# EMI - Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable : application aux modèles UML des systèmes embarqués

Valentin Besnard

▶ To cite this version:

Valentin Besnard. EMI - Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable : application aux modèles UML des systèmes embarqués. Génie logiciel [cs.SE]. ENSTA Bretagne - École nationale supérieure de techniques avancées Bretagne, 2020. Français. NNT : 2020ENTA0007 . tel-03371484

HAL Id: tel-03371484
https://theses.hal.science/tel-03371484

Submitted on 8 Oct 2021

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

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

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA BRETAGNE

# THÈSE DE DOCTORAT DE

L'ENSTA BRETAGNE

ÉCOLE DOCTORALE N° 601
*Mathématiques et Sciences et Technologies de l'Information et de la Communication*
Spécialité : *Informatique*


Par

**Valentin BESNARD**


# EMI : Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable

## Application aux modèles UML des systèmes embarqués


**Thèse présentée et soutenue à Angers, le 9 décembre 2020**
**Unité de recherche : Lab-STICC UMR CNRS 6285**


**Rapporteurs avant soutenance :**

| | |
|---|---|
| Frédéric BONIOL | Professeur des Universités, ONERA, DTIS |
| Benoît COMBEMALE | Professeur des Universités, Université de Rennes 1, IRISA |

**Composition du Jury :**

| | | |
|---|---|---|
| Présidente : | Isabelle BORNE | Professeur des Universités, Université Bretagne Sud, IRISA |
| Rapporteurs : | Frédéric BONIOL | Professeur des Universités, ONERA, DTIS |
| | Benoît COMBEMALE | Professeur des Universités, Université de Rennes 1, IRISA |
| Examinateurs : | Julien DEANTONI | Maître de conférences, HDR, Université Côte d'Azur, I3S |
| | Frédéric JOUAULT | Enseignant-Chercheur, ESEO |
| Directeur de thèse : | Philippe DHAUSSY | Enseignant-Chercheur, HDR, ENSTA Bretagne, Lab-STICC |
| Encadrants de thèse : | Matthias BRUN | Enseignant-Chercheur, ESEO |
| | Ciprian TEODOROV | Enseignant-Chercheur, ENSTA Bretagne, Lab-STICC |

**Invité :**

| | |
|---|---|
| David OLIVIER | Directeur R&D, Davidson Consulting |

# REMERCIEMENTS

Je souhaite tout d'abord remercier mes encadrants de thèse Matthias BRUN et Ciprian TEODOROV pour la confiance qu'ils m'ont accordée durant ces trois années très instructives passées à leurs côtés. Ils ont su créer un environnement de travail idéal à mon épanouissement personnel et à la réalisation de ce doctorat. En plus du soutien qu'ils m'ont apporté, Matthias et Ciprian ont su guider et orienter mes avancées scientifiques avec une grande perspicacité.

J'adresse également toute ma reconnaissance à Frédéric JOUAULT qui m'a accompagné au quotidien pendant ces trois années de thèse. Ses conseils avisés, ses remarques pertinentes et sa rigueur scientifique exemplaire ont permis de tirer le meilleur de moi-même tout au long de ce doctorat. Merci à lui pour le savoir qu'il m'a transmis et l'attention qu'il m'a accordée.

Je tiens à remercier mon directeur de thèse, Philippe DHAUSSY, pour ses conseils éclairés, sa bienveillance et son écoute qui ont permis de définir sereinement les grands axes de mes travaux de recherche.

Mes remerciements vont également à Benoît COMBEMALE et Frédéric BONIOL qui m'ont fait l'honneur d'évaluer mon travail de thèse et d'en être les rapporteurs. Je remercie aussi Isabelle BORNE et Julien DEANTONI d'avoir accepté de faire partie des membres du jury.

Je souhaite remercier Sébastien FAUCOU et Franck SINGHOFF d'avoir participé à mes comités de suivi individuel de thèse. Leurs conseils avisés m'ont permis de prendre du recul sur mes travaux et de mieux définir le contexte dans lequel ils s'inscrivent.

Ce travail a été financé par la société Davidson Consulting sans qui les résultats de ces travaux de recherche n'auraient pu voir le jour. Aussi, je tiens à remercier Davidson Consulting et en particulier David OLIVIER, le directeur R&D, pour son soutien, son expertise et sa bonne humeur communicative.

Ce projet de recherche a également été mené en collaboration avec l'ESEO et l'ENSTA Bretagne. Mes remerciements vont donc à l'ensemble de mes collègues des départements DIS et STIC avec qui j'ai pu évoluer au cours de ces trois années. Je remercie plus largement l'ensemble du personnel qui m'a accompagné dans les activités d'enseignement et de recherche. Mes remerciements vont également à l'ensemble des étudiants qui ont pris part à ces travaux de doctorat à travers la réalisation de cas d'études.

Je tiens également à remercier tous les membres de la communauté scientifique qui ont contribué par leurs remarques à l'amélioration de l'approche EMI et des publications associées. Je remercie en particulier les membres de l'initiative Gemoc avec qui j'ai pu avoir des discussions très enrichissantes sur l'analyse et l'exécution de modèles.

Je profite de ces remerciements pour saluer tous mes amis des Mauges, de l'ESEO, du club robot ou d'ailleurs pour m'avoir encouragé durant ces trois années. Vous êtes trop nombreux pour pouvoir être tous cités, mais vous vous reconnaîtrez sûrement à travers ces quelques lignes. Merci d'avoir été là !

J'adresse également toute ma reconnaissance à mes parents, mon frère et ma sœur pour leur soutien sans faille tout au long de mes études et plus particulièrement pendant ces trois années de thèse. Je remercie plus largement toute ma famille et ma belle-famille pour leurs encouragements et pour tout ce qu'ils ont fait pour moi. Cette réussite est aussi la vôtre !

Enfin, je souhaite adresser un remerciement tout particulier à celle qui partage ma vie depuis quelques années déjà. Merci à elle pour la patience qu'elle m'a accordée et pour l'amour qu'elle a su me prodiguer tout au long de ces trois années.

# Table des matières

**Introduction** 1

Contexte . . . . . 1
Problématiques . . . . . 2
Contributions . . . . . 3
Plan du manuscrit . . . . . 6
Liste des publications . . . . . 9

**I État de l'art** 13

**1 L'exécution de programmes** 15

1.1 Introduction . . . . . 15
1.2 Définition d'un langage . . . . . 16
1.2.1 Définition . . . . . 16
1.2.2 Sémantique structurelle vs comportementale . . . . . 18
1.2.3 Les différents styles de définition de la sémantique comportementale . . 18
1.2.3.1 Sémantique opérationnelle . . . . . 19
1.2.3.2 Sémantique dénotationnelle . . . . . 20
1.2.3.3 Sémantique translationnelle . . . . . 20
1.2.3.4 Sémantique axiomatique . . . . . 21
1.3 Approches pour l'exécution de programmes . . . . . 22
1.3.1 Définition des relations "conforme à" et "implémente" . . . . . 22
1.3.2 Approches basées sur une sémantique translationnelle . . . . . 23
1.3.2.1 Description . . . . . 23
1.3.2.2 Outils . . . . . 24
1.3.3 Approches basées sur une sémantique opérationnelle . . . . . 26
1.3.3.1 Description . . . . . 26
1.3.3.2 Outils . . . . . 27
1.3.4 Synthèse . . . . . 29

**2 L'analyse et la vérification de programmes** **31**
2.1 Introduction . . . 31
2.2 Panorama des techniques de V&V . . . 32
2.3 Vérification formelle . . . 34
2.3.1 Principes . . . 34
2.3.2 Preuve d'équivalence . . . 35
2.4 Le *model-checking* . . . 36
2.4.1 Principes du *model-checking* . . . 36
2.4.2 Mise en œuvre du *model-checking* . . . 38
2.4.3 Avantages et limites du *model-checking* . . . 40
2.5 Approches pour l'analyse et la vérification logicielle . . . 41
2.5.1 Approches par raffinement . . . 41
2.5.1.1 Description . . . 41
2.5.1.2 Langages et outils . . . 43
2.5.2 Approches avec une transformation vers un langage d'analyse . . . 44
2.5.2.1 Description . . . 44
2.5.2.2 Langages et outils . . . 45
2.5.3 Approches d'analyse dédiées au langage . . . 47
2.5.3.1 Description . . . 47
2.5.3.2 Outils . . . 48
2.5.4 Approches d'analyse par API . . . 49
2.5.4.1 Description . . . 49
2.5.4.2 Outils et protocoles . . . 50
2.6 Synthèse . . . 51

**3 Approches pour l'exécution et l'analyse de modèles** **53**
3.1 Introduction . . . 53
3.2 Contexte de développement . . . 54
3.3 Problèmes pour l'analyse et l'exécution réelle . . . 55
3.3.1 Fossés sémantiques . . . 55
3.3.2 Problème d'équivalence . . . 56
3.4 Approche 1 : traduction vers modèle vérifiable et code . . . 58
3.5 Approche 2 : traduction vers modèle vérifiable puis code . . . 59
3.6 Approche 3 : traduction vers code vérifiable . . . 60
3.7 Approche 4 : traduction modèle vérifiable vers code . . . 61
3.8 Approche 5 : raffinement de modèles jusqu'au code . . . 62
3.9 Approche 6 : interprétations spécifiques pour vérification et exécution réelle . . . 63
3.10 Analyse des six approches . . . 64

3.11 Approche X : interprétation unifiée pour vérification et exécution réelle . . . . . . 65
3.12 Objectifs de recherche . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
3.13 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**II L'approche EMI** **71**

**4 Présentation de l'approche EMI** **72**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.2 Description de l'architecture candidate . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.3 Un contrôleur d'exécution pour chaque activité . . . . . . . . . . . . . . . . . . . . 73
4.4 L'interface STR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.5 Analyse de modèles avec un interpréteur EMI . . . . . . . . . . . . . . . . . . . . . 75
4.6 Exécution embarquée de modèles avec un interpréteur EMI . . . . . . . . . . . . . 77
4.7 Présentation des différents modes d'exécution . . . . . . . . . . . . . . . . . . . . 78
4.8 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**5 Analyse de l'exécution de modèles** **80**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.2 Abstraction de l'environnement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.3 Simulation, débogage, et détection de *deadlocks* . . . . . . . . . . . . . . . . . . . 82
5.3.1 Simulation interactive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.3.2 Débogage multivers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.3.3 Détection de *deadlocks* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.4 Architecture de vérification formelle . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.4.1 Automates de Büchi et automates observateurs . . . . . . . . . . . . . . . . 88
5.4.2 Les interfaces APC et $A_{cc}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.4.3 Description de l'architecture conceptuelle . . . . . . . . . . . . . . . . . . . 89
5.4.4 Architecture conceptuelle pour le *model-checking* . . . . . . . . . . . . . . 90
5.4.5 Architecture conceptuelle pour le *monitoring* . . . . . . . . . . . . . . . . . 92
5.5 Composition synchrone . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.5.1 Description théorique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.5.2 Optimisation pour le *monitoring* . . . . . . . . . . . . . . . . . . . . . . . . 95
5.6 *Model-checking* avec des propriétés LTL . . . . . . . . . . . . . . . . . . . . . . . . 96
5.6.1 Spécification de propriétés LTL . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.6.2 Architecture logicielle pour la vérification LTL . . . . . . . . . . . . . . . . . 97
5.7 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

