, sont de plus en plus prometteurs, ce qui suscite un grand intérêt dans l'industrie et le monde universitaire. La sécurité du fonctionnement des véhicules est la préoccupation la plus importante, car ces systèmes doivent se déplacer et accomplir leurs tâches sans risquer d'entrer en collision avec d'autres objets statiques ou dynamiques dans l'environnement, tels que de gros rochers, des humains et d'autres machines mobiles.

Des algorithmes tels que A*[114], RRT [97] et Theta*[46] sont capables de faire naviguer le AV en évitant les obstacles statiques pour atteindre leur destination. Cependant, lorsqu'ils rencontrent des obstacles dynamiques qui peuvent apparaître et se déplacer arbitrairement dans l'environnement, ces algorithmes ne suffisent pas à éviter les collisions et doivent être complétés par des algorithmes tels que ceux basés sur les champs de flux dipolaires [130] ou l'approche par fenêtre dynamique [64, 68], qui sont capables de contourner les obstacles dynamiques. Bien que de nombreux algorithmes destinés à éviter des collisions aient été proposés par des chercheurs et des praticiens ces dernières années, peu d'entre eux ont été formellement vérifiés. Pourtant, la vérification formelle est un outil très important pour découvrir des problèmes au début de la conception d'un algorithme ou pour prouver l'absence des anomalies avant que les algorithmes ne soient déployés dans des systèmes réels.

Parallèlement au domaine des véhicules terrestres et aériens autonomes qui progresse très activement, l'avènement des bateaux autonomes a fait apparaître de nouveaux problèmes de

recherche et de technologie découlant des spécificités de la navigation maritime. Les bateaux autonomes sont censés naviguer en toute sécurité et éviter les collisions en suivant les règles de navigation appelées COLREG [6].

8.2.2 Vérification des bateaux autonomes

La demande de bateaux sans équipage a augmenté dans le but de réduire les coûts d'exploitation en raison d'un équipage minimal à bord et de la sécurité en mer, mais aussi de promouvoir le travail à distance. On s'attend à ce que les navires autonomes prennent de plus en plus de décisions en fonction de leur situation actuelle en mer, sans supervision humaine directe. Cela signifie qu'un bateau autonome devrait être capable de détecter d'autres navires et de faire les ajustements nécessaires pour éviter les collisions en respectant les règles du trafic maritime. Cependant, l'existence d'un "capitaine virtuel" du centre de contrôle à terre (SCC) est toujours nécessaire pour effectuer des opérations critiques ou difficiles [6]. D'autre part, il y a un besoin toujours d'une réévaluations et confirmation lorsque des commandes incohérentes ou corrompues sont détectées par le système embarqué.

L'un des problèmes de sécurité les plus critiques dans le développement des véhicules autonomes est leur mauvaise performance dans des conditions météorologiques défavorables, comme la pluie et le brouillard, en raison de la défaillance des capteurs [95]. Cependant, lors de la modélisation des spécifications maritimes, nous ne tenons pas compte des imprécisions des capteurs et des éventuelles erreurs de transmission, car il existe des mesures standard de conception de la redondance des capteurs et de correction des erreurs appliquées sur les navires modernes pour s'assurer que les navires se remarquent mutuellement en temps voulu. Pour garantir la sécurité, un navire est en mesure de communiquer avec un autre ou avec le centre à terre par radio VHF, services par satellite, etc.

Pour un protocole de navigation sans ambiguïté, l'organisation maritime internationale (IMO) [84] a publié des règles de navigation que doivent suivre les navires et autres bâtiments en mer, appelées Convention sur les règlements internationaux (COLREG).

Notre but dans ce chapitre est d'adapter la modélisation formelle sur des systèmes de bateaux autonomes pour vérifier et synthétiser leurs navigation sûre, tout en planifiant la route optimale et en programmant les manœuvres selon les règles COLREG.

6. Convention sur le Règlement international de 1972 pour prévenir les abordages en mer : https://www.imo.org/en/About/Conventions/Pages/COLREG.aspx

8.3 Les règles de COLREG

8.3.1 Généralités

Diverses études ont été menées sur la navigation autonome des bateaux conformément aux règles de COLREG. Parmi celles-ci, on peut citer la logique floue [98] et la programmation par intervalles [26]. Cependant, les approches précédentes ne couvrent pas la vérification de la sécurité de navigation. En outre, le comportement (potentiellement) non déterministe des navires autonomes, les retards de communication, les défaillances des capteurs ne sont pas pris en compte dans leurs modèles.

La conformité avec la prévention des collisions, dans son sens le plus général consiste à manœuvrer son navire de manière à interagir correctement avec un autre navire ("contact") pour une géométrie initiale donnée.

Dans le système de règles COLREG, des bateaux spécifiques adoptent l'un des deux rôles nommés "*stand-on*" ou "*give-way*" en fonction de leurs conditions géométriques, de propulsion et de fonctionnement les uns par rapport aux autres.

Un bateau dit "*stand-on*" est généralement requis à maintenir son cap et sa vitesse en accord avec les exigences de navigation raisonnables, comme s'il n'y avait pas de contact (par exemple, ralentir pour embarquer un pilote). Le navire "*give-way*" est celui qui est requis pour céder le passage.

Pour éviter la divergence entre les revendications de conformité et les performances réelles, le champ d'application peut être compartimenté pour permettre la quantification et la certification dans des domaines d'exigences similaires. Les règles COLREG peuvent donc être séparées en catégories pour permettre à un véhicule de démontrer sa conformité avec les sous-ensembles COLREG appropriés.

8.3.2 Catégories des règles COLREG

Les catégories proposées par [132] sont énumérées dans Tableau 8.1 et comprennent :

- les exigences générales des navires, y compris les règles 1 à 3
- la conduite des navires dans toutes les conditions de visibilité, y compris les règles 4 à 8
- les cas particuliers pour les chenaux et les dispositifs de séparation, y compris les règles 9-10
- la conduite de deux voiliers en vue l'un de l'autre et fonctionnant selon la règle 12
- les rencontres générales de navires, y compris la conduite des navires en vue l'un de l'autre et opérant selon les règles 13-17

- les responsabilités des navires en vue l'un de l'autre telles que comme indiqué dans les règles 11 et 18
- la conduite des navires par visibilité réduite en vertu de la règle 19
- les feux et les formes exigés des navires en vertu des règles 20 à 31
- signaux sonores et lumineux exigés des bateaux en vertu des règles 32-37
- les communications entre véhicules, notamment l'envoi, la réception et l'interprétation de messages destinés à d'autres navires ou à des tiers.
- performance cumulative des catégories ci-dessus pour assurer une approche holistique satisfaisante de la sécurité de la navigation et de la prévention des collisions

I	Règles générales (Règles 1-3)
II	Conduite générale des navires (Règles 4-8)
III	Régimes spéciaux de circulation (Règles 9-10)
IV	Navigation à la vue d'un autre navire à voile (règle 12)
V	Rencontres de navires en vue d'un autre (règles 13-17)
VI	Responsabilités en vue d'un autre navire (règles 11, 18)
VII	Visibilité réduite (règle 19)
VIII	Feux et formes (Règles 20-31)
IX	Signaux sonores et lumineux (Règles 32-37)
X	Communications inter-véhicules
XI	Performance cumulée incluant les coutumes locales

Tableau 8.1 – Catégories des règles COLREG selon leurs champs d'application

La catégorisation des règles permet à un concepteur de revendiquer la conformité dans une ou plusieurs catégories (par exemple, les exigences de manœvre des navires à moteur). En définissant le champ d'application des règles et en démontrant des niveaux de conformité quantifiables dans chaque catégorie, les concepteurs d'algorithmes de prévention des collisions des véhicules autonomes peuvent articuler plus précisément leurs contributions à la littérature. Il convient de noter que l'évaluation dans le cadre d'une catégorie peut reposer sur la conformité d'une autre catégorie avec certaine mesure. Par exemple, étant donné que la catégorie **II** comprend le maintien d'une veille, la détermination de la vitesse de sécurité, la détermination du risque de collision et la prise de mesures pour éviter une collision, elle influence fortement l'évaluation des catégories **III-VII**.

8.4 Prévention de collision

Au fil des siècles, la navigation des bateaux a traditionnellement été entièrement réalisée par des êtres humains. Toutefois, la technologie maritime vient aujourd'hui en aide aux équipages maritimes pour minimiser les erreurs de navigation. Dans un avenir proche, ces technologies

formeront un système de navigation intelligent totalement autonome qui sera capable de déterminer des trajectoires quasi optimales pour éviter les collisions entre navires.

La navigation autonome des navires peut être divisée en deux grands domaines de recherche : la prévention des collisions et le maintien de la trajectoire. Cette étude se concentre sur le premier concept, la prévention des collisions.

Il convient de noter que la plupart des travaux de recherche sur la prévention de collision intègrent les règles COLREG dans leurs algorithmes. En outre, la plupart des articles utilisent couramment la terminologie suivante :

- ***Own-ship*** : le navire à naviguer, dans le reste du document est référé comme "*OS*"
- ***Target-ship*** : le navire à éviter, dans le reste du document est référé comme "*TS*".

Dans notre étude, nous considérons seulement les règles COLREG de prévention de collision catégorie **V** dans Tableau 8.1 (règles **13-17**).

Les règles **13**, **14** et **15** énoncent ces trois situations, respectivement ("dépassement", "face à face" et "croisement"). Figure 8.1 illustre ces situations et la manière dont elles doivent être résolues. Les règles **16** et **17** décrivent les actions à entreprendre par les navires qui doivent respectivement "céder le passage" ou "maintenir sa trajectoire".

8.4.1 Machine à états

Afin de modéliser les différents règles de COLREG pour la prévention de collisions, nous avons d'abord proposé une machine à états afin de déterminer quelle règle est efficace pour chaque obstacle à proximité du navire (voir Figure 8.2).

(i) ***États*** : La machine à états générale contient les états suivants :

- *Unsafe* : Cet état est déclenché si l'obstacle (navire cible "*TS*") est trop proche ou se comporte de manière imprévisible. C'est l'état initial de la machine à états, car nous supposons que le navire (*OS*) n'est initialement pas sûr de pouvoir couvrir tous les scénarios possibles et éviter les collisions potentielles.
- *Safe (SfS)* : Désigne l'état de sécurité, ce qui implique que l'obstacle ne nécessite l'application d'aucune des règles du COLREG.
- *HdO* : signifie "situation de face". Cela implique que la règle **14** du COLREG s'applique pour éviter l'obstacle.
- *XsT* : Situation de croisement ou traversée , où le navire propriétaire (*OS*) est un navire qui privilégié (maintien sa vitesse et son cap).
- *XgV* : Situation de traversée où le *OS* est non privilégié et par conséquent doit céder le passage (règle **15** de COLREG).
- *OvT* : Situation de dépassement (règle **15**).

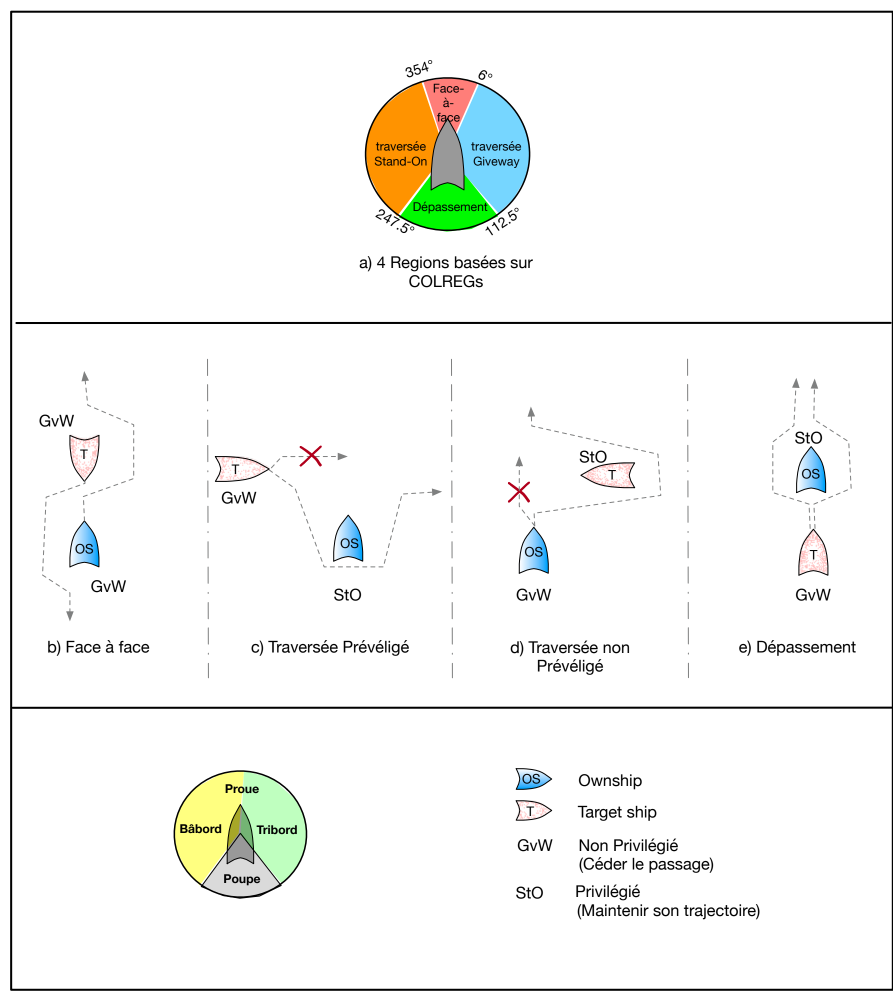


FIGURE 8.1 – Illustration des différentes situations d'abordage de deux navires et les actions à prendre

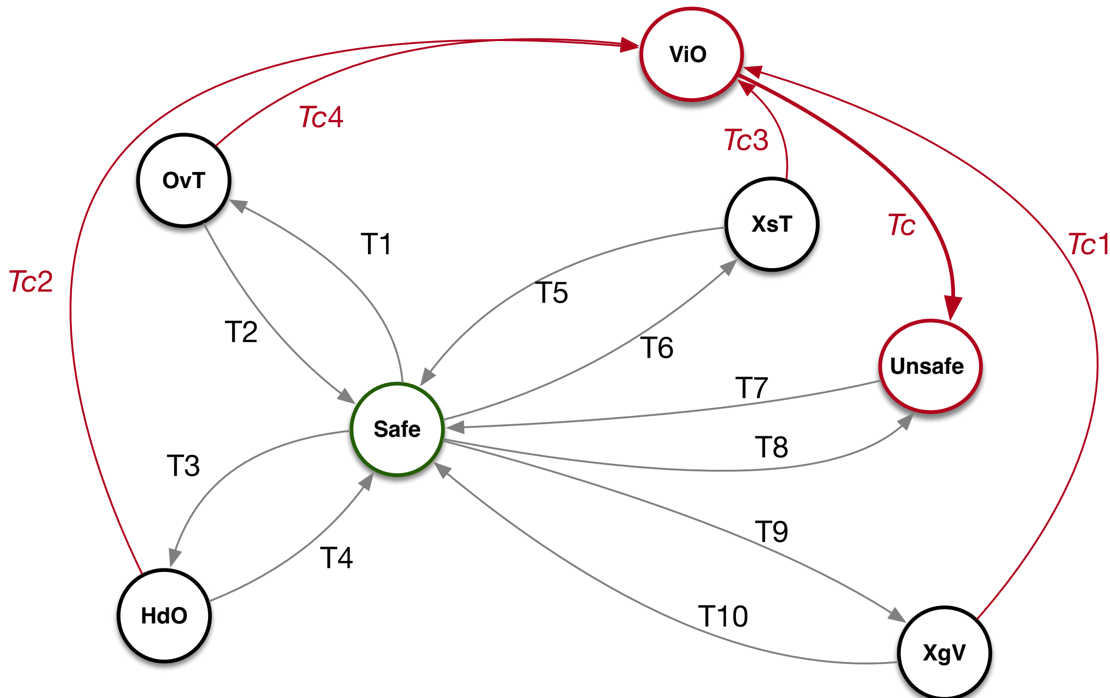


FIGURE 8.2 – Interprétation des règles ECDLs de prévention de collision par une machine à état

- *ViO* : signifie état de violation, cet état est déclenché si l'une des règles COLREG est violée. Par exemple, si le navire propriétaire se trouve dans une situation de face à face, il doit suivre les instructions mentionnées dans Figure 8.1.(b), sinon l'état de violation est déclenché.