**6 L'ordonnancement pour la vérification et l'exécution embarquée de modèles** **100**

6.1 Introduction . . . 100
6.2 Non-déterminisme . . . 101
6.3 Formalisation des opérateurs . . . 102
6.3.1 Opérateur de filtrage . . . 102
6.3.2 Opérateur d'ordonnancement . . . 104
6.3.3 Opérateur de composition asynchrone . . . 105
6.4 Vérification formelle et exécution embarquée avec ordonnancement . . . 106
6.4.1 Exécution réelle . . . 106
6.4.2 *Model-checking* avec filtrage . . . 109
6.4.3 *Model-checking* avec l'ordonnanceur dans la boucle de vérification . . . 110
6.4.4 *Model-checking* avec un environnement découplé . . . 112
6.5 Synthèse . . . 114

**7 Pilotage et observation de l'exécution de modèles 116**
7.1 Introduction . . . 116
7.2 Pilotage de l'interpréteur . . . 117
7.3 Le langage d'observation . . . 119
7.3.1 Présentation du langage d'observation . . . 119
7.3.2 Les opérateurs du langage d'observation . . . 122
7.3.3 Évaluation des prédicats . . . 123
7.4 Architecture logicielle d'analyse avec un interpréteur EMI . . . 123
7.4.1 Description de l'architecture logicielle . . . 123
7.4.2 Le serveur de langages . . . 124
7.4.3 Canonisation de la configuration . . . 125
7.5 Synthèse . . . 125

**8 Conception d'un interpréteur de modèles embarqué et pilotable 126**
8.1 Introduction . . . 126
8.2 Méthodologie de conception . . . 127
8.3 Métamodélisation . . . 129
8.3.1 Métamodélisation du langage de conception . . . 129
8.3.2 Métamodélisation des données d'exécution . . . 129
8.3.3 Le modèle objet et l'instanciation . . . 130
8.3.4 La notion de configuration . . . 133
8.3.5 Sérialisation des métamodèles . . . 135
8.4 Implémentation de la sémantique opérationnelle . . . 135
8.4.1 Implémentation d'une sémantique pilotable . . . 135
8.4.2 Une sémantique utilisable en vérification formelle . . . 136

8.5 Chargement du modèle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
8.6 Déploiement sur la plateforme d'exécution . . . . . . . . . . . . . . . . . . . . . 138
8.7 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139

**9 Unification du *model-checking* et du *monitoring* de spécifications UML exécutables 141**
9.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
9.2 Expression de propriétés dans le langage de modélisation . . . . . . . . . . . . 142
9.2.1 Modélisation de propriétés en UML . . . . . . . . . . . . . . . . . . . . . 142
9.2.2 Modélisation des automates de Büchi en UML . . . . . . . . . . . . . . . 143
9.2.3 Modélisation des automates observateurs en UML . . . . . . . . . . . . . 145
9.2.4 Conversion des propriétés LTL en PUSMs . . . . . . . . . . . . . . . . . 146
9.3 Composition synchrone . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
9.3.1 Mise en œuvre de la composition synchrone . . . . . . . . . . . . . . . . 148
9.3.2 Ajout de transitions implicites . . . . . . . . . . . . . . . . . . . . . . . 148
9.3.3 Composition synchrone avec un automate de Büchi . . . . . . . . . . . . 150
9.3.4 Composition synchrone avec un automate observateur . . . . . . . . . . 150
9.4 *Model-checking* et *monitoring* avec des machines à états UML . . . . . . . . . . 150
9.4.1 Architecture de *model-checking* avec des PUSMs . . . . . . . . . . . . . 151
9.4.2 Architecture de *monitoring* avec des PUSMs . . . . . . . . . . . . . . . . 152
9.5 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154

**III Évaluation de l'approche 157**

**10 Expérimentations 158**
10.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
10.2 Présentation de l'outil EMI-UML . . . . . . . . . . . . . . . . . . . . . . . . . 159
10.3 Présentation des cas d'études . . . . . . . . . . . . . . . . . . . . . . . . . . 161
10.3.1 Contrôleur de passage à niveau . . . . . . . . . . . . . . . . . . . . . . 161
10.3.2 Interface d'un régulateur de vitesse . . . . . . . . . . . . . . . . . . . . 162
10.4 Mise en œuvre des activités d'analyse de modèles . . . . . . . . . . . . . . . . 163
10.4.1 Simulation interactive . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
10.4.2 Débogage multivers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
10.4.3 *Model-checking* LTL et détection de *deadlocks* . . . . . . . . . . . . . . 166
10.4.4 *Model-checking* avec l'ordonnanceur . . . . . . . . . . . . . . . . . . . 169
10.4.5 *Model-checking* avec des PUSMs . . . . . . . . . . . . . . . . . . . . . 172
10.4.5.1 Spécification de PUSMs . . . . . . . . . . . . . . . . . . . . . . 172

10.4.5.2 *Model-checking* du comportement du système . . . . . . . . . . 174
10.4.6 *Model-checking* des mécanismes de sûreté de fonctionnement . . . . . . 176
10.4.7 Exécution embarquée et *monitoring* . . . . . . . . . . . . . . . . . . . . . 177
10.4.8 Évaluation des performances d'exécution . . . . . . . . . . . . . . . . . . 178
10.5 Expérimentations avec d'autres outils . . . . . . . . . . . . . . . . . . . . . . 180
10.5.1 EMI-LOGO : un interpréteur de modèles LOGO . . . . . . . . . . . . . . . 180
10.5.2 AnimUML : un outil d'interprétation et d'animation de modèles UML . . . 181
10.5.3 Intégration d'EMI-UML dans Gemoc Studio . . . . . . . . . . . . . . . . . 183
10.6 Synthèse des expérimentations . . . . . . . . . . . . . . . . . . . . . . . . . 184
10.7 Contributions scientifiques . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
10.7.1 C#1 Contribution à l'unification de l'analyse et de l'exécution embarquée de modèles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
10.7.2 C#2 Contribution à l'adoption des techniques de vérification formelle par les ingénieurs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187
10.8 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188

**Conclusion** **189**
Rappel du contexte et des problèmes considérés . . . . . . . . . . . . . . . . . . 189
Solution proposée . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
Perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 192

**Bibliographie** **197**

**Annexes** **228**

**A Le langage UML et ses fondements en IDM** **229**
A.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229
A.2 L'ingénierie dirigée par les modèles . . . . . . . . . . . . . . . . . . . . . . . 229
A.2.1 Principes de l'IDM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 230
A.2.1.1 Les relations clés de l'IDM . . . . . . . . . . . . . . . . . . . . . 230
A.2.1.2 L'approche MDA . . . . . . . . . . . . . . . . . . . . . . . . . . . 231
A.2.1.3 Transformation de modèles . . . . . . . . . . . . . . . . . . . . . 233
A.2.2 Les langages en IDM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 234
A.3 Le langage UML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 235
A.3.1 Historique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 235
A.3.2 Le standard UML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 235
A.3.3 Techniques d'exécution de modèles UML . . . . . . . . . . . . . . . . . . 237
A.3.4 Techniques d'analyse de modèles UML . . . . . . . . . . . . . . . . . . . 239

A.4 Synthèse . . . . . . . . . . 240

**B Formalisation en Lean** **242**

**C Langage d'action et langage d'observation pour les modèles UML** **252**

C.1 Langage d'expression . . . . . . . . . . 253
C.2 Langage d'effet . . . . . . . . . . 253
C.3 Extension du langage d'expression . . . . . . . . . . 254

**D Présentation des cas d'études** **257**

D.1 Contrôleur de passage à niveau . . . . . . . . . . 257
D.2 Interface du régulateur de vitesse . . . . . . . . . . 259

**E Liste des propositions atomiques** **266**

E.1 Liste des propositions atomiques pour le modèle du contrôleur de passage à niveau. . . . . . . . . . . 266
E.2 Liste des propositions atomiques pour le modèle d'interface du régulateur de vitesse. . . . . . . . . . . 267

**F Pages web archivées** **270**

**Acronymes** **272**

**Table des figures** **277**

**Liste des tableaux** **281**

**Liste des listings** **283**

# INTRODUCTION

## Contexte

Les systèmes informatiques deviennent de plus en plus complexes à concevoir car ils offrent davantage de fonctionnalités et sont désormais interconnectés par exemple via les réseaux de l'internet des objets (ou en anglais *Internet of Things*). Ces systèmes sont omniprésents dans notre vie quotidienne. Ils doivent donc être à la fois fiables et performants pour répondre aux exigences de plus en plus fortes de notre société en termes de déplacement, de communication ou encore d'automatisation. Les besoins croissants en logiciels tendent ainsi à réduire les temps de développement pour proposer ces produits plus rapidement sur le marché. En conséquence, la complexité croissante de ces systèmes et la réduction du temps de développement augmentent les risques d'erreurs de conception, de bogues logiciels, de comportements indéfinis et l'exposition de ces systèmes aux failles de sécurité. Ces défaillances sont également plus difficiles à identifier, à comprendre et à résoudre en raison de la complexité inhérente de ces systèmes.

De nombreux bogues logiciels sont survenus au cours de l'histoire de l'informatique. Voici quelques-uns des plus célèbres [BK08]. Lors du lancement de la fusée Ariane 5 le 4 juin 1996, la fusée a explosé 36 secondes après son lancement à la suite d'un problème de conversion d'un nombre flottant de 64 bits dans un entier de 16 bits. Au début des années 90, un bogue dans l'unité de division flottante du processeur Intel Pentium II a entraîné une perte de 475 millions de dollars pour remplacer tous les processeurs défectueux. Une erreur logicielle dans le système de gestion des bagages a repoussé l'ouverture de l'aéroport de Denver de 9 mois et entraîné une perte de 1,1 million de dollars par jour.

Par rapport au coût de correction d'un bogue en phase de spécification, le coût de correction est 2 à 6 fois plus important en phase de conception, 20 à 100 fois plus important en phase de développement, 500 à 1000 fois plus important en phase de test ou de production et peut encore être beaucoup plus important en phase de maintenance [BK08]. Il est donc primordial de détecter ces bogues au plus tôt c.-à-d. dans les phases amont de conception.

L'objectif est de développer des systèmes plus complexes avec moins d'erreurs et dans un intervalle de temps plus court. En conséquence, les besoins en activités de Vérification et de Validation (V&V) sont de plus en plus importants pour vérifier et valider le comportement de ces systèmes logiciels.

Une catégorie de systèmes s'identifiant particulièrement bien à ces forts besoins de V&V

est celle des systèmes embarqués. Les systèmes embarqués font partie de la classe des systèmes réactifs c.-à-d. des systèmes dont le fonctionnement réagit à des sollicitations de l'environnement extérieur. Ce sont des systèmes enfouis permettant d'embarquer des composants informatiques à des endroits où l'on ne soupçonne pas leur existence. De nombreux systèmes embarqués doivent être conçus pour répondre aux défis de demain comme les véhicules autonomes, les objets connectés, l'industrie 4.0 ou encore la ville intelligente. Ces systèmes se complexifient et peuvent maintenant s'organiser sous forme de systèmes cyber-physiques c.-à-d. de composants logiciels communicants devant collaborer avec des entités physiques au travers de lois de contrôle-commande pour remplir les fonctionnalités du système global.

Les logiciels embarqués ont généralement des cycles de conception lourds (p. ex. le cycle en V) faisant intervenir de nombreux ingénieurs avec des expertises multiples. La communication entre ces différents experts n'est pas toujours facile car chacun a un point de vue spécifique. En plus des défaillances usuelles, des problèmes de communication ou d'interopérabilité apparaissent généralement aux frontières des différentes briques logicielles. Par ailleurs, ces systèmes possèdent potentiellement des fonctionnalités critiques telles que toute défaillance a des conséquences dramatiques au niveau humain, économique ou environnemental. Pour assurer la sûreté de fonctionnement de ces systèmes, leurs développements nécessitent donc de nombreuses activités de V&V, des efforts de test très coûteux, et de forts besoins en certification.

## Problématiques

Pour concevoir un système logiciel (embarqué), un langage de modélisation est utilisé pour définir un modèle de conception de ce système. À partir de ce modèle de conception, les ingénieurs doivent mener des activités d'analyse (p. ex. simulation) afin de vérifier que le comportement du modèle du système satisfait ses exigences. À ce titre, cette thèse s'intéresse en particulier à une technique de vérification formelle, appelée *model-checking*, permettant de vérifier une propriété de manière exhaustive sur l'espace d'état d'un modèle. Une fois vérifié, ce modèle doit également pouvoir être déployé sur une cible embarquée pour l'exécution réelle du système. Toutes ces activités de développement logiciel reposent sur des transformations du modèle de conception (p. ex. transformations de modèles, génération de code) vers différents langages cibles permettant d'analyser l'exécution du modèle ou d'obtenir le code exécutable pouvant être déployé sur une plateforme embarquée (p. ex. un microcontrôleur).

Avec les techniques classiques d'analyse et d'exécution de modèles, trois problèmes majeurs subsistent. P#1 Un fossé sémantique entre le modèle de conception et le(s) modèle(s) d'analyse rend plus complexes l'utilisation des techniques d'analyse (notamment celles de *model-checking*) et la compréhension des résultats de vérification. P#2 Un fossé sémantique

entre le modèle de conception et le code exécutable rend plus difficile d'établir des correspondances entre les concepts de conception et ceux d'exécution. P#3 Une relation d'équivalence entre le(s) modèle(s) d'analyse et le code exécutable doit être établie, prouvée et maintenue pour assurer que ce qui est exécuté est bien ce qui a été vérifié. L'utilisation de transformations non prouvées est à l'origine de ces trois problèmes. Chaque outil de transformation capture la sémantique du langage de modélisation (c.-à-d. la signification des concepts de ce langage) créant ainsi plusieurs implémentations de la sémantique de ce langage. Chaque transformation duplique le modèle de conception créant ainsi plusieurs instances du même modèle. Lors de ces transformations, de subtiles différences peuvent être introduites soit au niveau du modèle soit au niveau de l'implémentation de la sémantique. Ces différences peuvent être à l'origine d'une modification du comportement du système ou de défaillances logicielles lors de son exécution réelle.

Cette thèse cherche donc à unifier les activités d'analyse et l'exécution réelle de modèles conformes à n'importe quel langage de modélisation. Elle cherche ainsi à garantir que ce qui est exécuté sur le système réel est bien ce qui a été vérifié lors de la phase de V&V.

## Contributions

L'approche développée dans cette thèse, appelée EMI (pour *Embedded Model Interpreter*), vise à utiliser une seule implémentation de la sémantique du langage de modélisation pour toutes les activités de développement logiciel. Le même couple (modèle + sémantique) est utilisé à la fois pour l'analyse du comportement du modèle et pour l'exécution réelle sur une cible embarquée. L'approche EMI n'utilise donc aucune transformation afin de préserver l'unicité de l'implémentation de la sémantique. Elle permet ainsi d'éviter les deux fossés sémantiques et la nécessité d'établir une relation d'équivalence.

Dans le contexte de l'Ingénierie Dirigée par les Modèles (IDM), cette approche utilise un interpréteur de modèles pour définir l'unique implémentation de la sémantique du langage de modélisation et centraliser tous les outils de développement autour de cette même définition. La spécificité de cet interpréteur est d'être pilotable c.-à-d. qu'il est possible de lui envoyer des commandes pour contrôler l'exécution du modèle et ainsi mener diverses activités d'analyse. Le pilotage est assuré via une interface de contrôle d'exécution qui offre différents services permettant de piloter, d'observer et de visualiser l'exécution du modèle. Cette interface est générique de sorte qu'elle puisse être réutilisée pour n'importe quel langage de modélisation.

Des outils d'analyse peuvent ainsi se connecter à l'interpréteur à travers cette interface pour examiner l'exécution du modèle. Pour mener des activités d'analyse, il est généralement nécessaire de fermer l'exécution du système avec une abstraction de son environnement réel (c.-à-d. de connecter ses entrées/sorties à un environnement abstrait) [DRB11 ; Dha+12] afin

de solliciter les différents aspects de son comportement. Pour réaliser cette abstraction, ce travail repose sur l'hypothèse que les ingénieurs sont capables de spécifier des modèles d'environnement exhaustifs à partir des spécifications du système. Cette hypothèse est particulièrement réaliste dans le cas des systèmes embarqués où les interactions avec l'environnement sont rigoureusement documentées. Grâce à cette abstraction de l'environnement réel, il est possible de mettre en œuvre de la simulation ou du débogage mais aussi des activités de vérification formelle comme le *model-checking* ou le *monitoring*. Dans le cadre de cette thèse, la mise en œuvre des activités de vérification formelle se borne à la vérification de propriétés non temporisées exprimées en Logique Temporelle Linéaire (LTL) ou sous forme d'automates de Büchi [BK08] ou d'automates observateurs [OGO06]. Cette architecture d'analyse basée sur l'interface de contrôle d'exécution pour faire interagir les outils d'analyse avec l'interpréteur de modèles apporte deux principaux avantages. *(i)* Les outils d'analyse bénéficient d'une interface générique leur permettant de gagner en modularité et d'être applicables sur des modèles de différents langages de modélisation. *(ii)* La compréhension des résultats d'analyse par des ingénieurs non-experts en vérification formelle peut être facilitée en exprimant ces résultats directement en termes des concepts de modélisation.

Pour l'exécution réelle, l'interpréteur et le modèle de conception peuvent également être déployés sur une cible embarquée. Il convient alors d'ordonnancer l'exécution du système afin de ne suivre qu'une seule trace d'exécution (c.-à-d. la trace d'exécution courante). Dans le cadre de cette thèse, nous considérons un ordonnanceur dans lequel il n'y a pas de création de processus et pas de prise en compte des interruptions.

Pour mettre en œuvre cette approche, la recherche scientifique effectuée dans cette thèse (Figure 1) s'est intéressée à différents axes se déclinant en trois objectifs :

O#1 Identifier les particularités d'un interpréteur permettant d'unifier l'exécution et l'analyse de modèles,

O#2 Définir l'interface nécessaire pour faire interagir des outils d'analyse avec un interpréteur de modèles pilotable,

O#3 Rendre les outils d'analyse plus génériques, plus modulaires et plus facilement utilisables par les ingénieurs.

Pour répondre à ces objectifs, différents travaux ont été réalisés pour caractériser l'approche EMI. Au travers de ces travaux, cette thèse revendique deux contributions scientifiques :

C#1 Contribution à l'unification de l'analyse et de l'exécution embarquée de modèles en se basant sur une seule implémentation de la sémantique du langage de modélisation pour garantir que ce qui est exécuté sur la cible embarquée est bien ce qui a été vérifié. Cette contribution repose sur les sous-contributions suivantes :

C#1.1 Proposition d'une interface de contrôle d'exécution permettant de faire interagir des outils d'analyse avec des moteurs d'exécution. Cette interface permet notam-

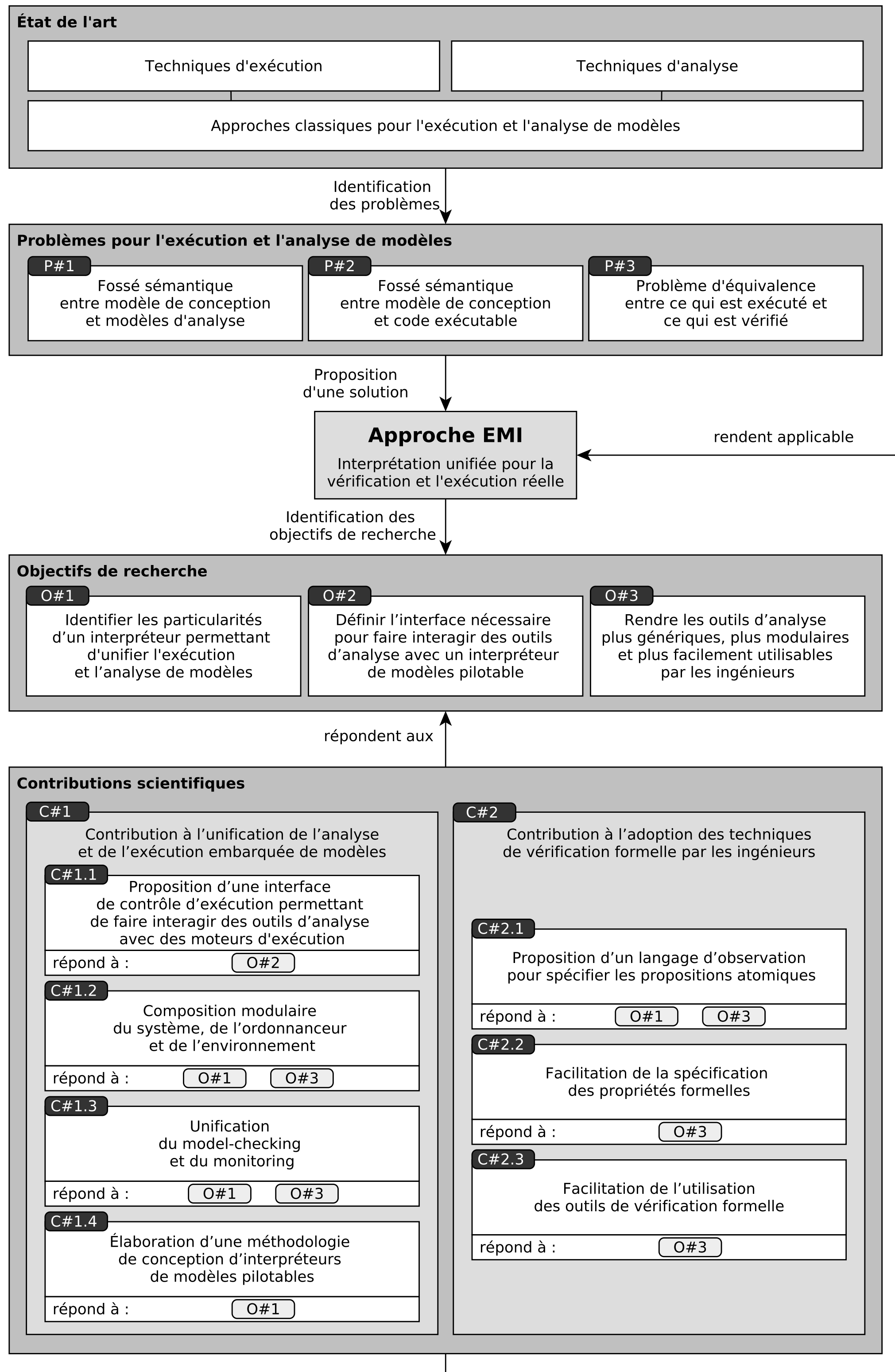



FIGURE 1 – Aperçu des problèmes, des objectifs et des contributions scientifiques.

ment un meilleur découplage des préoccupations et de rendre les outils d'analyse plus génériques.