(ii) ***Transitions*** : T_i où $i \in [1, 10]$: représentent les transitions qui doivent aller soit vers l'état sûr (*safe*), soit depuis cet état. Cela signifie que lorsque la machine à états détermine qu'une situation COLREG (ou non sûre) existe par rapport à un obstacle, elle ne permettra pas le passage à un autre état sans que la situation soit d'abord jugée sûre. Nous pourrons dire qu'il devrait être capable de passer d'un état spécifique à un autre, ce qui est un point intéressant, qui devrait faire l'objet d'un travail futur. Les transitions Tc et Tc_i où $i \in [1, 4]$ sont les transitions menant à l'état mauvais (unsafe) depuis l'état mauvais temporaire (*ViO*). Depuis l'état de violation (ViO) nous pourrons passer à un état *safe*, mais afin d'avoir un système autonome sûr, il faut prendre une violation comme un état menant directement à l'état de rejet.

Pour contrôler les transitions entre les différents états, nous combinons le temps et la distance au *CPA*, et une interprétation géométrique des situations ECDLs (Figure 8.2).

(iii) ***Facteurs déclencheurs :***

- *CPA (Closest Point of Approach)* : est un facteur crucial pour la prévention des collisions, et il est largement utilisé dans l'industrie maritime pour obtenir le domaine du navire et évaluer le risque de navigation. Compte tenu de la vitesse et du cap actuels d'un navire *OS* et d'un obstacle (ou navire cible) *TS*, le T_{CPA} et le D_{CPA} sont deux facteurs

clés pour calculer le risque de collision, où le T_{CPA} décrit le temps jusqu'au point où les deux navires sont les plus proches, et le D_{CPA} est la distance jusqu'à l'obstacle à ce point. Soit p_A et p_B les vecteurs de position des navires A et B respectivement, et v et v' sont leurs vecteurs de vitesse. Le temps jusqu'au CPA est calculé suivant le même raisonnement de [94] et [65] comme suit :

$$Tcpa = \begin{Bmatrix} 0 & if \, \|v - v'\| \leq \epsilon \\ \frac{(p_A - p_B).(v - v')}{\|v - v'\|^2} & \textbf{otherwise} \end{Bmatrix} \tag{8.1}$$

Où $\epsilon > 0$ est une petite constante afin d'éviter la division par zéro dans le cas où la vitesse relative entre le vaisseau et l'obstacle est nulle.

Étant donné T_{CPA} [7], la distance entre A et B est calculée comme suit :

$$d_{CPA} = \|(p_A + t_{CPA}\mathbf{v}) - (p_B + t_{CPA}\mathbf{v'})\| \tag{8.2}$$

Afin de définir les déclencheurs menant à l'un des états suivant $\{HdO, XsT, OvT, ...\}$, deux critères majeurs doivent être respectés, les paramètres CPA et la situation géométrique. Ainsi, pour que l'obstacle représente un risque, les paramètres T_{CPA} et D_{CPA} doivent se situer dans des limites réglables.

- *Situations géométriques* : Il se base sur la position relative, le relèvement et la trajectoire de l'obstacle par rapport au navire *OS*, En pratique, l'utilisation d'un domaine de navire dans une situation de rencontre peut être combinée avec l'un des quatre critères de sécurité tels qu'ils sont énoncés dans [125] comme suit :
  - Le domaine de navire (*OS*) ne doit pas être violé par le navire (*TS*),
  - Le domaine de navire (*TS*) ne doit pas être violé par le navire (*OS*),
  - aucun des domaines des navires ne doit être violé (conjonction des deux premières conditions),
  - les domaines des navires ne doivent pas se chevaucher. Leurs zones doivent rester mutuellement exclusives (l'espacement effectif sera la somme des espacements résultant de chaque domaine).

Comme Figure 8.3 montre, l'interprétation géométrique d'un obstacle est associé avec différents types de régions d'abordage tout dépend de son relèvement et sa position par rapport à l'OS. La trajectoire de navire OS est désignée par $\mathscr{C}$ (Voir Figure 8.3).

La décision de situation géométrique est effectuée sur la base de la région de relèvement relatif dans laquelle se trouve l'obstacle, et de sa région de cap. Par exemple, si l'obstacle est situé dans la région comprise entre ($\pi/8$, et $\pi/2$), et que le cap de l'obstacle est dans la zone jaune, le type de rencontre résultant pour l'obstacle sera le dépassement.

7. Un T_{CPA} négatif indique que le CPA a déjà eu lieu et que le TS s'éloigne de l'OS

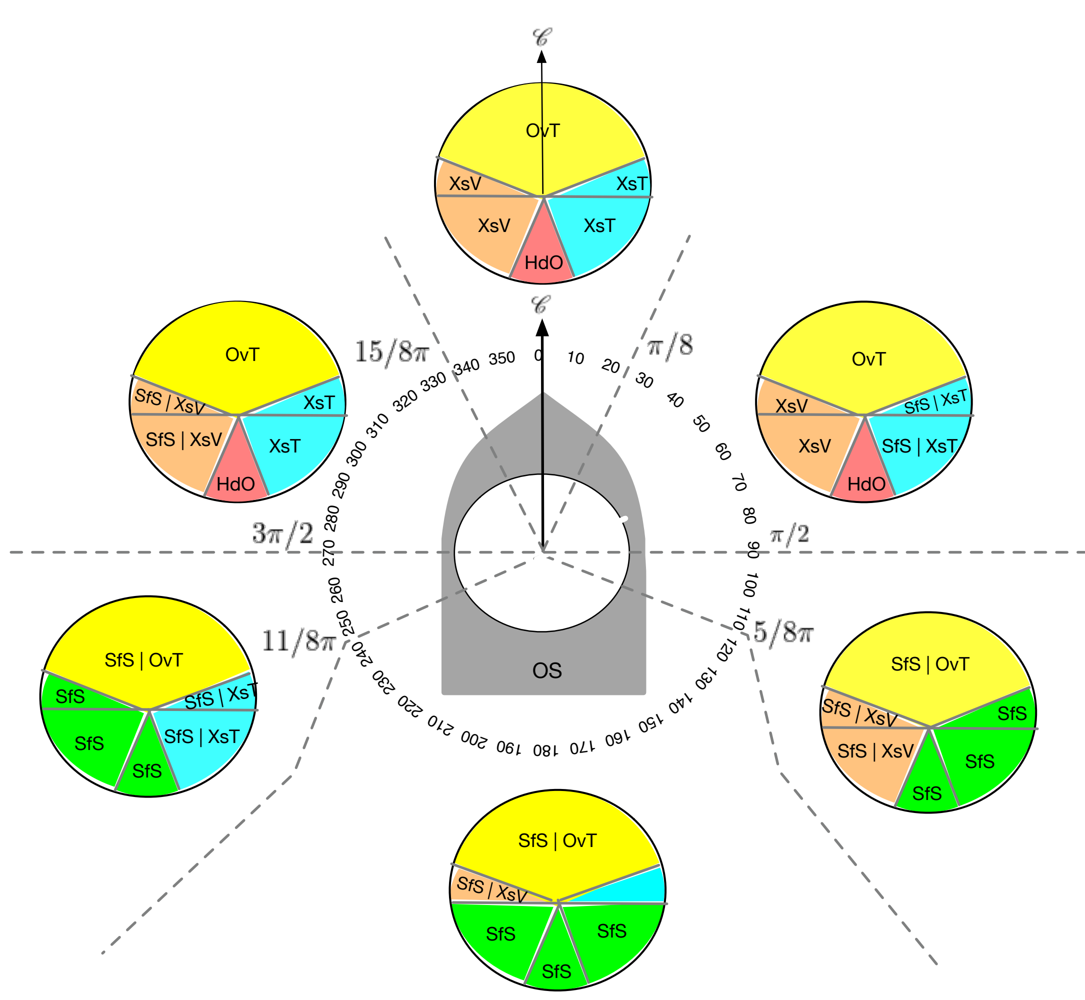


FIGURE 8.3 – Interprétation géométrique de COLREG

8.4.2 Expérimentations sur la machine d'état

La machine à états présentée dans ce mémoire est vérifiée à l'aide de données d'identification AIS réelles recueillies dans le port de Boston. Les simulations ont été effectuées sur des données réelles rassemblées dans une base de données SQL qui comprend près de 298 navires. Figure 8.4 montre le nombre de visites des états de rencontre (Frontale, croisements bâbord/tribord, dépassement). D'après ces résultats, l'état de rencontre frontale est le moins visité dans la machine à états, ce qui signifie que cette situation se produit moins souvent. Cependant, le dépassement est la situation la plus fréquente.

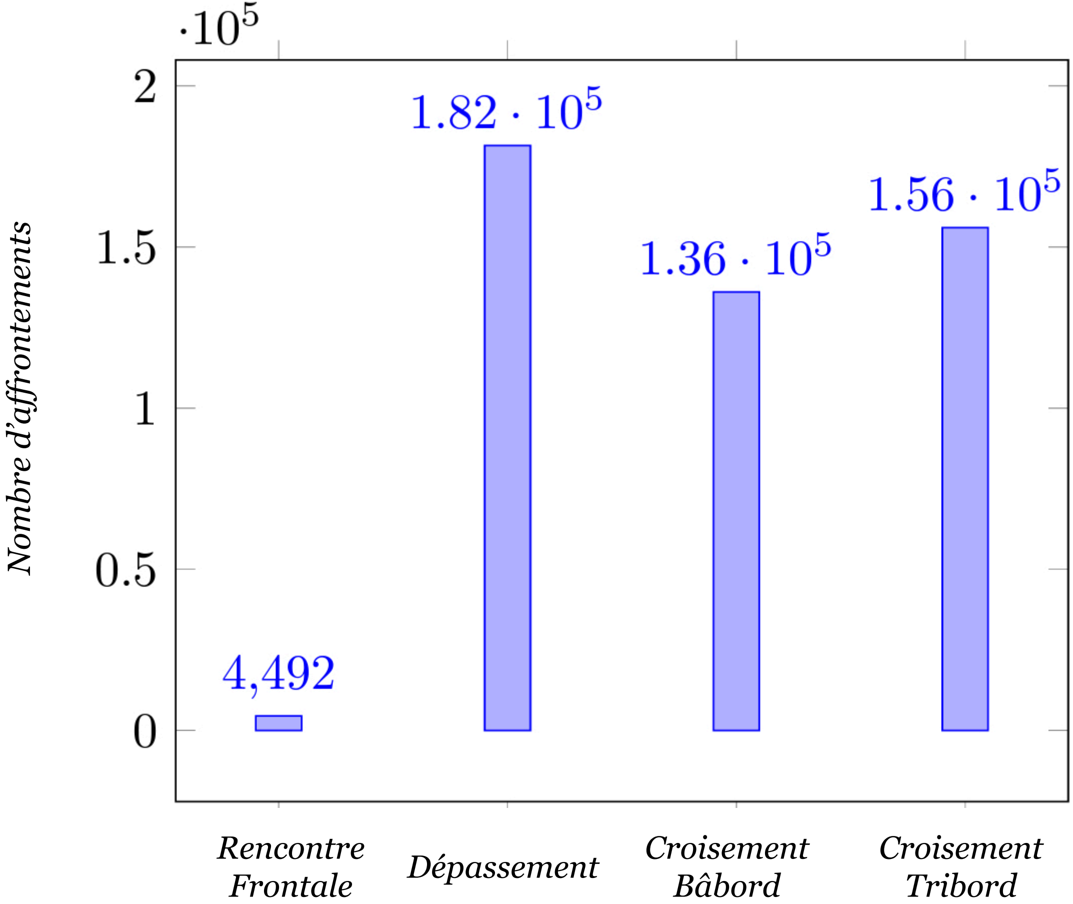


FIGURE 8.4 – Statistiques sur les types de rencontre des navires

8.4.3 Patrons ECDL pour les règles de COLREG

Dans cette section, nous formalisons les règles de navigation maritime COLREG qui prennent en compte l'évitement des collisions entre les navires. Les règles de direction et de navigation sont pertinentes pour spécifier les manœuvres de prévention des collisions et sont cruciales pour la formalisation des patrons ECDL pour les navires autonomes. Nous supposons les conditions suivantes pour notre formalisation des patrons représentant les règles de direction et de navigation :

- La profondeur de l'eau est suffisante pour tous les navires et ne limite pas les manœuvres possibles.
- Les chenaux et les marques de trafic maritime sont absents.
- Tous les bateaux sont motorisés.

Cependant, notre formalisation peut facilement être étendue à d'autres types de navires en intégrant une méthode (par exemple, un automate) qui sélectionne les règles applicables à la combinaison actuelle de types de navires. Nos principales contributions sont les suivantes :

- Au meilleur de notre connaissance, nous présentons la première formalisation COLREG en utilisant les patrons de propriétés temporelles et des automates observateurs.
- Nos propriétés sont paramétrables et utilisables pour des règles de trafic maritime supplémentaires.

8.4.3.1 Propriétés d'accessibilité

Comme mentionné dans Chapitre 5, Section 5.3, deux propriétés sont à vérifier dans cette catégorie : la propriétés d'accessibilité qui exige que chaque exécution se termine dans un bon état ("*Success*"). Contrairement à la propriété de non accessibilité qui vérifie qu'un mauvais état "*Reject*" donné n'est pas atteignable, dans aucune des exécutions possibles.