- C#1.2 Composition modulaire du système, de l'ordonnanceur et de l'environnement au travers de différents opérateurs permettant notamment de prendre en compte l'ordonnanceur dans la boucle de vérification.
- C#1.3 Unification du *model-checking* et du *monitoring* telle que les automates utilisés en *model-checking* puissent être déployés sur une cible embarquée sans transformation ni instrumentation de code pour faire du *monitoring*.
- C#1.4 Élaboration d'une méthodologie de conception d'interpréteurs de modèles pilotables pouvant fournir des services d'analyse mais aussi être déployés sur une plateforme embarquée.

C#2 Contribution à l'adoption des techniques de vérification formelle par les ingénieurs. Cette contribution repose sur les sous-contributions suivantes :

- C#2.1 Proposition d'un langage d'observation pour spécifier les propositions atomiques (c.-à-d. les prédicats constituant les propriétés formelles) avec les concepts du langage de modélisation.
- C#2.2 Facilitation de la spécification des propriétés formelles en permettant d'exprimer ces propriétés directement dans le langage de modélisation ou dans un autre formalisme au choix de l'utilisateur.
- C#2.3 Facilitation de l'utilisation des outils de vérification formelle en rendant ces outils plus modulaires, plus configurables et en facilitant la compréhension des résultats d'analyse par les ingénieurs.

Pour évaluer l'approche EMI, un interpréteur de modèles pour le langage *Unified Modeling Language* (UML®) a été développé comme preuve de concept. Le langage UML [OMG17d] (cf. section A.3) a été choisi car il est reconnu internationalement et utilisé par de nombreux industriels pour concevoir des logiciels. Pour valider l'approche, plusieurs expérimentations ont été réalisées avec cet interpréteur sur différents cas d'études de systèmes embarqués. Pour ce manuscrit, les systèmes retenus sont un contrôleur de passage à niveau et une interface utilisateur d'un régulateur de vitesse. Diverses activités d'analyse ont pu être menées sur ces modèles dont la simulation, le débogage, la détection de *deadlocks*, le *model-checking* et le *monitoring*. Ces modèles ont également pu être déployés avec l'interpréteur UML sur une cible embarquée STM32 pour l'exécution réelle.

## Plan du manuscrit

La Figure 2 donne un aperçu du plan de ce manuscrit en faisant des liens avec les problèmes à résoudre, les questions de recherches considérées, les contributions effectuées et

les publications réalisées. Ce manuscrit s'organise de la façon suivante.

La Partie I présente un état de l'art sur l'exécution et l'analyse de modèles conformes à un langage.

- Le chapitre 1 s'intéresse à la définition d'un langage de modélisation (ou de programmation) et aux techniques permettant d'exécuter des modèles (ou des programmes) conformes à ce langage.
- Le chapitre 2 dresse un panorama des différentes activités permettant d'analyser l'exécution d'un modèle (ou d'un programme) avant de s'intéresser plus particulièrement au *model-checking*. Il présente ensuite les différentes techniques d'analyse.
- Le chapitre 3 présente les approches utilisées dans la littérature pour analyser et exécuter des modèles en combinant les techniques identifiées dans les chapitres 1 et 2.

La Partie II présente l'approche EMI et les travaux scientifiques qui ont été menés au cours de cette thèse.

- Le chapitre 4 présente l'approche EMI d'un point de vue conceptuel, l'architecture candidate retenue pour la mettre en œuvre ainsi que ses principales caractéristiques.
- Le chapitre 5 présente la mise en œuvre des différentes activités d'analyse et notamment l'architecture logicielle retenue pour appliquer du *model-checking* et du *monitoring*.
- Le chapitre 6 présente différents opérateurs permettant de composer le système, l'ordonnanceur et l'environnement de façon modulaire. Cette technique permet ainsi de prendre en compte l'ordonnanceur au cours des activités de vérification formelle.
- Le chapitre 7 présente les mécanismes permettant de piloter et d'observer l'exécution du modèle en s'appuyant notamment sur un langage d'observation permettant d'exprimer des prédicats avec les concepts du langage de modélisation.
- Le chapitre 8 présente la méthodologie retenue pour concevoir un interpréteur conforme à l'approche EMI en insistant sur les particularités liées au déploiement de l'interpréteur sur une cible embarquée et à son pilotage.
- Le chapitre 9 présente une technique sans transformation permettant d'unifier le *model-checking* et le *monitoring* dans le cadre d'UML en s'appuyant sur des machines à états UML pour exprimer les propriétés.

La Partie III vise à évaluer l'approche EMI et son applicabilité dans le monde industriel.

- Le chapitre 10 présente l'interpréteur de modèles UML ayant servi comme preuve de concept à l'approche EMI et les différentes expérimentations réalisées pour évaluer cette approche à travers différents cas d'études.

Ce manuscrit possède également différentes annexes :

- L'annexe A présente les fondements de l'IDM, sur lesquels cette thèse s'appuie, et le langage UML utilisé pour réaliser une preuve de concept de cette approche. Ce chapitre

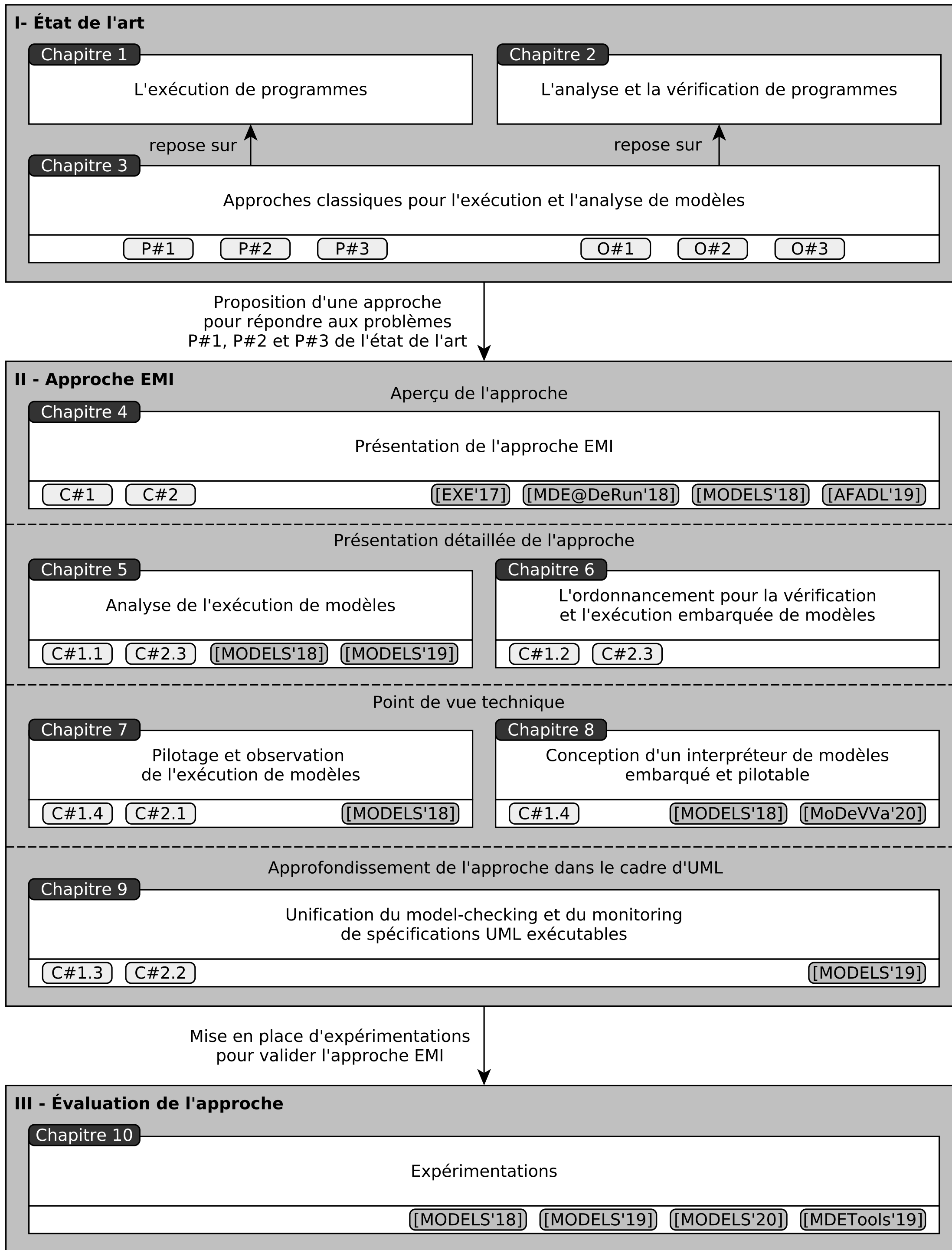



FIGURE 2 – Plan de lecture de ce manuscrit.

donne des prérequis pour les lecteurs non avertis.

— L'annexe B présente le code Lean ayant servi à formaliser la définition de nos interfaces et de nos opérateurs.
— L'annexe C décrit les langages d'action et d'observation de l'interpréteur EMI-UML.
— L'annexe D présente les deux cas d'études utilisés pour réaliser les expérimentations décrites dans ce manuscrit.
— L'annexe E donne une description détaillée des propositions atomiques utilisées dans ce manuscrit avec EMI-UML.
— L'annexe F donne les liens des pages web archivées et référencées dans ce document.

## Liste des publications

### Conférences internationales à comité de lecture

**[MODELS'18]** Valentin BESNARD, Matthias BRUN, Frédéric JOUAULT, Ciprian TEODOROV et Philippe DHAUSSY, « Unified LTL Verification and Embedded Execution of UML Models », in : *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, MODELS '18, Copenhagen, Denmark : Association for Computing Machinery, oct. 2018, p. 112-122, ISBN : 9781450349499, DOI : `10.1145/3239372.3239395`

**[MODELS'19]** Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, « Verifying and Monitoring UML Models with Observer Automata : A Transformation-Free Approach », in : *ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS '19)*, MODELS '19, Munich, Germany, sept. 2019, p. 161-171, DOI : `10.1109/MODELS.2019.000-5`

**[MODELS'20]** Frédéric JOUAULT, Valentin BESNARD, Théo LE CALVAR, Ciprian TEODOROV, Matthias BRUN et Jérôme DELATOUR, « Designing, Animating, and Verifying Partial UML Models », in : *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20)*, MODELS '20, Virtual Event, Canada, oct. 2020, ISBN : 978-1-4503-7019-6/20/10, DOI : `10.1145/3365438.3410967`

### Ateliers de conférences internationales à comité de lecture

**[EXE'17]** Valentin BESNARD, Matthias BRUN, Philippe DHAUSSY, Frédéric JOUAULT, David OLIVIER et Ciprian TEODOROV, « Towards one Model Interpreter for Both Design and Deployment », in : *3rd International Workshop on Executable Modeling (EXE 2017)*, Austin, United States, sept. 2017

**[TTC'18]** Valentin BESNARD, Frédéric JOUAULT, Théo LE CALVAR et Massimo TISI, « The TTC 2018 Social Media Case, by ATL and AOF », in : *11th Transformation Tool Contest, co-located with the 2018 Software Technologies : Applications and Foundations (STAF 2018)*, Toulouse, France, juin 2018

**[MDE@DeRun'18]** Valentin BESNARD, Matthias BRUN, Frédéric JOUAULT, Ciprian TEODOROV et Philippe DHAUSSY, « Embedded UML Model Execution to Bridge the Gap Between Design and Runtime », in : *Software Technologies : Applications and Foundations*, sous la dir. de Manuel MAZZARA, Iulian OBER et Gwen SALAÜN, Cham : Springer International Publishing, 2018, p. 519-528, ISBN : 978-3-030-04771-9, DOI : 10.1007/978-3-030-04771-9_38

**[MDE@DeRun'18]** Hugo BRUNELIERE, Romina ERAMO, Abel GÓMEZ, Valentin BESNARD, Jean Michel BRUEL, Martin GOGOLLA, Andreas KÄSTNER et Adrian RUTLE, « Model-Driven Engineering for Design-Runtime Interaction in Complex Systems : Scientific Challenges and Roadmap », in : *Software Technologies : Applications and Foundations*, sous la dir. de Manuel MAZZARA, Iulian OBER et Gwen SALAÜN, Cham : Springer International Publishing, 2018, p. 536-543, ISBN : 978-3-030-04771-9, DOI : 10.1007/978-3-030-04771-9_40

**[MDETools'19]** Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, « A Model Checkable UML Soccer Player », in : *3rd Workshop on Model-Driven Engineering Tools*, Munich, Germany, sept. 2019, p. 211-220

**[MoDeVVa'20]** Valentin BESNARD, Frédéric JOUAULT, Matthias BRUN, Ciprian TEODOROV et Philippe DHAUSSY, « Modular Deployment of UML Models for V&V Activities and Embedded Execution », in : *Proceedings of the 17th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, Virtual Event, Canada, oct. 2020, ISBN : 978-1-4503-8135-2/20/10, DOI : 10.1145/3417990.3419227

### Conférences nationales à comité de lecture

**[AFADL'19]** Valentin BESNARD, « Unification de la Vérification et de l'Exécution Embarquée de Modèles », in : *Actes des 18èmes journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, AFADL '19, Toulouse, France, juin 2019, p. 93-100

**[AFADL'19]** Valentin BESNARD, Matthias BRUN, Philippe DHAUSSY, Frédéric JOUAULT et Ciprian TEODOROV, « EMI : Un Interpréteur de Modèles Embarqué pour l'Exécution et la Vérification de Modèles UML », in : *Actes des 18èmes journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, AFADL '19, Toulouse, France, juin 2019, p. 101-104

### Posters à comité de lecture

**[MODELS'18]** Valentin BESNARD, Matthias BRUN, Frédéric JOUAULT, Ciprian TEODOROV et Philippe DHAUSSY, *One Model Interpreter for Simulation, Verification, and Execution of UML Models*, Posters of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '18), oct. 2018

**[MODELS'19]** Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, *Verifying and Monitoring UML Models with Observer Automata*, Posters of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS '19), sept. 2019

Première partie

# État de l'art

CHAPITRE 1

# L'EXÉCUTION DE PROGRAMMES

## Sommaire


**1.1 Introduction** . . . . . . . . . . 15

**1.2 Définition d'un langage** . . . . . . . . . . 16

1.2.1 Définition . . . . . . . . . . 16

1.2.2 Sémantique structurelle vs comportementale . . . . . . . . . . 18

1.2.3 Les différents styles de définition de la sémantique comportementale . 18

**1.3 Approches pour l'exécution de programmes** . . . . . . . . . . 22

1.3.1 Définition des relations "conforme à" et "implémente" . . . . . . . . . . 22

1.3.2 Approches basées sur une sémantique translationnelle . . . . . . . . . . 23

1.3.3 Approches basées sur une sémantique opérationnelle . . . . . . . . . . 26

1.3.4 Synthèse . . . . . . . . . . 29


## 1.1 Introduction

Pour concevoir des systèmes numériques, les ingénieurs doivent implémenter des programmes décrivant le comportement de ces systèmes et leurs interactions avec l'environnement. Pour définir ces programmes, des langages précisant la syntaxe (la structure) et la sémantique (le sens) des différents concepts de programmation doivent être utilisés. La syntaxe fournit une notation permettant l'écriture des programmes alors que la sémantique associe une signification aux éléments les constituants. Pour permettre l'exécution de ces programmes, la sémantique doit être capturée dans un moteur d'exécution. Elle peut être définie en utilisant différents styles et outils. Les deux approches les plus utilisées sont les sémantiques translationnelles et opérationnelles typiquement implémentées par des générateurs de code et des interpréteurs.

La section 1.2 de ce chapitre rappelle la définition d'un langage et focalise sur la notion de sémantique d'un langage. Les différentes approches et outils permettant l'exécution de programmes sont ensuite présentés dans la section 1.3.

## 1.2 Définition d'un langage

Cette section, inspirée de [Com08 ; Mos06 ; SK95 ; Kle07], présente les différentes composantes d'un langage et leurs relations. Elle fait ensuite une distinction entre sémantique structurelle et comportementale avant de présenter une classification des différents styles de définition de la sémantique comportementale d'un langage.

### 1.2.1 Définition

En informatique comme en linguistique, un langage est défini par sa syntaxe et sa sémantique. La syntaxe caractérise l'ensemble des concepts du langage et leurs représentations concrètes. La sémantique permet quant à elle d'associer une signification à chaque élément syntaxique du langage. Ces deux notions permettent de définir la forme (la structure et la notation) et le fond (le sens) des programmes [Com08].

Pour les langages de programmation, on dissocie la syntaxe abstraite de la syntaxe concrète. La *syntaxe abstraite* définit l'ensemble des concepts (ou constructions) du langage et leurs relations. Elle ne s'intéresse donc qu'à la structure interne des programmes. Elle peut par exemple être définie à l'aide d'une grammaire et ainsi se représenter sous la forme d'un arbre syntaxique (ou en anglais *Abstract Syntax Tree* (AST)). Les nœuds de l'arbre correspondent alors aux constructions du langage et les branches aux arguments de ces constructions. La *syntaxe concrète* permet quant à elle d'associer une représentation à chacun des concepts du langage. C'est cette représentation concrète qui sera utilisée par les ingénieurs pour définir des programmes dans ce langage. Elle peut prendre une forme textuelle (p. ex. avec Xtext [EB10], TCS [JBK06]) ou graphique (p. ex. avec Sirius [1]). Dans le cas d'une représentation textuelle, la syntaxe concrète est typiquement définie à l'aide d'une grammaire pouvant ensuite être utilisée par un parseur pour construire l'arbre syntaxique d'un programme.

Un langage est aussi caractérisé par son *domaine sémantique* qui représente l'ensemble des états atteignables d'un programme [Com08] (c.-à-d. l'ensemble des combinaisons de valeurs pouvant être prises par les attributs dynamiques d'un programme). La sémantique d'un langage définit comment passer d'un état à un autre (c.-à-d. l'évolution des valeurs des attributs dynamiques d'un programme).

Un langage informatique repose donc sur la combinaison de trois éléments : une syntaxe abstraite, une syntaxe concrète et un domaine sémantique. La définition d'un langage est centrée soit sur sa syntaxe concrète soit sur sa syntaxe abstraite. Des liens ($M_{ca}$, $M_{cs}$) ou ($M_{ac}$, $M_{as}$) permettent d'associer les concepts de la syntaxe concrète ou de la syntaxe abstraite avec les concepts des autres éléments. Ces deux possibilités sont illustrées sur la Figure 1.1.

1. Sirius : `http://www.eclipse.org/sirius/`.

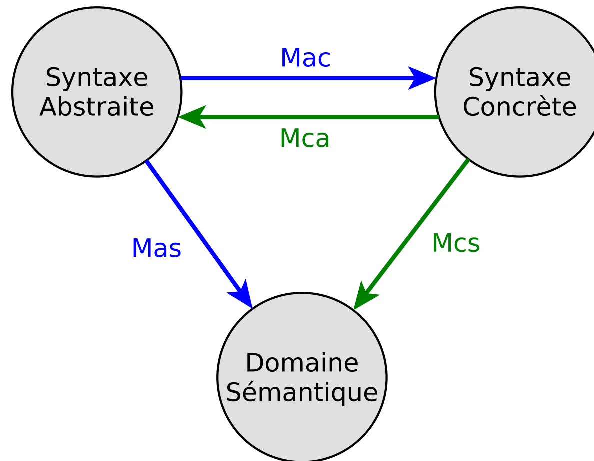



FIGURE 1.1 – Définition d'un langage basé soit sur sa syntaxe concrète (en vert) soit sur sa syntaxe abstraite (en bleu). (Figure adaptée de [Com08])

Pour des raisons historiques, les langages dont la définition est axée sur la syntaxe concrète sont principalement des langages de programmation. Un lien $M_{ca}$ dit de dérivation ou d'abstraction permet d'associer les éléments de la syntaxe concrète à ceux de la syntaxe abstraite. Ce lien est souvent obtenu via l'utilisation d'un parseur. Celui-ci permet d'enlever tout le *sucre* syntaxique (p. ex. les accolades, les points-virgules) servant uniquement à la mise en forme du programme. Un lien $M_{cs}$ permet de lier les concepts de la syntaxe concrète avec des états (ou des valeurs) du domaine sémantique du langage.

**Définition 1.1 (Langage centré sur la syntaxe concrète).** *"Un langage informatique est défini selon le tuple $\{AS, CS, M_{ca}, SD, M_{cs}\}$ où $AS$ est la syntaxe abstraite, $CS$ est la syntaxe concrète, $M_{ca}$ est le mapping de la syntaxe concrète vers sa représentation abstraite, $SD$ est le domaine sémantique et $M_{cs}$ est le mapping de la syntaxe concrète vers le domaine sémantique."* [Com08]

La définition d'un langage peut également être centrée sur sa syntaxe abstraite. L'utilisation de cette définition est privilégiée par les langages de modélisation. Elle permet notamment l'usage de multiples syntaxes concrètes offrant ainsi la possibilité d'utiliser différents formalismes pour différents besoins. Pour la modélisation, certains utilisateurs préfèrent un formalisme graphique alors que d'autres sont plus à l'aise avec un formalisme textuel. Pour le stockage des programmes, il peut également être intéressant d'utiliser un formalisme dédié (p. ex. *XML Metadata Interchange* (XMI®) [OMG15]) pour favoriser l'interopérabilité entre les outils. Pour chaque syntaxe concrète, la définition du langage doit donc fournir un lien $M_{ac}$ entre les concepts de la syntaxe abstraite et les décorations syntaxiques (graphiques ou textuelles) utilisées dans la syntaxe concrète. Un seul lien $M_{as}$ est cependant nécessaire pour lier les

concepts de la syntaxe abstraite avec des états (ou des valeurs) du domaine sémantique du langage.