- *Accessibilité* : Cette propriété n'est pas présentée dans ce chapitre. Les propriétés de non-accessibilité qui se terminent par un état de rejet nous intéresse plus dans cet exemple.
- *Non-Accessibilité* : L'exclusion mutuelle est généralement considérée comme une propriété de non-accessibilité : le mauvais état est défini comme un état où les navires se chevauchent, c'est à dire qu'un navire rentre dans la zone de sécurité d'un autre. Cette zone est définie par la notion de T_{CPA} et D_{CPA}.

  Soit OS et TS deux navires qui sont proches l'un de l'autre et qui doivent éviter tout risque de collision. Soit T_{OT} le temps actuel qui sépare les deux navires OS et TS, et D_{OT} est la distance actuelle séparant les deux navire. Pour que les deux navires évitent une collision, ils doivent respecter le T_{CPA} et le D_{CPA}. Alors que le D_{CPA} est le point où la distance à l'obstacle est minimale, le point critique est le point où la distance à l'obstacle est inférieure à la distance critique d_{crit}. Cette distance critique décrit le seuil d'une

distance minimale à l'obstacle définie par l'algorithme de *COLAV*. Le temps jusqu'au point critique t_{crit} peut être calculé en résolvant l'équation suivante :

$$\|(p_A + t_{crit}\,\boldsymbol{v}) - (p_B + t_{crit}\,\boldsymbol{v'})\| = d_{crit} \tag{8.3}$$

Description :
Chaque occurrence d'une action d'approche du CPA, *OS* doit respecter le temps et la distance critiques.
ECDL :   **AN**   Each occurence **of** CPA_Approach   Exists   **Between** T_{CPA} **and** t_{crit} $d_{TO} <= d_{crit}$ must never occur

Tableau 8.2 – Patron de Non-Accessibilité

Le transition entre un état donné à un état de rejet ne se produit que si *OS* ne manœuvre pas selon les COLREG, ou si un nouvel obstacle dynamique est détecté et se rapproche dangereusement du navire. Comme dans [65], la distance et le temps critique sont utilisées pour définir le passage à un état mauvais (voir Équation (8.3). La propriété ainsi que l'observateur correspondant sont donnés dans Tableau 8.2. Comme montré dans Figure 8.5, l'observateur commence dans un état initial s_0 en initialisant l'horloge $x_{obs} := 0$. L'action $CPA_{Approach}$ se produit (dans Figure 8.5 représenté comme *a*). Si l'une des conditions (temps ou distance) n'est pas satisfait, l'observateur passe au mauvais état (reject) indiquant un risque de collision.

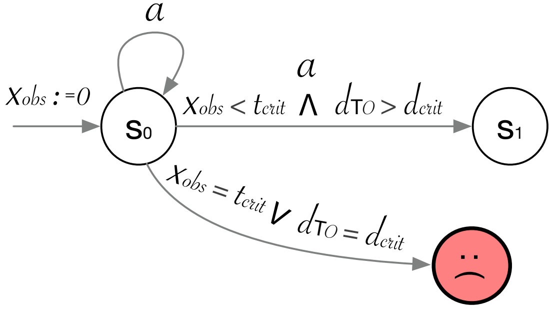


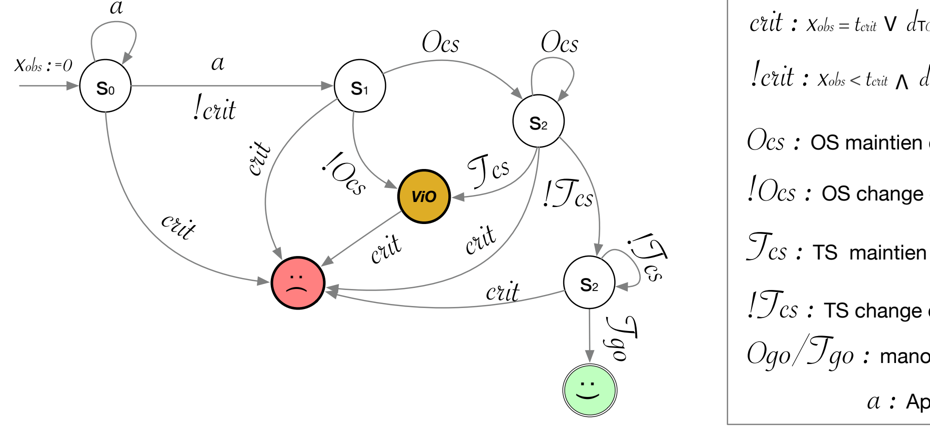
Figure 8.5 – Observateur de Propriété de non-accessibilité de COLREG

8.4.3.2 Propriété de croisement bâbord

Dans le cas de croisement, deux cas sont à considérer, le premier est le cas où le *TS* s'approche de l'*OS* à ***bâbord***, et où l'*OS* et le *TS* ont les mêmes vitesses initiales. Dans ces cas de croisements les navires doivent appliquer la règle 15 des COLREG pour éviter toute collision possible. D'après cette règle, dans un tel scénario de trafic, l'*OS* est la partie passive et n'est pas tenue de changer de cap dans une rencontre avec un stand-on, tandis que le *TS* est la partie active qui a la responsabilité de changer de cap, en traversant par l'arriére de l'*OS* (give a way).

Description :
Pour chaque occurrence d'approche de *TS* à bâbord de *OS*, *OS* doit garder son cap et sa vitesse et par conséquent sa trajectoire, tandis que *TS* doit changer son cap pour traverser par l'arrière de *OS* .
ECDL :   **AN**   Each occurence **of** starboard_Approach   **LeadsTo**   **AllOrdered**   One **or** more occurence **of** OS_keep_course_speed   One **or** more occurence **of** TS_change_course_speed   **Between** T_{CPA} **and** t_{crit} $d_{TO} <= d_{crit}$ must never occur

TABLEAU 8.3 – Patron de croisement bâbord- *Règle 15*



crit : $x_{obs} = t_{crit} \vee d_{TO} = d_{crit}$

!crit : $x_{obs} < t_{crit} \wedge d_{TO} > d_{crit}$

Ocs : OS maintien cap&vitesse

!Ocs : OS change cap/vitesse

Tcs : TS maintien cap&vitesse

!Tcs : TS change cap/vitesse

Ogo/Tgo : manoeuvre avec succès

a : Approach

FIGURE 8.6 – Observateur de règle de croisement bâbord

8.4.3.3 Propriété de croisement tribord

Dans le cas de croisement tribord, le *OS* s'approche de *TS* à ***bâbord***, donc *TS* se trouve dans le tribord de *OS*. D'après cette règle 15, dans un tel scénario de trafic, *TS* est la partie passive et n'est pas tenue de changer de cap dans une rencontre avec un *stand-on*, tandis que le *OS* est la partie active qui a la responsabilité de changer de cap, en traversant par l'arrière de *TS*.

Description : Pour chaque occurrence d'approche de *OS* à bâbord de *TS*, *TS* doit garder son cap et sa vitesse. Tandis que *OS* doit changer son cap et sa vitesse pour traverser par l'arrière de *TS* .
ECDL : **AN** Each occurence **of** tribord_Approach **LeadsTo** **AllOrdered** One **or** more occurence **of** TS_keep_course_speed One **or** more occurence **of** OS_change_course_speed **Between** T_{CPA} **and** t_{crit} $d_{TO} <= d_{crit}$ must never occur

Tableau 8.4 – Patron de croisement Tribord - *Règle 15*

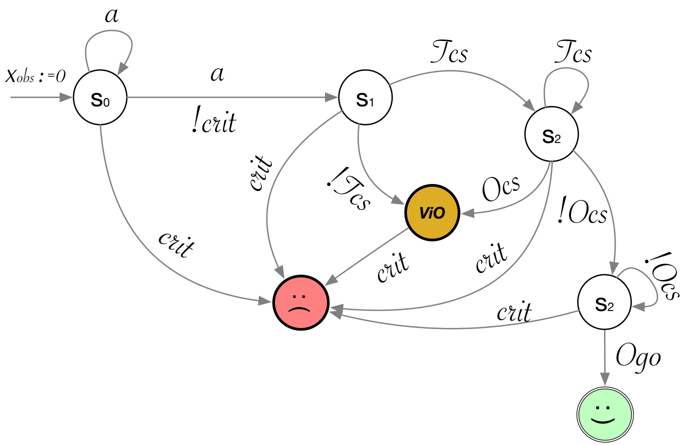


Figure 8.7 – Observateur de règle de croisement tribord

8.4.3.4 Propriété de rencontre frontale "*Head-On*"

Dans ce scénario de rencontre frontale, le TS approche l'*OS* de front. Pour éviter tout risque de collision les deux navires doivent appliquer la règle 14 du COLREG, qui stipule que les

navires doivent se croiser *bâbord* à *bâbord*. Étant donné que les conditions de trafic initiales sont les mêmes, que ce soit du point de vue du *OS* ou du *TS*, et que les caps initiaux des deux navires les placent directement face à face, l'observation de la trajectoire de navigation du seul point de vue du *OS* est suffisante pour étudier l'interprétation de COLREG d'un point de vue différent. Puisque le point de vue du *TS* sur le trafic est simplement une image miroir du point de vue du *OS*, les deux navires manœuvrent conformément à la règle 14 de COLREG.

Description :
Pour chaque occurrence de rencontre frontale de *OS* et *TS*, les deux navires *TS* et *OS* doivent changer leurs caps et vitesses pour se traverser bâbord à bâbord.

ECDL :

```
AN
Each occurence of headOn_Approach
LeadsTo
AllCombined
One or more occurence of OS_change_course_speed
One or more occurence of TS_change_course_speed
Between T_CPA and t_crit
d_TO <= d_crit must never occur
```

Tableau 8.5 – Patron de rencontre frontale (*HeadOn*) - *Règle 14*

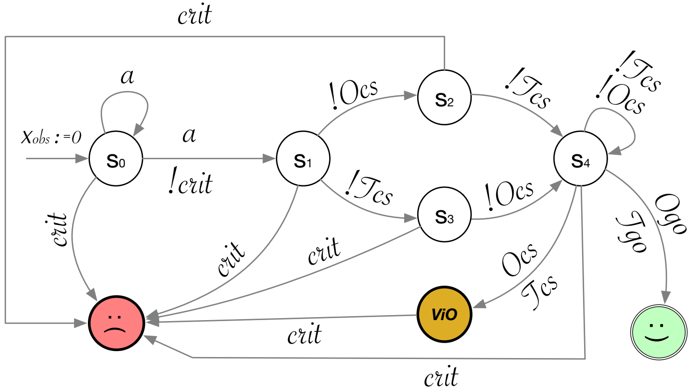


Figure 8.8 – Observateur de croisement frontale (HeadOn)

8.4.3.5 Propriété de rencontre avec un dépassement "*Overtaking*"

Dans le scénario de dépassement, l'*OS* s'approche du *TS* par la poupe du *TS*. Ce scénario permet d'évaluer l'interprétation de règle **13** des COLREG comme patrons ECDL. Afin d'effectuer le dépassement en sécurité, l'OS doit manœuvrer à bâbord ou à tribord pour éviter de

pénétrer dans la zone de sécurité de TS. La vitesse de l'OS doit aussi être supérieure à celle du TS tout au long de dépassement.

Description :
Pour chaque occurrence de rencontre de dépassement de OS et TS, le navire OS doit augmenter sa vitesse et changer son caps pour dépasser TS de bâbord ou tribord. Dans cette propriété, nous ne prenons pas en compte le comportement de TS car elle est la partie passive, et elle n'a pas d'action à effectuer. L'OS est le navire qui doit s'en charger d'effectuer le manoeuvre en toute sécurité.
ECDL :   **AN**   Each occurence **of** Stern_Approach   **LeadsTo**   **AN**   One **or** more occurence **of** OS_change_course_speed   **Between** T_{CPA} **and** t_{crit} d_{TO} <= d_{crit} must never occur

Tableau 8.6 – Patron de dépassement (*Overtacking*) - *Règle 13*

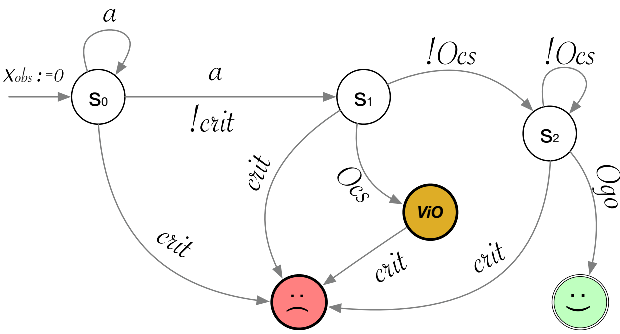


Figure 8.9 – Observateur de croisement avec dépassement (Overtaking)

Remarque. *Les propriétés ci-dessous sont définies d'un point de vue OS, qui signifie que tout automate observateur représente la propriété de OS par rapport au TS (par exemple : OS doit éviter de rentrer dans la zone de TS et pas le contraire). Le cas échéant n'est pas pris en compte dans notre étude vu que nous ne pouvons pas contrôler les autres navires.*

8.5 Discussions

Le respect des règlements de navigation est un facteur important de la sécurité de la navigation. Afin de permettre aux systèmes de ponts d'incorporer des règlements tels que les COLREGs d'une manière vérifiable, une formalisation des règlements de navigation est nécessaire. Le travail présenté dans ce chapitre propose la formalisation des propriétés d'évitement de collision pour les bateaux suivant les règlements internationaux pour la prévention des collisions en mer (COLREG). Il s'agit du premier cadre de formalisation des règles de navigations maritimes connu pour l'évitement autonome des collisions et permettrait de tester et de certifier des algorithmes avec des paramètres configurables avant la mise en service à l'aide des automates observateurs. Le raisonnement logique des automates permet à des outils automatisés (model-checking) de vérifier l'exactitude des formalisations. Cela permet aux développeurs de logiciels pour les systèmes de ponts de vérifier leurs logiciels. De plus, cette formalisation des COLREGs peut également être utilisée directement dans les systèmes de navigation : les pilotes automatiques pouvant déterminer des routes qui sont connues pour être conformes aux COLREGs ou les affichages de cartes peuvent identifier les violations des COLREGs et signaler les avertissements appropriés.