**Définition 1.2 (Langage centré sur la syntaxe abstraite).** *"Un langage informatique est défini selon le tuple $\{AS, CS^*, M^*_{ac}, SD, M_{as}\}$ où $AS$ est la syntaxe abstraite, $CS^*$ est la (les) syntaxe(s) concrète(s), $M^*_{ac}$ est l'ensemble des mappings de la syntaxe abstraite vers la (les) syntaxe(s) concrète(s), $SD$ est le domaine sémantique et $M_{as}$ est le mapping de la syntaxe abstraite vers le domaine sémantique."* [Com08]

Dans cette thèse, nos travaux se focalisent sur l'exécution et l'analyse dynamique de programmes. Après avoir défini la notion de langage, notre intérêt se porte donc sur la définition de la sémantique des langages.

### 1.2.2 Sémantique structurelle vs comportementale

Dans la sémantique d'un langage, on distingue deux parties : celle qui se rapporte à la structure du programme et celle liée à son comportement.

La *sémantique structurelle* correspond à un ensemble de contraintes permettant de vérifier si un programme est valide (c.-à-d. conforme au langage) [Com08 ; SK95]. Ces contraintes sont directement exprimées sur la syntaxe abstraite du langage et peuvent être vérifiées de façon statique (c.-à-d. sans exécution du programme). Elles permettent d'ajouter certaines restrictions qui ne peuvent être décrites directement dans la syntaxe abstraite (p. ex. pour contraindre la multiplicité d'un élément ou exprimer des relations d'ordre). Cette sémantique est aussi parfois qualifiée de "sémantique statique" mais ce terme est également utilisé pour désigner des contraintes davantage liées à la syntaxe du langage (p. ex. la résolution de noms, le typage des programmes) [OMG17d ; SK95 ; Mos06]. Pour éviter toute ambiguïté, la dénomination sémantique structurelle est donc privilégiée.

La *sémantique comportementale* permet, quant à elle, de modéliser le comportement dynamique des programmes [Com08 ; SK95]. À ce titre, elle est aussi qualifiée de "sémantique dynamique" d'un programme. Elle permet de définir le comportement observable de l'exécution d'un programme ainsi que son évolution au cours du temps. Cela inclut par exemple les effets de bords sur la mémoire ou encore les interactions avec l'environnement. La sémantique comportementale est celle qui est capturée pour exécuter des programmes.

### 1.2.3 Les différents styles de définition de la sémantique comportementale

Dans la littérature, différents styles peuvent être utilisés pour définir la sémantique d'un langage [Kle07 ; SK95 ; Mos06 ; Com08]. Les approches opérationnelle, dénotationnelle, translationnelle et axiomatique permettent de définir la sémantique comportementale d'un langage.

La sémantique axiomatique peut également être utilisée pour spécifier la sémantique structurelle sous forme de contraintes. D'autres approches existent [SK95 ; Mos06] mais elles peuvent souvent être vues comme des variantes ou des combinaisons des quatre approches mentionnées précédemment. Pour illustrer chacune d'entre elles, nous allons décrire la sémantique de la fonction $add$ qui permet d'obtenir le résultat ($res$) de l'addition de deux nombres ($op1$ et $op2$).

#### 1.2.3.1 Sémantique opérationnelle

La sémantique opérationnelle permet de décrire l'exécution d'un programme sous la forme d'une séquence de pas d'exécution [Kle07]. Elle est souvent représentée sous la forme d'un système de transitions où chaque nœud est appelé un état d'exécution ou une configuration. Un nœud regroupe les valeurs à un instant donné de l'ensemble des attributs dynamiques qui évoluent au cours de l'exécution. L'ensemble des nœuds modélisent donc tous les états possibles de l'exécution du programme, c.-à-d. son espace d'état. Les transitions correspondent aux pas d'exécution permettant de passer d'une configuration à une autre en appliquant un ensemble d'opérations sur l'état d'exécution courant. Cette sémantique est aussi qualifiée d'*intentionnelle* [GDT14] car elle se focalise sur l'intention c.-à-d. la séquence des pas d'exécution permettant d'obtenir le résultat. La sémantique opérationnelle peut donc être vue comme une suite de transformations successives modélisant la façon dont l'exécution du programme progresse d'un état d'exécution à un autre.

Pour la fonction $add$, la Figure 1.2 représente le système de transitions qui capture sa sémantique opérationnelle. Les valeurs $a$, $b$ et $c \in \mathbb{Z}$ représentent respectivement les valeurs initiales de $op1$, $op2$, et $res$ pour un programme donné. L'application de la fonction $add$ est modélisée par la transition de la configuration $i$ à la configuration $j$.

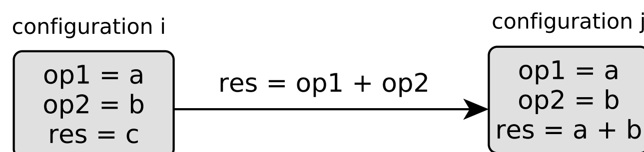



FIGURE 1.2 – Sémantique opérationnelle de la fonction $add$.

Les sémantiques opérationnelles sont habituellement classifiées en deux catégories [SK95]. Les sémantiques à petits pas (ou en anglais *Structural Operational Semantics* (SOS) [Plo04]) utilisent des pas atomiques pour décrire l'exécution d'un programme. Dans ce cas, les pas d'exécution ne peuvent pas être découpés plus finement. Les sémantiques à grands pas (ou sémantiques naturelles [Kah87]) décrivent globalement comment les résultats de l'exécution d'un programme sont obtenus. Chaque pas est donc composé d'un ensemble de pas d'exécution atomiques mais seul le résultat global de l'exécution de tous ces pas est observable.

#### 1.2.3.2 Sémantique dénotationnelle

La sémantique dénotationnelle associe à chaque élément $e$ d'un programme sa dénotation $\mathcal{A}[\![e]\!]_\sigma$ dans le domaine sémantique choisi c.-à-d. un objet mathématique qui représente le comportement désigné par $e$ [Kle07 ; SK95]. Une dénotation est typiquement construite à l'aide d'une fonction prenant en argument des données d'entrées et fournissant en sortie des données représentant le résultat de cette fonction. La sémantique dénotationnelle ne s'intéresse pas aux différents pas d'exécution nécessaires pour calculer les sorties mais seulement aux liens entre les entrées et les sorties c.-à-d. à l'effet des programmes. Elle est aussi qualifiée de sémantique *extensionnelle* [GDT14] car seules les relations visibles entre les entrées et les sorties (c.-à-d. l'extension) comptent. En ce sens, la sémantique dénotationnelle correspond à une abstraction de la sémantique opérationnelle.

La particularité d'une sémantique dénotationnelle est d'être composable [SK95] c.-à-d. que la dénotation d'un élément peut être construite à partir des dénotations de ces sous-éléments. La dénotation d'un élément dépend aussi de l'environnement d'exécution (c.-à-d. de l'état de la mémoire) tel qu'il existe une fonction $\sigma$ qui associe à chaque variable $x$ sa valeur telle que $\sigma : x \rightarrow \mathbb{Z}$. La sémantique dénotationnelle de la fonction $add$ peut ainsi être exprimée de la façon suivante :

$$\mathcal{A}[\![add(op1, op2)]\!]_\sigma = \mathcal{A}[\![op1]\!]_\sigma + \mathcal{A}[\![op2]\!]_\sigma = \sigma(op1) + \sigma(op2).$$

#### 1.2.3.3 Sémantique translationnelle

La sémantique translationnelle, aussi appelée sémantique par traduction, permet d'exprimer la sémantique d'un langage en traduisant des programmes de ce langage source vers un langage cible dont la sémantique est rigoureusement définie [Kle07]. Autrement dit, les concepts des programmes du langage source sont exprimés en termes des éléments de la syntaxe (abstraite ou concrète) du langage cible. La sémantique du langage cible étant énoncée précisément, il est ainsi possible de connaître la signification des éléments du programme source.

Le Listing 1.1 montre le résultat de la traduction en assembleur Thumb de la fonction $add$. L'outil permettant de faire la traduction capture ainsi la sémantique de cette fonction sous forme translationnelle.

```
.thumb              // Type du jeu d'instructions
.text               // Section de code
.global add         // Rend la visibilité de la fonction add publique
add:
```

```
ADD r0, r0, r1  // Addition de op1 et op2 (dans r0 et r1): r0 = r0 + r1
MOV pc, lr      // Retour à l'appelant avec le résultat dans r0
```

Listing 1.1 – Fonction $add$ en assembleur Thumb obtenue grâce à une sémantique translationnelle.

#### 1.2.3.4 Sémantique axiomatique

La sémantique axiomatique permet de définir la sémantique d'un langage à l'aide de propositions logiques (les axiomes) qui décrivent l'effet d'un programme sous forme d'assertions sur l'état du programme [Kle07]. Les axiomes peuvent être spécifiés sous forme d'invariants pour indiquer que toutes les exécutions d'un programme doivent satisfaire certaines conditions. Ils peuvent aussi être exprimés sous forme de préconditions et de postconditions sur les opérations d'un programme. L'algorithme d'une opération doit ainsi s'assurer de satisfaire les postconditions à partir des hypothèses posées par les préconditions. La sémantique axiomatique peut être vue comme une abstraction de la sémantique dénotationnelle car elle définit une relation entre les entrées et les sorties sans modéliser l'état du programme.

La sémantique axiomatique de la fonction $add$ est présentée dans le Listing 1.2 avec le formalisme ACSL de Frama-C [2] [Kir+15]. Le mot-clé *ensures* exprime une post-condition sur le résultat (`\result` en ACSL) à partir des valeurs initiales de $op1$ et $op2$ identifiées avec le mot-clé `\old`.

```
/*@ ensures \result == \old(op1) + \old(op2) */
int add (int op1, int op2);
```

Listing 1.2 – Sémantique axiomatique de la fonction $add$ avec le formalisme ACSL.

**Conclusion sur la sémantique des langages** La signification des concepts d'un langage est définie par sa sémantique. Celle-ci peut être capturée en utilisant différents styles dont les quatre principaux (c.-à-d. opérationnel, dénotationnel, translationnel et axiomatique) ont été présentés dans cette section. Dans le cadre de cette thèse, nos travaux s'intéressent principalement à la sémantique comportementale des langages afin de définir des moteurs d'exécution permettant d'exécuter des programmes conformes à ces langages. Dans la suite de ce document (sauf mention contraire), l'utilisation du terme "sémantique" fera donc référence à la sémantique comportementale et non à la sémantique structurelle.

2. Frama-C : `https://frama-c.com/`.

## 1.3 Approches pour l'exécution de programmes

Pour exécuter des programmes, des outils doivent permettre de passer d'une représentation textuelle ou graphique à une exécution concrète du système. Pour effectuer cette tâche, les sémantiques translationnelles et opérationnelles sont particulièrement appropriées car elles peuvent être mises en œuvre à l'aide respectivement de générateurs de code et d'interpréteurs. Ces deux approches peuvent être appliquées aussi bien sur les programmes des langages de programmation que sur les modèles des langages de modélisation. Un *modèle* est une abstraction d'un système qui est créée dans une intention particulière afin d'obtenir une représentation de ce système pour le but énoncé (cf. section A.2.1). Dans le cadre de l'IDM, un modèle est également un programme conforme à un langage de modélisation (cf. section A.2.1) utilisé pour définir une représentation abstraite d'un système. Le langage permet ainsi de concevoir des modèles grâce à sa syntaxe et de leur donner du sens grâce à sa sémantique. Dans cette thèse, nous allons utiliser des modèles au sens de l'IDM. Néanmoins, afin de rester indépendant des techniques propres à l'IDM, nous considérerons dans cet état de l'art que tout langage informatique peut être vu comme un langage de modélisation avec un niveau d'abstraction plus ou moins élevé.

Dans cette section, nous allons donc étudier plus en détail le fonctionnement des deux principales approches d'exécution ainsi que les outils permettant de les appliquer aux langages de modélisation. Après avoir défini des relations clés pour l'exécution de modèles en section 1.3.1, la section 1.3.2 s'intéresse aux techniques basées sur une sémantique translationnelle et la section 1.3.3 à celles basées sur une sémantique opérationnelle.

### 1.3.1 Définition des relations “conforme à” et “implémente”

Pour présenter les approches d'exécution, cette thèse s'appuie sur deux relations illustrées en Figure 1.3 qu'il convient de définir plus précisément.

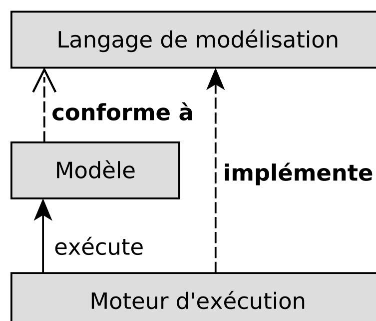



FIGURE 1.3 – Les relations “conforme à” et “implémente” .

La relation "conforme à" indique que le *Modèle* est conforme à la syntaxe abstraite du *Langage de modélisation*. Autrement dit, les concepts utilisés pour définir le *Modèle* doivent être spécifiés par la syntaxe abstraite du langage de modélisation (c.-à-d. sa grammaire). La définition de cette relation est raffinée dans la section A.2.1.1 de l'annexe A pour l'ingénierie des modèles.

La relation "implémente" indique que le *Moteur d'exécution* utilisé pour exécuter le *Modèle* implémente la sémantique du *Langage de modélisation*. Autrement dit, les informations permettant de donner du sens aux concepts du *Modèle* doivent être capturées par le *Moteur d'exécution* par exemple sous forme de fonctions ou de règles de transformation. Cette relation reste également vraie lorsque le *Moteur d'exécution* est une plateforme d'exécution embarquée (p. ex. un microprocesseur). Dans ce cas, les composants hardware formant les registres, l'unité arithmétique et logique, ou encore les bus de communication permettent de décoder le sens associé à chaque instruction. Ils permettent ainsi de créer des effets de bords sur la mémoire, d'afficher des informations sur la sortie standard ou encore de piloter des actionneurs. En ce sens, une plateforme d'exécution matérielle encode donc bien la sémantique du langage d'exécution utilisé pour écrire les instructions du programme exécuté.

Par souci de simplification des figures de ce manuscrit, ces deux relations ("conforme à" et "implémente") qui font référence soit à la syntaxe abstraite ou soit à la sémantique du langage de modélisation seront représentées par des flèches pointant directement sur le *Langage de modélisation*.

### 1.3.2 Approches basées sur une sémantique translationnelle

Une approche basée sur une sémantique translationnelle permet de capturer la sémantique d'un langage de modélisation, souvent définie autre part (p. ex. sous forme textuelle), par traduction vers du code exécutable. Des générateurs de code ou des compilateurs sont généralement utilisés pour transformer les éléments du modèle de conception en termes des concepts d'un langage cible. Le langage cible est en général un langage de programmation exécutable pour lequel il existe une sémantique précise et des outils de développement associés.

#### 1.3.2.1 Description

Le principe de cette approche est présenté sur la Figure 1.4. Un *Modèle* décrit le système à concevoir dans un *Langage de modélisation* noté LM. Une transformation appelée *génération de code* permet de traduire ce modèle de conception en *Code* exécutable. L'outil qui capture la sémantique de LM et qui réalise la transformation est un traducteur appelé *Générateur de code*. Le *Code* généré est conforme au *Langage d'exécution* noté LE et peut être exécuté sur une *Plateforme d'exécution* implémentant la sémantique de LE. Cette technique permet

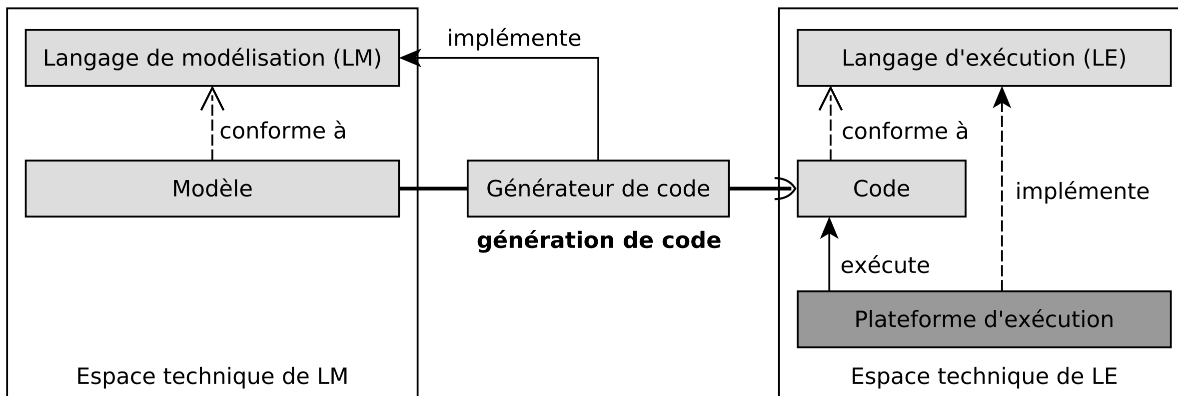



FIGURE 1.4 – Approche d'exécution basée sur une sémantique translationnelle.

ainsi d'exécuter un *Modèle* conforme à LM dans l'*Espace technique de LE*. Informellement, on appelle espace technique [Jou06] l'ensemble des outils de développement permettant de manipuler des programmes ou des modèles conformes à un langage. Le langage LE pouvant être de diverses natures (p. ex. *bytecode*, assembleur, *General Purpose Language* (GPL)), les plateformes d'exécution sont également très variées (p. ex. machines virtuelles (ou en anglais *Virtual Machines* (VMs)), systèmes d'exploitation (ou en anglais *Operating Systems* (OS)), microcontrôleurs).

Cette approche permet donc via une transformation de changer d'espace technique afin de bénéficier des plateformes d'exécution définies pour le langage cible. La sémantique du langage de modélisation est ainsi capturée dans un outil de transformation (modèle vers texte) appelé générateur de code (ou compilateur).

#### 1.3.2.2 Outils

Dans la communauté du génie logiciel, de nombreux outils ont été développés pour capturer la sémantique d'un langage de façon translationnelle. En pratique, ces outils ne traduisent pas les modèles directement vers du code machine mais vers un langage intermédiaire ayant un niveau d'abstraction plus ou moins élevé. Plusieurs étapes de traduction sont donc nécessaires avant d'obtenir du code machine qui soit exécutable par une plateforme d'exécution matérielle.

**Différents niveaux d'abstraction.** Pour classifier les générateurs de code, cette thèse propose de les regrouper en trois grands groupes dépendants essentiellement du niveau d'abstraction du langage cible.

*Vers un GPL.* Une première méthode est de définir un générateur de code vers un langage de programmation générique (GPL) (p. ex. C, C++, Java, Objective-C, Ada). De nombreux outils

ont notamment été définis pour UML dont Papyrus Software Designer (Papyrus SD) [Pha+17], Rhapsody [HK04], Rational Software Architect (RSA) [LNH06], UML4CPP [Jäg+16] ou le simulateur UML [KDH07]. Concernant les systèmes temps réel, les outils Fujaba [Bur+05], Rational Software Architect Real Time Edition (RSARTE) [Moh15] et Papyrus-RT [HDB17] permettent de générer du code prenant en compte des contraintes temps réel.

*Vers une représentation intermédiaire dédiée.* Une autre approche propose de transformer le modèle de conception vers un langage intermédiaire plus proche des concepts d'implémentation. Les travaux [PM03a ; PM03b ; SM05b] reposent sur cette approche pour exécuter des modèles UML en passant par les formalismes intermédiaires *Extended Hierarchical Automata* (EHA) [PM03a ; PM03b] ou *Executable State Machine* (ESM) [SM05b]. L'intérêt de ces formalismes est notamment de mettre à plat les machines à états UML pour simplifier la génération de code [PM03a ; PM03b] ou rendre l'exécution plus efficace [SM05b].

*Vers un langage assembleur.* D'autres générateurs de code, souvent qualifiés de *compilateurs*, ont été définis vers des langages bas niveau (c.-à-d. très proche du langage machine). Des compilateurs existent notamment pour les langages de programmation comme C (p. ex. GCC) ou Rust (p. ex. rustc). Pour UML, les compilateurs de modèles GUML [CMB12] et Unicomp [Cic18] permettent de transformer des modèles UML vers du code binaire en passant par une représentation intermédiaire de GCC [3] ou LLVM [4].

Tous ces outils permettent d'exécuter des modèles et sont généralement adaptés pour l'exécution embarquée. En ce qui concerne plus spécifiquement UML, la section A.3.3 donne un aperçu des outils implémentant une sémantique translationnelle pour UML. Parmi tous ces outils, très peu utilisent une approche formelle. En particulier, aucun des outils cités précédemment ne permet de garantir que le code exécutable obtenu est conforme au modèle de conception initial. Pour le langage UML, il n'existe pas, à notre connaissance, d'outils permettant de remplir cet objectif. Dans la littérature, plusieurs travaux [LP99 ; Fra+98 ; Liu+13a] ont proposé une définition formelle de la sémantique d'UML en se basant sur une notation mathématique. Néanmoins, ces définitions formelles servent uniquement à appliquer des activités d'analyse sur des modèles UML mais pas à exécuter ces modèles sur des plateformes d'exécution réelles.

**Garantir la sémantique d'exécution.** Plusieurs outils permettent de garantir que la sémantique d'exécution est conforme à la sémantique du langage de modélisation. Il est par exemple intéressant de noter l'initiative du projet CompCert [5] [Ler09] qui a permis de construire le premier compilateur de code C certifié en utilisant Coq [6] [BC10]. Pour les systèmes embarqués,

3. GCC : `https://gcc.gnu.org/`.
4. LLVM : `https://llvm.org/`.
5. CompCert : `http://compcert.inria.fr/`.
6. Coq : `https://coq.inria.fr/`.

l'environnement de développement SCADE [Ber07] possède quant à lui un générateur de code certifié, appelé *KCG compiler*, pour rendre exécutable les modèles SCADE. En couplant CompCert avec le générateur de code KCG, il serait possible de garantir que la sémantique d'exécution des modèles SCADE est conforme à sa spécification. Dans le cadre de l'ingénierie des modèles, il est également intéressant de mentionner l'approche en [Com+09]. Elle permet de définir une sémantique translationnelle pour un langage de modélisation donné et de valider celle-ci par rapport à une sémantique opérationnelle de référence grâce aux techniques de bisimulation.