Nous avons limité notre étude aux rencontres de navires à moteur en haute mer. Les données ne comprennent que les rencontres de deux navires, mais les règles peuvent être évaluées sur n'importe quel nombre de navires grâce à l'approche compositionnelle. Cependant, pour ces situations, les COLREGS sont parfois peu précises, par exemple, une situation avec trois navires où un navire est dépassé par un autre, alors qu'un troisième navire croise la trajectoire du navire dépassé par la gauche. Dans ce cas, le bateau dépassé se trouve à la fois en position d'attente et en position de dégagement. En outre, l'intégration de la hiérarchie de prévention des collisions entre les types de navires (voir la règle 18 du COLREGS) augmenterait l'applicabilité et pourrait être réalisée en passant d'un ensemble de règles à un autre selon les types de navires. En outre, la règle de manœuvre de dernière minute, qui s'applique lorsqu'un des navires enfreint les règles présentées, dépend fortement de la situation (c'est-à-dire du trafic, des obstacles statiques et des conditions météorologiques) et n'est pas suffisamment spécifiée dans le COLREGS pour être facilement formalisable.

Chapitre

9

Conclusion et perspectives

« A pattern system does not belong to an individual, but to the community of experts and practitioners who contribute to and use it. »

— Dwyer et al.

Sommaire

9.1 Conclusion 150

9.1.1 ECDL et grammaire structurelle 151

9.1.2 Observateurs de patrons de propriétés 151

9.1.3 Validation des transformations 152

9.1.4 Exemple d'illustration 152

9.2 Travaux futurs 153

9.2.1 Enrichissement du langage ECDL 153

9.2.2 Extension de l'application des patrons ECDL pour les navires autonomes 153

9.1 Conclusion

Nous présentons dans ce chapitre un bilan du travail que nous avons effectué ainsi qu'un ensemble d'ouvertures et de perspectives pour ce travail.

La vérification des systèmes complexes et en temps réel est un défi important. Au cours des dernières décennies, de nombreuses techniques formelles de vérification des systèmes complexes concurrents et temps réel, ainsi que de nombreux langages de propriétés, ont été proposés. Malheureusement, beaucoup de ces techniques impliquent des formalismes qui ne sont pas toujours faciles à manipuler par les ingénieurs des outils de vérification en industrie.

En particulier, les logiques temporelles offrent un moyen très puissant d'exprimer les propriétés de vérification des systèmes concurrents. Cependant, elles sont souvent considérées comme trop compliquées (et peut-être aussi trop riches) pour être largement adoptées par les ingénieurs. De plus, ils nécessitent généralement des outils avancés dédiés au model-checking de leurs propriétés.

Généralement, la méthode la plus utilisée pour vérifier les propriétés des systèmes embarqués consiste à transformer ces propriétés en automates. Les automates résultant peuvent être vérifiés en utilisant les outils de model-checking (Uppaal, OBP, ...). Néanmoins, l'un des problèmes qui se pose généralement pour cette méthode, est la preuve de correction des transformations de propriétés formelles proposées.

Nous nous sommes intéressé dans cette thèse au problème de validation des transformations des propriétés en automates observateurs. Mais, le langage choisi pour notre étude CDL souffre de beaucoup de problèmes et les propriétés définies dans ce langage manquent de formalisation ainsi que la notion de temps. Notre deuxième préoccupation consistait donc a définir une formalisation des patrons existants et enrichir le langage CDL.

Par conséquent, nous avons identifié de nouveaux patrons de propriétés couramment utilisées pour les systèmes temps réel, recueillis à partir de plusieurs travaux sur la vérification. Nous avons proposé pour chaque patron une syntaxe aussi lisible que possible par les humains, afin que des ingénieurs non experts en méthodes formelles puissent les utiliser. De plus, nous montrons comment les traduire en propriétés d'accessibilité pures en utilisant des observateurs simples, c'est-à-dire des sous-systèmes supplémentaires qui observent certaines actions du système et peuvent également utiliser la notion de temps. Alors qu'une classe de patron doit être traduite en observateurs "must-reach" (c'est-à-dire pour lesquels un bon état doit être atteint à chaque exécution), tous les autres patrons peuvent être traduits en observateurs de non-accessibilité (c'est-à-dire que la propriété est correcte si un mauvais état donné n'est jamais atteint). Par conséquent, leur vérification en pratique évite l'utilisation d'algorithmes de vérification complexes ou d'outils dédiés, et les développeurs d'outils peuvent les mettre en œuvre à coût modéré.

9.1.1 ECDL et grammaire structurelle

Nous proposons ici un ensemble de patrons couramment utilisés pour spécifier la correction des systèmes temps réel complexes. Les principaux avantages sont les suivants. Premièrement, l'ingénieur non-expert en méthodes formelles peut facilement spécifier la correction du système à partir d'une bibliothèque de propriétés communes exprimées dans un langage intuitif. Cependant, un catalogue de patrons de spécification de propriétés complet et cohérent ne garantit pas à lui seul une application efficace dans des scénarios du monde réel. Les difficultés résident dans la complexité de la spécification et des techniques mathématiques associées. Les deux requièrent une expertise considérable. Les ingénieurs système n'ont pas toujours le niveau de formation nécessaire pour maîtriser leur utilisation. C'est pourquoi nous équipons notre catalogue d'un langage naturel ("Grammaire anglaise structurée") pour permettre une traduction syntaxique des instances concrètes des patrons en patrons ECDL et patrons observateurs. Un catalogue de patrons est un travail en progression. Notre catalogue de patrons ne fait pas exception. De nouveaux patrons peuvent être découverts lors de la définition des propriétés du système pour les domaines émergents.

9.1.2 Observateurs de patrons de propriétés

Dans cette thèse nous proposons un ensemble étendu de patrons couramment utilisés pour spécifier la correction des systèmes temps réel complexes. Le principal avantage de ce travail est qu'il aide les ingénieurs non experts en méthodes formelles et en logiques temporelles à spécifier les propriétés permettant de vérifier la correction des systèmes d'une manière plus facile. Dans la majorité des cas, les patrons observateurs sont simples à écrire et leur signification est claire au premier coup d'œil pour l'utilisateur d'outils de vérification. Cependant, lorsque le mode de fonctionnement ou l'historique d'exécution est important pour la propriété à exprimer, leur écriture devient fastidieuse. C'est un problème que l'on rencontre également avec les logiques formelles. Un autre point important est lié à la difficulté, dans certains cas, d'exprimer des propriétés formelles complexes. Pour pouvoir faciliter l'utilisation de la technique proposée, et en particulier l'expression des observateurs, il nous est apparu nécessaire de proposer une approche compositionnelle qui sert à définir des sous systèmes observateurs et les composer. Une conjonction de patrons peut être utile dans le cas où l'on veut vérifier plusieurs propriétés. Dans le cas de nos patrons, nous avons mis en place une méthode de composition permettant de vérifier différentes propriétés en même temps, en utilisant une seule définition du mauvais état. Ainsi, si le mauvais état n'est pas atteignable, toutes les propriétés sont vraies. Néanmoins, aucun exemple n'a été fournis. Au travers de ce travail, nous avons tenté de contribuer à apporter des réponses à certaines questions relatives à la technique de vérification basée sur les observateurs. Nous nous sommes concentrés sur l'aide qui peut être apportée à l'utilisateur pour définir et manipuler ces automates codant des propriétés et pour contourner l'inévitable complexité de cette opération.

9.1.3 Validation des transformations

Comme le système serait plus efficace si les observateurs peuvent être générés automatiquement, le développement d'un outil d'assistance pour cette tâche était l'un de nos contributions dans cette recherche. L'exactitude d'une telle traduction devait être vérifiée, à cette fin, l'utilisation d'un outil de preuve de théorème (l'assistant de preuve COQ) était la solution adéquate. La vérification de la cohérence des spécifications nécessite de traduire les patrons de propriétés écrit en ECDL à des automates observateurs. Pour que la méthode définie dans cette thèse soit formelle, l'implémentation d'une partie des règles de traduction a été prouvée avec l'assistant de preuve Coq. Cette traduction permet de générer un outil de traduction validé d'une sous-partie du langage ECDL vers les observateurs. Pour faciliter l'utilisation de la méthode définie dans ce document, des règles de constructions et de compositions des observateurs sont définies. Ces règles sont un catalogue de règles jugées utiles pour les spécifications. Elles sont accompagnées d'une esquisse de leur preuve de validité, contenant les arguments principaux pour s'assurer de la correction de ces règles. Seule une partie de l'outil de traduction des patrons vers des observateurs est actuellement prouvée avec Coq. Dans l'ensemble donc, pour être mise en place dans le cadre de projets industriels, un meilleur outillage de la méthode de spécification est nécessaire.

9.1.4 Exemple d'illustration

La particularité de ce travail réside dans l'introduction de la notion des propriétés de navigations maritimes comme un cas d'étude de correction de systèmes embarqués en utilisant les patrons observateurs ECDL. L'utilité pratique de notre approche est de développer la théorie de la navigation autonome des navires en toute sécurité et, à cette fin, d'analyser le problème de la navigation dans le cadre d'un modèle rigoureux basé sur l'état. L'approche pour la synthèse de la stratégie de navigation sûre a été présentée comme observateurs avec l'objectif d'éviter les collisions. La prise en compte de plusieurs contraintes secondaires importantes en pratique, telles que le vent, les courants, les erreurs de navigation du navire de l'adversaire et la présence d'autres obstacles (panneaux nautiques, petits bateaux), complique la tâche de synthèse et suppose la validation de l'approche sous des contraintes supplémentaires qui n'ont pas encore été prise en compte dans notre étude. Bien que limité à deux navires naviguant dans des scénarios offshore, notre travail est la première tentative de synthèse d'une stratégie de navigation sûre et optimale. Le problème de navigation est illustré par des exemples basés sur les spécifications de navigation des règles de COLREG de prévention de collision.

L'expérimentation de vérification de propriétés a été menée sur une navigation maritime d'un navire autonome en collaboration avec une équipe d'ingénieurs nous a permis d'entrevoir l'intérêt d'une telle approche.

9.2 Travaux futurs

Ce travail ouvre des perspectives d'axes d'études sur plusieurs aspects. A court terme, l'ensemble des règles de transformation des patrons de propriétés ECDL vers des automates observateurs doivent être complètement définies et implantées. Étant persuadés de l'intérêt essentiel de disposer d'un tel outil dans une activité de modélisation, nous avons réaliser une partie importante de ce travail, nous devons compléter ce travail ayant comme objectif la mise à disposition d'un éditeur graphique.

9.2.1 Enrichissement du langage ECDL

L'expérimentation nous a permis de conclure que les patrons de définition de propriétés sont très utiles. Les patrons proposés dans ce travail restent toutefois limités et simples. Sur la base d'une analyse des documents d'exigences et l'étude d'un nombre pertinent de spécifications du monde réel, ils peuvent être enrichis pour donner naissance à tout un nouveau ensemble de patrons pouvant constituer une bibliothèque de référence. Si nécessaire, le système de patrons sera mis à jour à la suite de cette étude. La conception de patrons de correction pour les extensions probabilistes des systèmes temps réel sera parmi les premieres extensions à considérer.

Dans cette thèse, seuls les patrons exprimant des propriétés qui doivent être vraies (ou fausses) pour toutes les exécutions (scope "*Globally*") ont été pris en compte, car ils sont de loin les plus courants dans le large ensemble d'études de cas que nous avons considérés. L'extension de notre ensemble de patrons observateurs (avec d'autre scopes) fait l'objet de travaux futurs. Une autre extension est aussi considérée,il s'agit de celle des patrons représentant des propriétés de systèmes hybrides. Dans les systèmes hybrides, les horloges sont généralisées aux variables continues, qui peuvent avoir un flux (éventuellement non linéaire). Bien que nos patrons s'étendent d'une manière directe aux systèmes hybrides, ils peuvent ne pas être suffisants pour capturer les propriétés de correction communes. En effet, la correction des systèmes hybrides est souvent spécifiée en termes de variables qui se situent entre certaines limites, par exemple, le niveau d'un réservoir d'eau doit rester au-dessus d'un certain minimum et en dessous d'un certain maximum.

9.2.2 Extension de l'application des patrons ECDL pour les navires autonomes

Dans le cadre de travaux futurs, nous comptons développer un outil complet basé sur le concept de code porteur de preuve qui permet aux développeurs de logiciels de vérifier leur logiciel par rapport à une formalisation de COLREG. Nous souhaitons ensuite intégrer cet outil dans des logiciels de navigation autonomes réels afin d'améliorer la sécurité des technologies

de navigation. La prise en compte de plusieurs contraintes secondaires importantes en pratique, telles que le vent, les courants, les erreurs de navigation du navire de l'adversaire et la présence d'autres obstacles (panneaux nautiques, petits bateaux), est à prendre en considération dans nos travaux futurs. Le développement de la théorie maritime pour capturer les situations de navigation multi-navires dans les zones portuaires à fort trafic et l'intégration de la navigation hivernale restent des travaux futurs.

Cinquième partie

Annexes

Annexe

Grammaire structurelle

Dans cette thèse nous avons définis une grammaire Chapitre 4 comme une approche formelle qui génère le langage des patrons de propriétés ECDL. Nous utilisons une grammaire structurelle pour spécifier une sémantique opérationnelle des patrons de propriétés ECDL définissant une traduction à des automates observateurs.

Nous fournissons une définition formelle d'une grammaire utilisée comme entrée par un autre outil de reconnaissance du langage (ANTLR) pour générer un vérificateur de modèle. De plus, la technique d'implémentation des actions sémantiques dans ANTLR est présentée, c'est-à-dire le concept de connexion entre l'évaluation des attributs dans la grammaire.

Les règles de la grammaire sont présentées sous format écrit ainsi que sous forme de figure afin de clarifier les différentes relations entre ces règles.

A.1 Grammaire de patrons de propriétés

A.1.1 Grammaire ECDL

```
1  /**
2   * E-CDL notation lexical rules.
3   */
4  grammar ECDL ;
5
6  /** A rule called `ecdlFile` which will be our entry point
7   *  This rule means that a program is a sequence of one or more `pattern`
8   */
9  ecdlFile : patterns=pattern+ EOF ;
```

Listing 1 – E-CDL

Une règle appelée **ecdlFile** sera notre point d'entrée, cette règle signifie qu'un programme est une séquence d'un ou plusieurs **Pattern**.

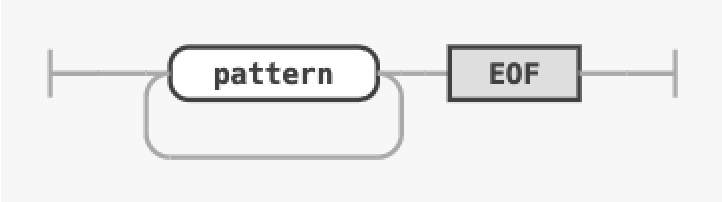


FIGURE A.1 – Grammaire ECDL

A.1.2 Règle Pattern

```
10 pattern  : left+=arity LEADSTO right+=arity timeExpression optionsType+ END # Response
11         | left+=arity PRECEDS right+=arity timeExpression optionsType END  # Precedence
12         | arity+ EXISTS  timeExpression optionsType END  # Existence
13         | arity+ ABSENTS  timeExpression optionsType END  # Absence
14         ;
```

Listing 2 – Pattern

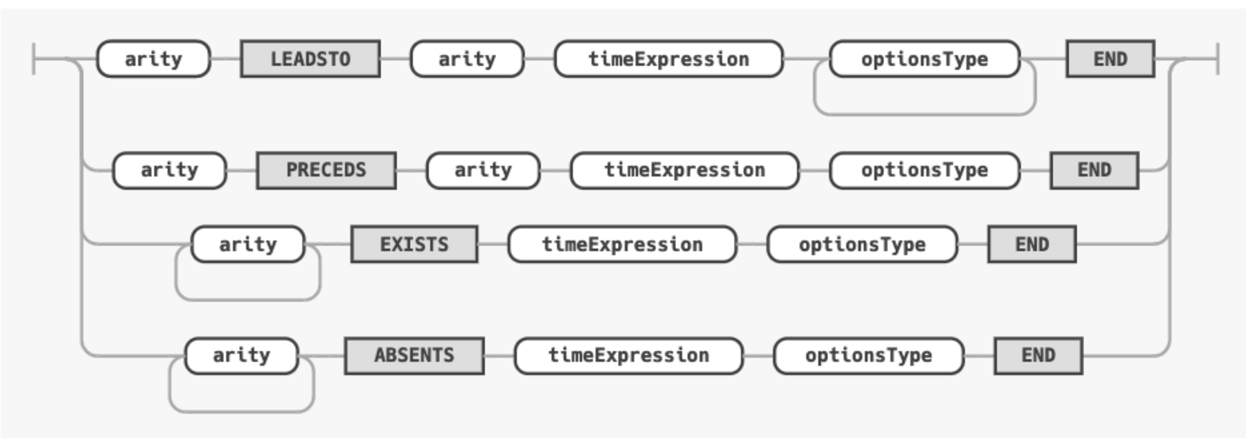


FIGURE A.2 – Pattern

A.1.3 Règles Arity

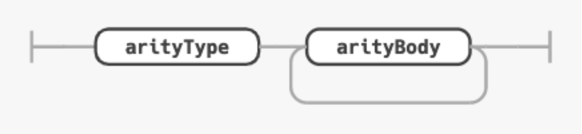


FIGURE A.3 – Arity

```
15  /**
16  * Arity Rule
17  */
18  arity : arityType arityBody+ ;
19  arityBody : arityOccurence arityExpression ;
20  arityType : AN # AnArityType
21          | ALLCOMBINED # AllCombinedArityType
22          | ALLORDERED # AllOrderedArityType
23          ;
24  arityOccurence : EXACTLYONE # ExactlyOneOccurenceType
25              | ONEORMORE # OneOrMoreOccurenceType
26              | EACH # EachOccurenceType
27              ;
28  arityExpression : OCCURENCE ID ;
```

Listing 3 – Arity Rule

Figure A.4 – Le corps Arity

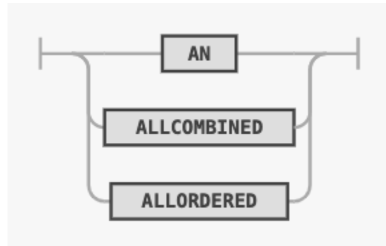


Figure A.5 – Type Arity

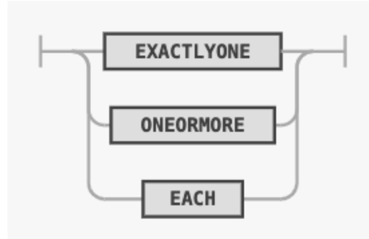


Figure A.6 – Occurence Arity

Figure A.7 – Expression Arity

A.1.4 Règles de Temps

```
1  /**
2   * Time Rule
3   */
4  timeExpression : AFTERATMOST timeDuration # AfterMostTimeExpression
5                 | WITHIN timeDuration # WhithinTimeExpression
6                 | BETWEEN left=timeDuration AND right=timeDuration # BetweenTimeExpression
7                 ;
8  timeDuration : NUMBER timeUnit ;
9  timeUnit : 'seconds'
10          | 'ms'
11          ;
```

Listing 4 – Time Rule

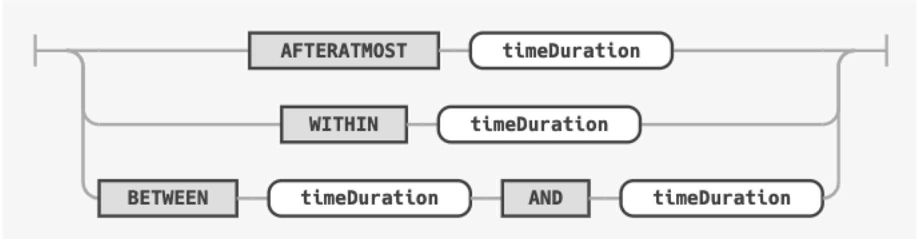


Figure A.8 – Expression de Temps

Figure A.9 – Durée

Figure A.10 – Unité de temps

A.1.5 Options

```
/**
 * Option Rule
 */
optionsType : optionPrecedency # Precedency
        | optionRepeatability # Repeatability
        | optionNullity # Nullity
        ;
optionRepeatability : REPEATABILITY ' :' BOOLEAN ;
optionNullity : OCCURENCE ID occurenceProbability NEVER ;
optionPrecedency : OCCURENCE ID occurenceProbability OCCURS BEFORE THEFIRST OCCURENCE ID ;
occurenceProbability : MAY # MayProbabilityType
                  | MUST # MustProbabilityType
                  ;
```

Listing 5 – Règle de définition des options

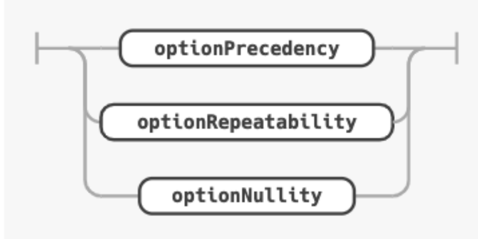


FIGURE A.11 – Type Options

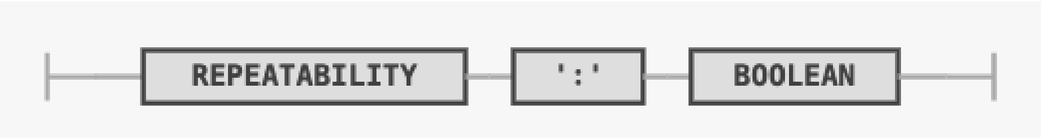


FIGURE A.12 – Option Repeatability

FIGURE A.13 – Option Nullity

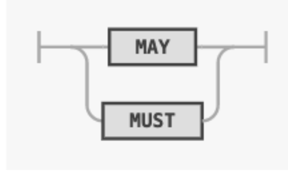


FIGURE A.14 – Occurence Probability

A.1.6 Tokens

```
/**
 * Tokens
 */
AN : 'AN' ; // Match AN keyword
END : 'end' ; // Match end keyword
EXACTLYONE : 'Exactly one' ;
ALLORDERED : 'ALL Ordered' ;
ALLCOMBINED : 'ALL Combined' ;
AFTERATMOST : 'After at most' ;
WITHIN : 'Within' ;
BETWEEN : 'Between' ;
ONEORMORE : 'One or more' ;
OCCURENCE : 'Occurence of' ;
EACH : 'Each' ;
LEADSTO : 'Leads To' ;
PRECEDS : 'Preceds' ;
EXISTS : 'Exists' ;
ABSENTS : 'Absents' ;
OCCURS  : 'occurs' ;
BEFORE  : 'before' ;
THEFIRST  : 'the first' ;
MAY  : 'May' ;
MUST  : 'Must' ;
AND  : 'and' ;
NEVER  : 'never' ;
REPEATABILITY  : 'Repeatability' ;

// Types
BOOLEAN : 'true' | 'false' ;
ID : [a-zA-Z][a-zA-Z0-9_-]* ;      // Match identifiers
DURATION : [a-z]+ ;      // Match Duration ex : seconds
NUMBER : [0-9]+  ; // Match numbers

// Skippable tokens
WS : [ \t\n]+ -> skip ;
COMMENT : '--' ~[\r\n]* -> skip ;
```

Listing 6 – ECDL Tokens

Annexe

B Patrons observateurs

B.1 Observateurs de patron de précédence

B.1.1 Précédence AllOrdered version cyclique

Similairement aux patrons avec le variant *allOrdered* présenté dans Chapitre 5, nous considérons la chaîne d'actions qui se produit en premier. Soit : **`AllOrdered`** a_1, a_2 `Occur before (`a_3`)` un exemple de patron de précédence version AllOrdered. Dans ce patron, nous vérifions donc que si une action a_3 se produit, alors la dernière action a_2 de la chaîne d'actions ordonnées a_1, a_2 s'est produite (au moins une fois) avant la première occurrence de a_3 et ainsi de suite de manière cyclique. La description de ce patron est donnée dans Tableau B.1.

Syntaxe Abstraite : Chaque fois que a_3 se produit Alors toute la chaîne AllOrdered a_1, a_2 s'est produit avant.
Description : ***FR*** : "Chaque fois que a_3 se produit au moins une fois, alors a_2 s'est produit avant la première occurrence de a_3 (s'il y'en a une)". ***EN*** : "Every time a_3 happens, then a_2 has happened before the first occurrence of a_3 (if any)".

Tableau B.1 – Patron de Précédence AllOrdered version Cyclique

Nous donnons l'observateur correspondant sur Figure B.1(a). Si a_3 se produit en premier (avant la dernière action de la chaîne ordonnée, dans notre exemple a_2), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. Dés que a_2 se produit, l'observateur entre dans un autre emplacement (s_2), et a_3 peut se produire.

B.1.2 Précédence AllOrdered version strictement cyclique

Dans ce patron, nous vérifions que si une action a_3 se produit, alors la dernière action a_2 de la chaîne d'actions ordonnées a_1, a_2 s'est produite exactement une fois depuis la dernière

occurrence de a_3 et ainsi de suite de manière cyclique. La description de ce patron est donnée dans Tableau B.2. L'observateur correspondant est montré dans Figure B.1(b). les actions a_1, a_2

Syntaxe Abstraite : Depuis la derniére occurence a_3 la chaîne AllOrdered a_1, a_2 se produit exactement une fois.
Description : ***FR*** : "Chaque fois que a_3 se produit au moins une fois, alors a_2 s'est produit avant la première occurrence de a_3 et continue a se produire exactement une fois". ***EN*** : "Every time a_3 happens, then a_2 has happened before exactly one time ".

TABLEAU B.2 – Patron de Précédence AllOrdered version strictement Cyclique

n'ont pas le droit de se produire plusieurs fois avant qu'une autre occurrence de a_3 se produise, sinon la transition avec a_1, a_2 part directement vers un état "reject".

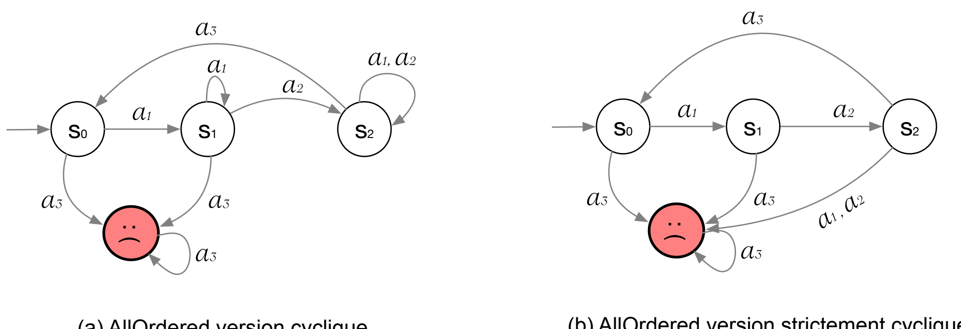


FIGURE B.1 – Automates observateurs pour le patron "Précédence" AllOrdered versions cycliques

B.1.2.1 Précédence AllCombined version cyclique

Soit : **`AllCombined`** a_1, a_2 `Occur before (`a_3`)`. Dans ce patron, nous vérifions que si une action a_3 se produit, alors au moins une des chînes d'actions $\{a_1, a_2\}$ s'est produite (au moins une fois) avant la première occurrence de a_3 et ainsi de suite de manière cyclique. La description de ce patron est montré par Tableau B.3.

L'observateur correspondant à ce patron est montré sur la figure Figure B.2(a). Si a_3 se produit en premier (avant l'une des actions de la chaîne a_1, a_2), l'observateur entre dans son mauvais état "*reject*" et y reste pour toujours. Dés que a_3 se produit, les actions $a1, a_2$ peuvent se produire d'une manière cyclique, mais toujours après a_3.

Syntaxe Abstraite : Chaque fois que a_3 se produit Alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produite avant.
Description : ***FR*** : "Chaque fois a_3 se produit au moins une fois, alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit avant la première occurrence de a_3". ***EN*** : "Every time a_3 happens at least once, then either $\{a_1, a_2\}$ or $\{a_2, a_1\}$ has happened before the first occurrence of a_3 (if any)".

Tableau B.3 – Patron de précédence AllCombined version cyclique

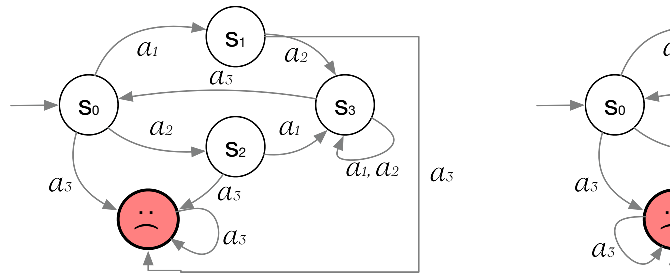
B.1.2.2 Précédence AllCombined version strictement cyclique

Le même raisonnement de patron précédent s'applique pour ce patron, sauf que cette fois-ci les chînes $\{a_1, a_2\}$, $\{a_2, a_1\}$ sont autorisées a se produire exactement une fois (ou pas) après l'occurrence de a_3. La description de ce patron est montrée par Tableau B.4.

Syntaxe Abstraite : Chaque fois a_3 Alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produite avant exactement une fois.
Description : ***FR*** : "Chaque fois a_3 se produit au moins une fois, alors $\{a_1, a_2\}$, $\{a_2, a_1\}$ s'est produit avant la première occurrence de a_3 exatement une fois". ***EN*** : "Every time a_3 happens at least once, then either $\{a_1, a_2\}$ or $\{a_2, a_1\}$ has happened before the first occurrence of a_3 exactly ones (if any)".

Tableau B.4 – Patron de précédence AllCombined version strictement cyclique

L'observateur correspondant à ce patron est montré sur la figure Figure B.2(b).



(a) AllCombined version cyclique

(b) AllCombined version strictement cyclique

Figure B.2 – Automates observateurs pour le patron "Précédence" AllCombined versions cycliques

B.2 Observateurs de patron de réponse

B.2.1 Réponse AllOrdered version cyclique

Dans ce patron, nous vérifions que si la chaîne d'actions a_1, a_2 se produit, alors en réponse, l'action a_3 se produit et ce même comportement se répète de manière cyclique. La description de ce patron est montré dans Tableau B.5.

Syntaxe Abstraite : Chaque fois a_1, a_2 Alors éventuellement a_3.
Description : ***FR*** : "Chaque fois a_1 se produit suivie par a_2, alors a_3 finit par se produire". ***EN*** : "Every time a_1 happens followed by a_2, then a_3 eventually happens".

Tableau B.5 – Patron de réponse AllOrdered version cyclique

Nous donnons l'observateur correspondant sur Figure B.3(a). Si l'ordre est respecté, et a_2 se produit après a_1, l'observateur entre dans un état intermédiaire qui n'est pas bon s_2 en attendant l'arrivée du a_3. Dés que a_3 arrive, l'observateur entre dans un état bon "succes" et le comportement se répète.

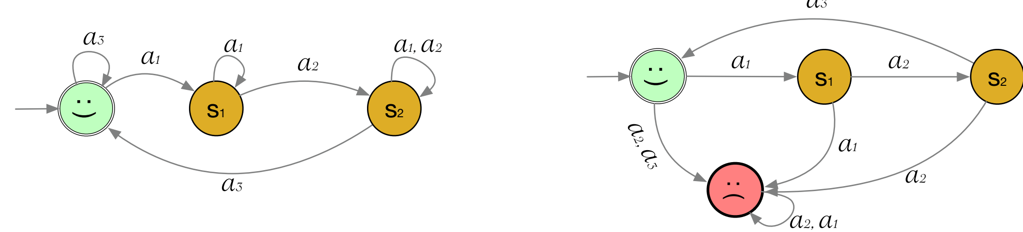
B.2.2 Réponse AllOrdered version strictement cyclique

C'est le même patron que celui de la version cyclique avec la seule différence que a_3 ne peut se produire qu'une seule fois avant la prochaine occurrence de a_1 (Tableau B.6).

Syntaxe Abstraite : Chaque fois a_1, a_2 Alors éventuellement a_3 une fois avant la prochaine a_1.
Description : ***FR*** : "Chaque fois a_1 se produit suivie par a_2, alors a_3 finit par se produire exactement une fois avant la prochaine a_1". ***EN*** : "Every time a_1 happens followed by a_2, then a_3 eventually happens exatly once before the next a_1".

Tableau B.6 – Patron de résponse AllOrdered version strictement cyclique

L'observateur correspondant est montré sur Figure B.3(b).



(a) Response AllOrdered Version Cyclique

(b) Response AllOrdered Version Strictement Cyclique

FIGURE B.3 – Automates Observateurs pour le patron “Réponse” AllOrdered versions cycliques

B.2.3 Réponse AllCombined version cyclique

Ce patron suit le même raisonnement que le patron réponse AllCombined sauf que dans cette version le comportement est répétable (Les actions et la réponse alternent de manière cyclique) (Tableau B.7).

Syntaxe Abstraite : Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ Alors a_3 finit par se produire une fois avant la prochaine a_1 ou a_2.
Description : ***FR*** : "Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ se produit, alors a_3 finit par se produire". ***EN*** : "Every time $\{a_1, a_2\}$ or $\{a_2, a_1\}$ happens, then a_3 eventually happens".

TABLEAU B.7 – Patron de réponse AllCombined version cyclique

B.2.4 Réponse AllCombined version strictement cyclique

Syntaxe Abstraite : Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ Alors a_3 une fois avant la prochaine a_1 ou a_2.
Description : ***FR*** : "Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ se produit, alors a_3 finit par se produire exactement une fois avant la prochaine a_1 ou a_2". ***EN*** : "Every time $\{a_1, a_2\}$ or $\{a_2, a_1\}$ happens, then a_3 eventually happens exatly once before the next a_1 or a_2".

TABLEAU B.8 – Patron de réponse AllCombined version strictement cyclique

La description de patron est montrée dans (Tableau B.8). C'est le même patron que celui de la version cyclique avec la seule différence que a_3 ne peut se produire qu'une seule fois avant la prochaine occurrence de a_1 ou a_2. L'observateur correspondant est montré dans Figure B.4(b).

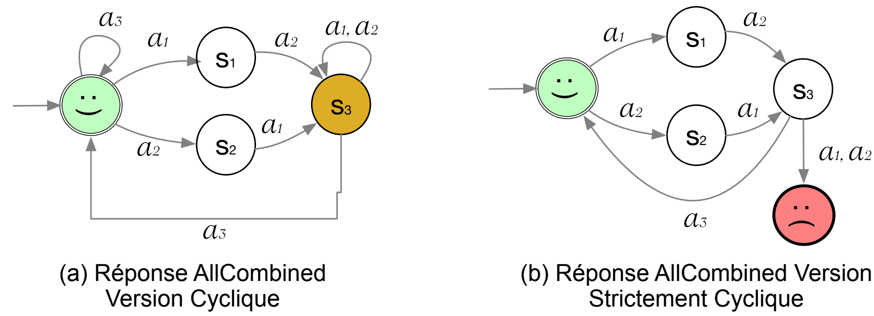


FIGURE B.4 – Automates observateurs pour le patron “Réponse” AllOrdered versions cycliques

B.3 Observateurs de patron de Précédence Bornée

Cette classe de patrons modélise le cas d'une action qui ne peut se produire que si une autre (ou une liste) s'est produite dans un intervalle de temps donné auparavant. Les observateurs de ce type de patron suivent le même raisonnement que les précédents de même types sauf qu'ici il faut prendre en considération la notion de temps .

B.3.1 Précédence Bornée version cyclique

La description de ce patron est montré dans Tableau B.9.

Syntaxe Abstraite : Chaque fois a_3 se produit Alors toute la chaîne AllOrdered a_1, a_2 s'est produit avant au plus d unités de temps.
Description : ***FR*** : "Chaque fois que a_3 se produit au moins une fois, alors a_2 s'est produit au maximum d unités de temps avant la première occurrence de a_3 ". ***EN*** : "Every time a_3 happens, then a_2 has happened at most d units of time before the first occurrence of a_3 ".

TABLEAU B.9 – Patron de précédence bornée AllOrdered version cyclique

Nous donnons l'observateur correspondant sur Figure B.5(a).

B.3.2 Précédence bornée AllOrdered version strictement cyclique

Dans ce patron, nous vérifions que si une action a_3 se produit, alors la dernière action a_2 de la chaîne d'actions ordonnées a_1, a_2 s'est produite exactement une fois et au plus tard d unités de temps depuis la dernière occurrence de a_3 et ainsi de suite de manière cyclique. La description de ce patron est donné dans Tableau B.10. L'observateur correspondant est montré

Syntaxe Abstraite :   Depuis la dernière occurrence a_3 la chaîne AllOrdered a_1, a_2 se produit exactement une fois et avant d unités de temps.
Description :   ***FR*** : "Chaque fois que a_3 se produit au moins une fois, alors a_2 s'est produit au maximum d unités de temps avant la première occurrence de a_3 et continue a se produire exactement une fois".   ***EN*** : "Every time a_3 happens, then a_2 has happened at most d time units before and exactly once (if any)".

Tableau B.10 – Patron de précédence bornée AllOrdered version strictement cyclique

dans Figure B.5(b).

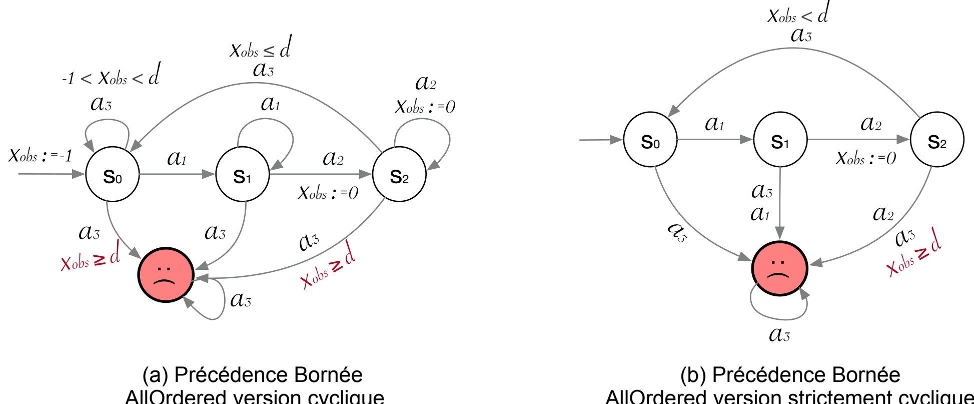


Figure B.5 – Automates observateurs pour le patron "Précédence Bornée" AllOrdered versions cycliques

B.3.3 Précédence bornée AllCombined version cyclique

Dans ce patron, nous devons vérifier les mêmes conditions que dans le patron précédence AllCombined cyclique plus du fait que le temps doit être respecté. (Tableau B.11).

L'observateur correspondant à ce patron est montré sur Figure B.6(a). Afin de détecter la première occurrence de a_3, nous désactivons l'horloge de l'observateur ($x_{obs} := -1$ avant l'état initial (début de l'automate)).

Syntaxe Abstraite : Chaque fois a_3 Alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit avant d unités de temps et avant la prochaine a_3.
Description : ***FR*** : "Chaque fois a_3 se produit, alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit avant la prochaine occurrence de a_3 au plus d unités de temps". ***EN*** : "Every time a_3 happens, then $\{a_1, a_2\}$ or $\{a_2, a_1\}$ happens at most d units of time before the next occurrence of a_3".

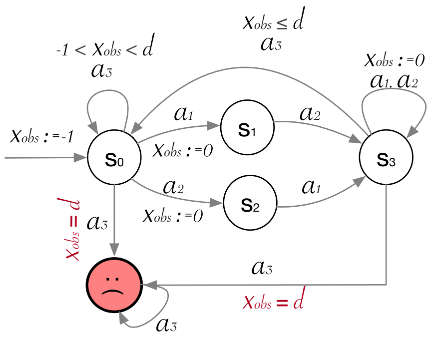
TABLEAU B.11 – Patron de Précédence AllCombined version cyclique

B.3.4 Précédence bornée AllCombined version strictement cyclique

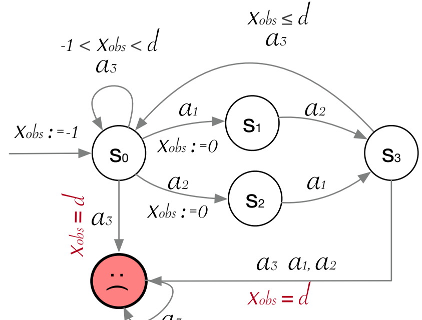
La description de ce patron est montrée par Tableau B.12. L'observateur correspondant à ce patron est montré sur Figure B.6(b).

Syntaxe Abstraite : Chaque fois a_3 Alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit exactement une fois avant d unités de temps et avant la prochaine a_3.
Description : ***FR*** : "Chaque fois a_3 se produit, alors $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ s'est produit exactement une fois avant la prochaine occurrence de a_3 au plus d unités de temps". ***EN*** : "Every time a_3 happens, then $\{a_1, a_2\}$ or $\{a_2, a_1\}$ happens at most d units of time and exactly once before the next occurrence of a_3".

TABLEAU B.12 – Patron de Précédence AllCombined version cyclique



(a) Précédence Bornée AllCombined version cyclique



(b) Précédence Bornée AllCombined version strictement cyclique

FIGURE B.6 – Automates observateurs pour le patron "Précédence Bornée" AllCombined versions cycliques

B.4 Observateurs de patron de réponse bornée

Cette classe de propriétés considère le cas d'une liste d'actions suivies éventuellement d'une action dans un certain intervalle de temps.

Pour éviter la répétition des explications, nous allons fournir seulement la description et les observateurs correspondants aux patrons de cette catégories.

B.4.1 Réponse bornée AllOrdered version cyclique

La description de ce patron est montré dans Tableau B.13.

Syntaxe Abstraite : Chaque fois a_1, a_2 Alors éventuellement a_3 dans d unités de temps.
Description : ***FR*** : "Chaque fois a_1 se produit suivie par a_2, alors a_3 finit par se produire dans d unités de temps et avant la prochaine occurrence de a_2 (s'il y en a une)"". ***EN*** : "Every time a_1 happens followed by a_2, then a_3 eventually happens within d units of time, and before the next occurrence of a_2 (if any)".

Tableau B.13 – Patron de Réponse bornée AllOrdered version cyclique

Nous donnons l'observateur correspondant sur Figure B.7(a).

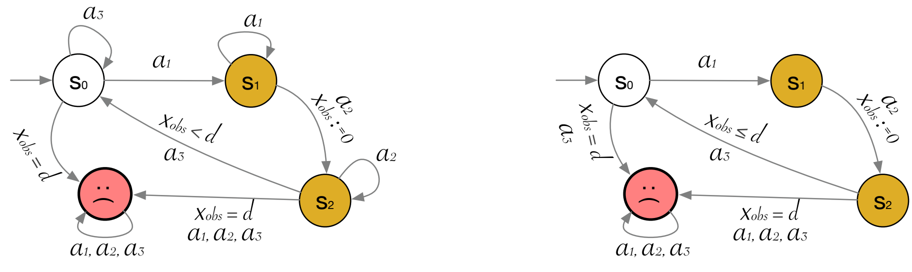
B.4.2 Réponse AllOrdered version strictement cyclique

La description de ce patron est montrée dans (Tableau B.14).

Syntaxe Abstraite : Chaque fois a_1, a_2 Alors éventuellement a_3 avant la prochaine a_2 dans d unités de temps.
Description : ***FR*** : "Chaque fois a_1 se produit suivie par a_2, alors a_3 finit par se produire exactement une fois dans d unités de temps avant la prochaine a_1". ***EN*** : "Every time a_1 happens followed by a_2, then a_3 eventually happens within d time units and exactly once before the next a_2".

Tableau B.14 – Patron de réponse bornée AllOrdered version strictement cyclique

L'observateur correspondant est montré sur Figure B.7(b).



(a) Réponse Bornée AllOrdered
version cyclique

(b) Réponse Bornée AllOrdered
version strictement cyclique

FIGURE B.7 – Automates observateurs pour le patron “Réponse” AllOrdered versions cycliques

B.4.3 Réponse bornée AllCombined version cyclique

(Tableau B.15) montre la description de ce patron. L'observateur correspondant est montré

Syntaxe Abstraite : Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ Alors se produit a_3 une fois avant la prochaine a_1 ou a_2 dans d unités de temps.
Description : ***FR*** : "Chaque fois $\{a_1, a_2\}$ ou $\{a_2, a_1\}$ se produit, alors a_3 finit par se produire dans d unités de temps et avant la prochaine a_1 ou a_2". ***EN*** : "Every time $\{a_1, a_2\}$ or $\{a_2, a_1\}$ happens, then a_3 eventually happens within d time units before the next occurence a_1 or a_2".

TABLEAU B.15 – Patron de réponse bornée AllCombined version cyclique

dans Figure B.8(a).

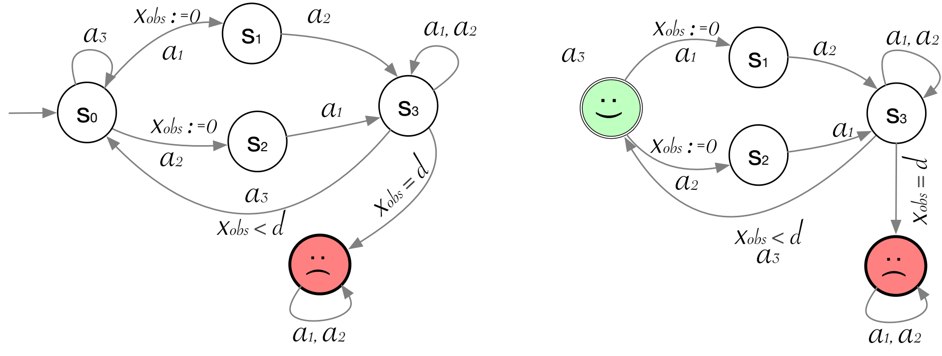
B.4.4 Réponse AllCombined version strictement cyclique

Syntaxe Abstraite : Chaque fois a_1 ou a_2 Alors éventuellement a_3 une fois avant la prochaine a_1 ou a_2.
Description : ***FR*** : "Chaque fois a_1 ou a_2 se produit, alors a_3 doit se produire dans d unités de temps exactement une fois avant la prochaine a_1 ou a_2". ***EN*** : "Every time a_1 or a_2 happens, then a_3 happens within d time units exactly once before the next a_1 or a_2" .

TABLEAU B.16 – Patron de réponse bornée AllCombined version strictement cyclique

La description de patron est montré dans (Tableau B.16).

L'observateur correspondant est montré dans Figure B.8(b).



(a) Réponse Bornée AllCombined
Version cyclique

(b) Réponse Bornée AllCombined
Version strictement cyclique

Figure B.8 – Automates observateurs pour le patron "Réponse bornée" AllCombined versions cycliques

Abréviations

AIS Automatic identification systems. 141

AN2S Akhbar Neb Security System. 68, 76, 96, 97

AV Véhicules Autonomes. 132

AWS Amazon Web Services. 16

CCS Calculus of Communicating Systems. 40

CCTL Cadenced Computational Tree Logic - logique arborescente computationnelle cadencée. 59

CDL Context Description Language. xix, 6–9, 30, 32–36, 43, 50–52, 103, 150

COLAV Collision Avoidance. 143

COLREG Collision Regulations : Convention sur le règlement international pour prévenir les collisions en mer, 1972. xv, xx, 131, 133, 134, 136, 138, 140, 142–146, 148, 153

CPA Closest Point of approach. 138, 139, 143

CSP Communicating Sequential Processes. 40

CTL Computation tree logic. 17, 20, 31

ECDL Extended Context Description Language. xx, 9, 50, 59, 67, 76, 86, 89, 92, 94–97, 153

EN English. 67–82, 163–172

FR French. 67–82, 163–172

FSA Finite-State Automaton. 34

IMO International Maritime Organization. 133

LTL Linear temporal logic. 17, 20, 31, 54–56

LTS Labeled Transition system = Système de transitions étiqueté. 33, 117

MITL Metric Interval Temporal Logic. 18

MTL Metric temporal logic. 18

OBP Observer-Based Prover. 30, 32, 33, 66, 95, 150

OPENJML Open Java Modeling Language. 16

OS Ownship. 136, 139

OWL Web Ontology Language. 21

PI Patterns Indicators. 117

PVS Prototype Verification System - Système de vérification prototype. 22

QRE Query Request. 31

RATP Régie Autonome des Transports Parisiens. 16

RRT Rapidly-exploring Random Tree. 132

RTGIL Real Time Graphical Interval Logic. 18

SCC Shore Control Center. 133

SLTS Sub Labeled Transition system. 117, 118

TCTL Timed Computation tree logic. 18

TLA+ Temporal Logic of Actions. 16

TS Target ship. 136, 139, 145, 147

VHF very high frequency. 133

Bibliographie

[1] N. Abid, S. Dal Zilio et D. Le Botlan. « Real-time specification patterns and tools ». In : *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2012, p. 1-15 (cf. p. 20, 21, 35).

[2] L. Aceto. « GSOS and finite labelled transition systems ». In : *Theoretical Computer Science* 131.1 (1994), p. 181-195 (cf. p. 38).

[3] L. Aceto, A. Burgueno et K. G. Larsen. « Model checking via reachability testing for timed automata ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1998, p. 263-280 (cf. p. 36, 42).

[4] L. Aceto et al. « The power of reachability testing for timed automata ». In : *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1998, p. 245-256 (cf. p. 20, 37).

[5] L. Aceto et al. « The power of reachability testing for timed automata ». In : *Theoretical Computer Science* 300.1-3 (2003), p. 411-475 (cf. p. 7, 20, 42, 45).

[6] S. Ahvenjärvi. « The human element and autonomous ships ». In : *TransNav : International Journal on Marine Navigation and Safety of Sea Transportation* 10.3 (2016) (cf. p. 133).

[7] hamid alavi et al. *Specification patterns*. `http://patterns.projects.cs.ksu.edu/`. Accessed : 2019-01-16 (cf. p. 31, 98).

[8] H. Aliee. « Reliability analysis and optimization of embedded systems using stochastic logic and importance measures ». In : (2017) (cf. p. 5).

[9] M. Alpuente, B. Cook et C. Joubert. *Formal Methods for Industrial Critical Systems : 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009, Proceedings*. T. 5825. Springer, 2009 (cf. p. 16).

[10] R. Alur et D. L. Dill. « A theory of timed automata ». In : *Theoretical computer science* 126.2 (1994), p. 183-235 (cf. p. 37-39, 44).

[11] R. Alur, T. A. Henzinger et M. Y. Vardi. « Parametric real-time reasoning ». In : *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, p. 592-601 (cf. p. 68).

[12] É. André. « Observer patterns for real-time systems ». In : *2013 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2013, p. 125-134 (cf. p. 56, 67, 83).

[13] J. W. Atwood, M. Ghodrat et D. Tasak. « Using formal specification and observers to specify and validate the ATM signaling protocols ». In : *Proceedings 24th Conference on Local Computer Networks. LCN'99.* IEEE. 1999, p. 117-120 (cf. p. 42).

[14] S. Austin. « Formal methods : A survey ». In : *Technical Report* (1993) (cf. p. 17).

[15] M. Autili et al. « Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar ». In : *IEEE Transactions on Software Engineering* 41.7 (2015), p. 620-638 (cf. p. 19, 36, 56, 59).

[16] J.-M. Ayache, J.-P. Courtiat et M. Diaz. « Self-Checking software in distributed systems. » In : *ICDCS.* 1982, p. 163-170 (cf. p. 20).

[17] S. Bacherini et al. « A story about formal methods adoption by a railway signaling manufacturer ». In : *International Symposium on Formal Methods.* Springer. 2006, p. 179-189 (cf. p. 17).

[18] T. Ball et al. « SLAM2 : Static driver verification with under 4% false alarms ». In : *Formal Methods in Computer Aided Design.* IEEE. 2010, p. 35-42 (cf. p. 17).

[19] M. Barjaktarovic et M. Nassiff. « The state-of-the-art in formal methods ». In : *AFOSR Summer Research Technical Report for Rome Research Site, Formal Methods Framework-Monthly Status Report, F30602-99-C-0166, WetStone Technologies* (1998) (cf. p. 17).

[20] D. Baroudi, P. Dhaussy et S. Nait-Bahloul. « Formalisation d'une Approche Compositionnelle de Patrons de Propriétés ». In : *Approches Formelles dans l'Assistance au Developpement de Logiciels* (2016), p. 35 (cf. p. 83, 94, 95).

[21] D. Baroudi et S. Nait-Bahloul. « Observer Patterns for Timed Properties ». In : *International Journal of Software Innovation (IJSI)* 9.2 (2021), p. 1-17 (cf. p. 52, 59, 67).

[22] P. Behm, P. Desforges et J.-M. Meynadier. « MÉTÉOR : An industrial success in formal development ». In : *International Conference of B Users.* Springer. 1998, p. 26-26 (cf. p. 17).

[23] G. Behrmann, A. David et K. G. Larsen. « A tutorial on uppaal ». In : *Formal methods for the design of real-time systems* (2004), p. 200-236 (cf. p. 20).

[24] G. Behrmann, K. G. Larsen et J. I. Rasmussen. « Beyond liveness : Efficient parameter synthesis for time bounded liveness ». In : *International Conference on Formal Modeling and Analysis of Timed Systems.* Springer. 2005, p. 81-94 (cf. p. 79).

[25] P. Bellini, P. Nesi et D. Rogai. « Expressing and organizing real-time specification patterns via temporal logics ». In : *Journal of Systems and Software* 82.2 (2009), p. 183-196 (cf. p. 4, 18).

[26] M. R. Benjamin et al. « Navigation of unmanned marine vehicles in accordance with the rules of the road ». In : *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE. 2006, p. 3581-3587 (cf. p. 134).

[27] S. Berezin. « Model checking and theorem proving : a unified framework ». Thèse de doct. PhD thesis, Carnegie Mellon University, 2002 (cf. p. 25).

[28] B. Berthomieu*, P.-O. Ribet et F. Vernadat. « The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets ». In : *International journal of production research* 42.14 (2004), p. 2741-2756 (cf. p. 8).

[29] B. Berthomieu et al. « Fiacre : an intermediate language for model verification in the topcased environment ». In : *ERTS 2008*. 2008 (cf. p. 32).

[30] N. S. Bjørner et al. « Deductive verification of real-time systems using STeP ». In : *International AMAST Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software*. Springer. 1997, p. 22-43 (cf. p. 25).

[31] R. Bloomfield et al. « Formal methods diffusion : Past lessons and future prospects ». In : *International Conference on Computer Safety, Reliability, and Security*. Springer. 2000, p. 211-226 (cf. p. 17).

[32] A. Bouajjani, R. Echahed et R. Robbana. « Verification of context-free timed systems using linear hybrid observers ». In : *International Conference on Computer Aided Verification*. Springer. 1994, p. 118-131 (cf. p. 42).

[33] J.-L. Boulanger. *Formal methods : industrial use from model to the code*. John Wiley & Sons, 2013 (cf. p. 17).

[34] P. Bouyer. « Modèles et algorithmes pour la vérification des systèmes temporisés ». In : *These de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France* (2002) (cf. p. 37).

[35] M. Bozzano et A. Villafiorita. *Design and safety assessment of critical systems*. CRC press, 2010 (cf. p. 19).

[36] P. Buchholz. « Bisimulation relations for weighted automata ». In : *Theoretical Computer Science* 393.1-3 (2008), p. 109-123 (cf. p. 108).

[37] K. C. Castillos et al. « A compositional automata-based semantics for property patterns ». In : *International Conference on Integrated Formal Methods*. Springer. 2013, p. 316-330 (cf. p. 98, 103).

[38] E. M. Clarke. *years of model checking. chapter The Birth of Model Checking*. 2008 (cf. p. 23).

[39] E. M. Clarke et E. A. Emerson. « Design and synthesis of synchronization skeletons using branching time temporal logic ». In : *Workshop on Logic of Programs*. Springer. 1981, p. 52-71 (cf. p. 23).

[40] E. M. Clarke Jr et al. *Model checking*. MIT press, 2018 (cf. p. 23).

[41] D. R. Cok. « Java automated deductive verification in practice : lessons from industrial proof-based projects ». In : *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, p. 176-193 (cf. p. 16).

[42] P. Cousot et R. Cousot. « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, p. 238-252 (cf. p. 21).

[43] P. Cousot et al. « Static determination of dynamic properties of programs. » In : (1977) (cf. p. 22).

[44] P. Cousot et al. « The ASTRÉE analyzer ». In : *European Symposium on Programming*. Springer. 2005, p. 21-30 (cf. p. 16).

[45] D. Craigen, S. Gerhart et T. Ralston. « An international survey of industrial applications of formal methods ». In : *Z User Workshop, London 1992*. Springer. 1993, p. 1-5 (cf. p. 17).

[46] K. Daniel et al. « Theta* : Any-angle path planning on grids ». In : *Journal of Artificial Intelligence Research* 39 (2010), p. 533-579 (cf. p. 132).

[47] S. De Gouw et al. « OpenJDK's Java. utils. Collection. sort () is broken : The good, the bad and the worst case ». In : *International Conference on Computer Aided Verification*. Springer. 2015, p. 273-289 (cf. p. 17).

[48] P. Dhaussy et J.-C. Roger. « Cdl (context description language) : syntax and semantics ». In : *Rapport technique, ENSTA-Bretagne* 37 (2014) (cf. p. 32, 50, 53).

[49] P. Dhaussy, J.-C. Roger et F. Boniol. « Reducing state explosion with context modeling for model-checking ». In : *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*. IEEE. 2011, p. 130-137 (cf. p. 32-34, 52, 56).

[50] P. Dhaussy et al. « Evaluating context descriptions and property definition patterns for software formal validation ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, p. 438-452 (cf. p. 6-8, 32, 34-36, 50, 56).

[51] P. Dhaussy et al. « Using context descriptions and property definition patterns for software formal verification ». In : *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE. 2008, p. 89-96 (cf. p. 4, 6, 8, 32, 86).

[52] R. J.-C. Dhaussy Philippe. *OBPe (OBP Explorer) V. 1.3 Documentation*. 2012 (cf. p. 8, 45).

[53] M. Diaz. « Modeling and analysis of communication and cooperation protocols using Petri net based models ». In : *Computer Networks (1976)* 6.6 (1982), p. 419-441 (cf. p. 20).

[54] M. Diaz, G. Juanole et J.-P. Courtiat. « Observer-a concept for formal on-line validation of distributed systems ». In : *IEEE Transactions on Software Engineering* 20.12 (1994), p. 900-913 (cf. p. 20, 41, 42).

[55] J. Dingel et T. Filkorn. « Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving ». In : *International Conference on Computer Aided Verification*. Springer. 1995, p. 54-69 (cf. p. 25).

[56] R. DO-178C. *Software considerations in airborne systems and equipment certification*. Committee : SC-205., 2011 (cf. p. 16).

[57] R. DO-333. *formal methods supplement to DO-178C and DO-278A*. Committee : SC-205., 2011 (cf. p. 16).

[58] A. Dokhanchi, B. Hoxha et G. Fainekos. « Formal requirement debugging for testing and verification of cyber-physical systems ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2017), p. 1-26 (cf. p. 18).

[59] A. Dokhanchi, B. Hoxha et G. Fainekos. « Metric interval temporal logic specification elicitation and debugging ». In : *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, p. 70-79 (cf. p. 18).

[60] J. S. Dong et al. « Timed automata patterns ». In : *IEEE Transactions on Software Engineering* 34.6 (2008), p. 844-859 (cf. p. 39, 40).

[61] B. P. Douglass. *Doing hard time : developing real-time systems with UML, objects, frameworks, and patterns*. T. 1. Addison-Wesley Professional, 1999 (cf. p. 4).

[62] M. B. Dwyer, G. S. Avrunin et J. C. Corbett. « Patterns in property specifications for finite-state verification ». In : *Proceedings of the 21st international conference on Software engineering*. ACM. 1999, p. 411-420 (cf. p. 4, 6, 8, 17, 18, 30, 35, 98).

[63] M. B. Dwyer, G. S. Avrunin et J. C. Corbett. « Property specification patterns for finite-state verification ». In : *Proceedings of the second workshop on Formal methods in software practice*. ACM. 1998, p. 7-15 (cf. p. 4, 6, 8, 17, 20, 21, 30, 32, 33, 35, 51, 53, 58, 98, 103).

[64] B.-O. H. Eriksen et al. « A modified dynamic window algorithm for horizontal collision avoidance for AUVs ». In : *2016 IEEE Conference on Control Applications (CCA)*. IEEE. 2016, p. 499-506 (cf. p. 132).

[65] B.-O. H. Eriksen et al. « Hybrid collision avoidance for ASVs compliant with COLREGs rules 8 and 13–17 ». In : *Frontiers in Robotics and AI* 7 (2020), p. 11 (cf. p. 139, 143).

[66] A. Fantechi, W. Fokkink et A. Morzenti. « Some trends in formal methods applications to railway signaling ». In : *Formal methods for industrial critical systems : a survey of applications* (2013), p. 61-84 (cf. p. 17).

[67] S. Flake, W. Müller et J. Ruf. « Structured English for Model Checking Specification. » In : *MBMV*. 2000, p. 99-108 (cf. p. 59).

[68] D. Fox, W. Burgard et S. Thrun. « The dynamic window approach to collision avoidance ». In : *IEEE Robotics & Automation Magazine* 4.1 (1997), p. 23-33 (cf. p. 132).

[69] E. Gamma et al. *Elements of reusable object-oriented software*. T. 99. Addison-Wesley Reading, Massachusetts, 1995 (cf. p. 52).

[70] S. Gerhart, D. Craigen et T. Ralston. « Experience with formal methods in critical systems ». In : *IEEE Software* 11.1 (1994), p. 21-28 (cf. p. 5).

[71] D. Giannakopoulou et K. Havelund. « Automata-based verification of temporal properties on running programs ». In : *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, p. 412-416 (cf. p. 42).

[72] S. Gnesi et T. Margaria. *Formal methods for industrial critical systems : A survey of applications*. John Wiley & Sons, 2012 (cf. p. 5).

[73] K. Gödel et al. *Le théorème de Gödel*. T. 1989. Seuil, 1989 (cf. p. 23).

[74] V. Gruhn et R. Laue. « Patterns for timed property specifications ». In : *Electronic Notes in Theoretical Computer Science* 153.2 (2006), p. 117-133 (cf. p. 9, 20, 35).

[75] L. Grunske. « Specification patterns for probabilistic quality properties ». In : *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE. 2008, p. 31-40 (cf. p. 36, 51, 59).

[76] A. Hagerer et al. « Model generation by moderated regular extrapolation ». In : *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2002, p. 80-95 (cf. p. 17).

[77] N. Halbwachs, F. Lagnier et P. 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*. AMAST '93. Berlin, Heidelberg : Springer-Verlag, 1994, p. 83-96. ISBN : 3-540-19852-0. URL : `http://dl.acm.org/citation.cfm?id=646055.677894` (cf. p. 19, 36, 37, 42).

[78] N. Halbwachs, F. Lagnier et P. Raymond. « Synchronous observers and the verification of reactive systems ». In : *Algebraic Methodology and Software Technology (AMAST'93)*. Springer, 1994, p. 83-96 (cf. p. 19, 42).

[79] D. Hendrix et al. « Model-based robustness analysis of indoor lighting systems ». In : (2015) (cf. p. 17).

[80] T. Henties et al. « Java for safety-critical applications ». In : *2nd international workshop on the certification of safety-critical software controlled systems (SafeCert 2009)*. Citeseer. 2009 (cf. p. 16).

[81] C. A. R. Hoare. « Communicating sequential processes ». In : *Communications of the ACM* 21.8 (1978), p. 666-677 (cf. p. 40).

[82] G. Huet, G. Kahn et C. Paulin-Mohring. « The Coq proof assistant : a tutorial : version 7.2 ». In : (2002) (cf. p. 22, 109).

[83] M. Huisman, D. Gurov et A. Malkis. « Formal methods : from academia to industrial practice. A travel guide ». In : *arXiv preprint arXiv :2002.07279* (2020) (cf. p. 17).

[84] *IMO : Convention on the international regulations for preventing collisions at sea (COLREGs)*. (1972) (cf. p. 133).

[85] W. Janssen et al. « Model checking for managers ». In : *International SPIN Workshop on Model Checking of Software*. Springer. 1999, p. 92-107 (cf. p. 33, 35, 45).

[86] C. Jard, J.-F. Monin et R. Groz. « Development of VEDA, a prototyping tool for distributed algorithms ». In : *IEEE Transactions on Software Engineering* 14.3 (1988), p. 339-352 (cf. p. 42).

[87] D. Kapur, X. Nie et D. R. Musser. « An overview of the Tecton proof system ». In : *Theoretical Computer Science* 133.2 (1994), p. 307-339 (cf. p. 22).

[88] P. Kars. « The application of Promela and Spin in the BOS project. » In : *The Spin Verification System*. 1996, p. 51-63 (cf. p. 17).

[89] J.-F. Kempf. « On computer-aided design-space exploration for multi-cores ». Thèse de doct. Université de Grenoble, 2012 (cf. p. 41).

[90] G. A. Kildall. « A unified approach to global program optimization ». In : *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1973, p. 194-206 (cf. p. 21).

[91] F. Kirchner et al. « Frama-C : A software analysis perspective ». In : *Formal Aspects of Computing* 27.3 (2015), p. 573-609 (cf. p. 16).

[92] G. Klein et al. « seL4 : Formal verification of an OS kernel ». In : *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, p. 207-220 (cf. p. 17).

[93] S. Konrad et B. H. Cheng. « Real-time specification patterns ». In : *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, p. 372-381 (cf. p. 4, 17-19, 35, 36, 51, 56, 59).

[94] D. K. M. Kufoalor, E. F. Brekke et T. A. Johansen. « Proactive collision avoidance for ASVs using a dynamic reciprocal velocity obstacles method ». In : *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, p. 2402-2409 (cf. p. 139).

[95] M. Kutila et al. « Automotive LiDAR performance verification in fog and rain ». In : *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2018, p. 1695-1701 (cf. p. 133).

[96] K. G. Larsen, P. Pettersson et W. Yi. « UPPAAL in a nutshell ». In : *International journal on software tools for technology transfer* 1.1 (1997), p. 134-152 (cf. p. 40).

[97] S. M. LaValle et al. « Rapidly-exploring random trees : A new tool for path planning ». In : (1998) (cf. p. 132).

[98] S.-M. Lee, K.-Y. Kwon et J. Joongseon. « A fuzzy logic for autonomous navigation of marine vehicles satisfying COLREG guidelines ». In : *International Journal of Control, Automation, and Systems* 2.2 (2004), p. 171-181 (cf. p. 134).

[99] X. Leroy et al. *The CompCert memory model*. 2014 (cf. p. 17).

[100] L. Li et Y. Li. « MODEL-CHECKING OF LINEAR-TIME PROPERTIES IN POSSIBILISTIC KRIPKE STRUCTURE ». In : *Quantitative Logic And Soft Computing*. World Scientific, 2012, p. 287-294 (cf. p. 23).

[101] P. Liggesmeyer et O. Maeckel. « Quantifying the reliability of embedded systems by automated analysis ». In : *2001 International Conference on Dependable Systems and Networks*. IEEE. 2001, p. 89-94 (cf. p. 5).

[102] G. Lowe. « Breaking and fixing the Needham-Schroeder public-key protocol using FDR ». In : *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1996, p. 147-166 (cf. p. 17).

[103] M. Lumpe, I. Meedeniya et L. Grunske. « PSPWizard : machine-assisted definition of temporal logical properties with specification patterns ». In : *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, p. 468-471 (cf. p. 17, 19).

[104] K. L. McMillan. « Verification of infinite state systems by compositional model checking ». In : *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 1999, p. 219-237 (cf. p. 25).

[105] C. Meadows. « Emerging issues and trends in formal methods in cryptographic protocol analysis : Twelve years later ». In : *Logic, rewriting, and concurrency*. Springer, 2015, p. 475-492 (cf. p. 17).

[106] R. Milner. *Communication and concurrency*. T. 84. Prentice hall Englewood Cliffs, 1989 (cf. p. 40).

[107] A. Miné. « Static analysis of embedded real-time concurrent software with dynamic priorities ». In : *Electronic Notes in Theoretical Computer Science* 331 (2017), p. 3-39 (cf. p. 16).

[108] M. Moskal et al. « Verifying C programs : a VCC tutorial ». In : *MSR Redmond, EMIC Aachen* (2012) (cf. p. 17).

[109] Y. Moy et al. « Testing or formal verification : Do-178c alternatives and industrial experience ». In : *IEEE software* 30.3 (2013), p. 50-57 (cf. p. 16).

[110] C. Newcombe et al. « Use of formal methods at Amazon Web Services ». In : *See http ://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf* (2014) (cf. p. 16).

[111] S. Owre et al. « PVS : Combining specification, proof checking, and model checking ». In : *International Conference on Computer Aided Verification*. Springer. 1996, p. 411-414 (cf. p. 25).

[112] C. Ponsard, J.-C. Deprez et R. De Landtsheer. « High-level guidance for managers deploying formal methods in their organisation ». In : *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2013, p. 139-153 (cf. p. 17).

[113] J.-P. Queille et J. Sifakis. « Specification and verification of concurrent systems in CESAR ». In : *International Symposium on programming*. Springer. 1982, p. 337-351 (cf. p. 23).

[114] S. Rabin. « Game programming gems, chapter a* aesthetic optimizations ». In : *Charles River Media* (2000) (cf. p. 132).

[115] S. Rajan, N. Shankar et M. K. Srivas. « An integration of model checking with automated proof checking ». In : *International Conference on Computer Aided Verification*. Springer. 1995, p. 84-97 (cf. p. 25).

[116] A. Raji et P. Dhaussy. « Use cases modeling for scalable model-checking ». In : *2011 18th Asia-Pacific Software Engineering Conference*. IEEE. 2011, p. 65-72 (cf. p. 32).

[117] J.-C. Roger. « Exploitation de contextes et d'observateurs pour la validation formelle de modèles ». Thèse de doct. Télécom Bretagne, 2006 (cf. p. 6, 32, 37, 42, 43).

[118] J. Rushby. *Formal methods and the certification of critical systems*. T. 37. SRI International, Computer Science Laboratory, 1993 (cf. p. 5).

[119] J. Rushby. « From refutation to verification ». In : *Formal Methods for Distributed System Development*. Springer, 2000, p. 369-374 (cf. p. 23).

[120] J. Rushby. « Theorem proving for verification ». In : *Summer School on Modeling and Verification of Parallel Processes*. Springer. 2000, p. 39-57 (cf. p. 22).

[121] R. L. Smith et al. « Propel : an approach supporting property elucidation ». In : *Proceedings of the 24th International Conference on Software Engineering*. ACM. 2002, p. 11-21 (cf. p. 4, 20, 34, 44, 56).

[122] J. Souyris et al. « Formal verification of avionics software products ». In : *International symposium on formal methods*. Springer. 2009, p. 532-546 (cf. p. 16).

[123] A. Steed et L. Delligatti. *Software Requirements Specification for Akhbar Neb Security System*. http://mysite.du.edu/~lbarne28/SE/project/GroupF-SRS-FireSecurityAlarmSystem_.pdf. Accessed : 2019-01-26 (cf. p. 96).

[124] B. Steffen et T. Margaria. « META Frame in practice : design of intelligent network services ». In : *Correct System Design*. Springer, 1999, p. 390-415 (cf. p. 17).

[125] R. Szlapczynski et J. Szlapczynska. « Review of ship safety domains : Models and applications ». In : *Ocean Engineering* 145 (2017), p. 277-289 (cf. p. 139).

[126] S. Taha et al. « A compositional automata-based semantics and preserving transformation rules for testing property patterns ». In : *Formal Aspects of Computing* 27.4 (2015), p. 641-664 (cf. p. 20, 86, 94, 98, 103).

[127] C. Teodorov, L. Leroux et P. Dhaussy. « Context-aware verification of a cruise-control system ». In : *International Conference on Model and Data Engineering*. Springer. 2014, p. 53-64 (cf. p. 32).

[128] W. Thomas. « Automata on infinite objects ». In : *Formal Models and Semantics*. Elsevier, 1990, p. 133-191 (cf. p. 98).

[129] J. Tretmans, K. Wijbrans et M. Chaudron. « Software engineering with formal methods : The development of a storm surge barrier control system revisiting seven myths of formal methods ». In : *Formal Methods in System Design* 19.2 (2001), p. 195-215 (cf. p. 17).

[130] L. Trinh, M. Ekström et B. Çürüklü. « Dipole flow field for dependable path planning of multiple agents ». In : *IEEE/RSJ International Conference on Intelligent Robots and Systems IROS*. T. 24. 2017 (cf. p. 132).

[131] B. Weyers et al. *The handbook of formal methods in human-computer interaction*. Springer, 2017 (cf. p. 17).

[132] K. L. Woerner et al. « Collision avoidance road test for COLREGS-constrained autonomous vehicles ». In : *OCEANS 2016 MTS/IEEE Monterey*. IEEE. 2016, p. 1-6 (cf. p. 134).

[133] J. Woodcock et al. « Formal methods : Practice and experience ». In : *ACM computing surveys (CSUR)* 41.4 (2009), p. 1-36 (cf. p. 17).

[134] J. Yu et al. « Pattern based property specification and verification for service composition ». In : *International Conference on Web Information Systems Engineering*. Springer. 2006, p. 156-168 (cf. p. 21, 94).