### 1.3.3 Approches basées sur une sémantique opérationnelle

Pour exécuter des modèles, une autre approche consiste à capturer la sémantique opérationnelle du langage de modélisation dans un moteur d'exécution appelé interpréteur. Cet outil encode la sémantique du langage sous forme de règles d'exécution qui sont appliquées par transformations successives sur l'état d'exécution du modèle.

#### 1.3.3.1 Description

Le principe de cette approche est présenté sur la Figure 1.5. Contrairement à l'approche translationnelle, c'est directement le *Modèle* de conception qui est exécuté. Ce *Modèle* se conforme au *Langage de modélisation* et est directement embarqué sur la *Plateforme d'exécution*. Un *Interpréteur* doit également être conçu pour implémenter la sémantique de LM. Pour chaque concept exécutable de LM, une règle (c.-à-d. un algorithme) est définie pour modifier l'état d'exécution du modèle (c.-à-d. les données d'exécution dynamiques) en conformité avec

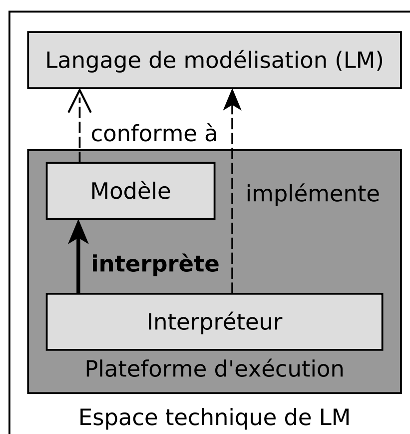



FIGURE 1.5 – Approche d'exécution basée sur une sémantique opérationnelle.

la sémantique du langage. Cet outil permet ainsi d'exécuter directement le modèle de conception sur la *Plateforme d'exécution*. Suivant le langage interprété, la plateforme d'exécution peut ici aussi être de différentes natures (p. ex. une autre VM, un OS, un microcontrôleur).

Cette approche n'utilise pas de transformation et permet d'implémenter un moteur d'exécution directement dans l'*Espace technique de LM*. Plusieurs patrons de conception (ou en anglais *design patterns*) ont été élaborés afin de faciliter l'implémentation d'une sémantique opérationnelle. On trouve notamment les patrons Interpréteur et Visiteur définis par le *Gang of Four* dans [EHV95]. Plus récemment, d'autres patrons [Led+17 ; LDC18 ; WO16 ; OC12 ; JBF11] ont également été développés afin de faciliter la réutilisation et l'extension de la syntaxe et de la sémantique des langages.

#### 1.3.3.2 Outils

Différents interpréteurs permettent d'exécuter des modèles avec une sémantique opérationnelle. Cette section présente les caractéristiques de ces outils basés sur des approches spécifiques (applicables à un seul langage) ou des méta-approches (pouvant être mises en œuvre sur différents langages).

**Approches spécifiques.** Une première méthode pour implémenter une sémantique opérationnelle est de concevoir un interpréteur uniquement pour un langage donné (c.-à-d. de façon spécifique). Par conséquent, cet interpréteur n'est applicable qu'à des modèles de ce langage. L'un des langages sur lequel l'implémentation d'une sémantique opérationnelle a été le plus appliquée est UML. Le travail en [Rie+01] définit notamment une architecture permettant d'implémenter une machine virtuelle pour UML (UML VM). Les outils ACTi [CD08] et Pópulo [FMS08] permettent d'interpréter les actions et les activités d'UML 2.0 tandis que UVM [SM05a] base son exécution sur les machines à états et les diagrammes de séquence. Le standard *foundational UML* (fUML™) [OMG17c], un sous-ensemble d'UML, peut également être interprété par différents outils dont l'implémentation de référence [7], Moka [Rev+18] et Moliz [ML12].

**Méta-approches — L'initiative Gemoc.** Gemoc Studio [8] est un Environnement de Développement Intégré (EDI) permettant à la fois de définir des langages et de concevoir des modèles conformes à ces langages. La sémantique d'un langage est développée à l'aide d'un méta-langage pouvant lui-même être exécuté à l'aide d'un moteur d'exécution appelé méta-outil. Ce moteur d'exécution permet de concevoir un interpréteur pour le langage conçu et de faire le lien avec les autres outils de développement (p. ex. débogueur, animateur). Chaque métalangage et son méta-outil définissent une méta-approche supportée par Gemoc Studio. À partir

7. fUML Ref. Impl. : `http://modeldriven.github.io/fUML-Reference-Implementation/`.
8. GEMOC Studio : `http://gemoc.org/studio.html`.

de la définition d'un langage, Gemoc Studio offre un environnement de développement complet permettant la modélisation, l'exécution, et l'analyse de modèles conformes à ce langage. L'initiative Gemoc propose ainsi une démarche systématique pour définir la sémantique des langages. Les cinq métalangages supportés dans Gemoc Studio sont Java + Kermeta 3, ALE, xMOF, MoCCML + Kermeta 3 et Henshin.

Nous allons maintenant décrire chacune des cinq méta-approches associées en commençant par celles qui permettent de définir des langages séquentiels (c.-à-d. des langages définissant un unique chemin d'exécution correspondant à une séquence d'instructions). *(1)* La méta-approche Java + Kermeta 3 [Bou+16] permet de définir la sémantique opérationnelle à l'aide de méthodes Java qui sont ensuite tissées dans la syntaxe abstraite du langage avec Kermeta 3 (K3) [JBF11 ; Dre+09], un tisseur d'aspects. *(2)* Le langage *Action Language for Ecore* (ALE) [9] est le successeur de K3. Il permet de définir la sémantique opérationnelle d'un langage via le mécanisme d'*open class* [Cli+00]. *(3)* Le métalanguage xMOF [May+13 ; MLW12] permet, quant à lui, de définir la sémantique opérationnelle d'un langage sous forme d'activités fUML. Gemoc Studio possède également deux méta-approches permettant de définir des langages concurrents (c.-à-d. des langages définissant différents processus qui rentrent en compétition pour leurs exécutions). *(4)* Le métalangage MoCCML + Kermeta 3 [Bou+16] est le premier métalangage conçu par l'équipe Gemoc pour définir des langages concurrents. Le langage *Model of Concurrency and Communication Modeling Language* (MoCCML) [Com+13 ; Dea+15 ; Lat+15] permet de spécifier de façon explicite les différents aspects liés à la concurrence d'un langage. Le modèle de concurrence permet de définir des contraintes sur le flot de contrôle du langage et donc de caractériser l'ensemble des chemins d'exécution possibles. Ce modèle de concurrence est ensuite compilé vers *Clock Constraint Specification Language* (CCSL) [OMG ; And09]. Pour l'exécution, le moteur d'exécution doit coordonner l'interpréteur de la sémantique opérationnelle du langage et TimeSquare [DM12], le solveur de contraintes CCSL utilisé dans Gemoc Studio. *(5)* La méta-approche basée sur Henshin [Zsc18] permet de définir la sémantique opérationnelle des langages concurrents sous forme de règles déclaratives écrites avec le langage Henshin. Pour permettre l'exécution, ces règles reposent sur les techniques de transformation de graphes décrites dans le paragraphe suivant.

**Méta-approches — La réécriture de graphes ou de termes.** Un autre paradigme pour définir une sémantique opérationnelle est d'utiliser des techniques de réécriture. Pour la réécriture de graphes, l'état d'exécution du modèle est vu comme un graphe qu'il est possible de modifier à l'aide de règles de réécriture ou de transformations pour permettre l'exécution. L'outil Henshin [Are+10] permet de définir la sémantique opérationnelle d'un langage sous forme de règles déclaratives. Son moteur d'exécution permet ensuite d'exécuter ces règles par transformation

9. Action Language for EMF : `http://gemoc.org/ale-lang/`.

de modèles. L'interpréteur TGG [KRW04] utilise, quant à lui, des techniques de transformations de graphes basées sur les *Triple Graph Grammars* (TGGs). Des outils de réécriture de termes, comme K framework [RŞ10], peuvent aussi être utilisés pour définir la sémantique opérationnelle d'un langage. K framework a notamment été utilisé pour définir la sémantique de plusieurs langages de programmation [10] dont C [HER15], Java [BR15], Python et Haskell. En théorie, cet outil est aussi applicable pour définir la sémantique des langages de modélisation.

En résumé, on peut constater que très peu d'outils d'interprétation de modèles sont, à notre connaissance, adaptés pour le domaine de l'embarqué c.-à-d. déployables sur une cible embarquée et conçus pour s'exécuter avec peu de ressources. Les travaux les plus proches s'y rapportant sont ceux qui concernent le développement de VMs pour l'embarqué (p. ex. Squawk [SC05], HaLVM [11]), pour les réseaux de capteurs (p. ex. Mote Runner [Car+09], Maté [LC02]), ou pour le temps réel (p. ex. Ovm [Bak+06], IBM VM [Aue+07]). Cependant, ces VMs ne permettent pas de définir des méta-approches et elles opèrent à un niveau d'abstraction inférieur (c.-à-d. au niveau du bytecode). Nous pouvons aussi remarquer que les approches formelles sont rarement utilisées pour définir des interpréteurs. K framework est la seule approche formelle parmi toutes celles citées. En ce qui concerne plus spécifiquement UML, la section A.3.3 donne un aperçu des outils implémentant une sémantique opérationnelle pour UML.

### 1.3.4 Synthèse

Pour positionner nos travaux par rapport aux techniques existantes, cette section synthétise les caractéristiques et les limitations des différentes approches d'exécution de modèles (ou de programmes).

Dans ce chapitre, nous avons identifié deux types d'approches permettant d'implémenter la sémantique comportementale d'un langage dans un moteur d'exécution : *(1)* la génération de code qui permet de capturer une sémantique par traduction vers un autre langage et *(2)* l'interprétation qui permet de définir une sémantique opérationnelle. Pour chacune d'entre elles, nous avons identifié différentes techniques dont les caractéristiques sont synthétisées dans le Tableau 1.1. Les critères de comparaison sont (LM == LE) pour indiquer si les formalismes de modélisation et d'exécution sont identiques et (Réutilisabilité des outils) pour indiquer si l'outil est applicable à un seul ou un ensemble de langages.

D'une manière générale, chacune de ces techniques offre des avantages et des inconvénients. Le choix de l'une ou l'autre dépend donc du contexte d'utilisation et des besoins du projet en question. Néanmoins, l'interprétation est souvent préconisée pour appliquer des outils d'analyse alors que la génération de code offre de meilleures performances pour l'exécution réelle [Dév+15].

10. K framework : `https://github.com/kframework`
11. HaLVM : `https://github.com/GaloisInc/HaLVM`.

| Types d’approche | | LM == LE | Réutilisabilité des outils |
|---|---|---|---|
| Génération de code | | non | non [12] |
| Interprétation | spécifique | oui | non |
| | méta-approche | oui | oui |

TABLE 1.1 – Synthèse des différentes approches pour l’exécution de modèles.

Le Tableau 1.1 montre que, pour l’interprétation, les formalismes de modélisation et d’exécution sont identiques alors que ce n’est pas le cas pour la génération de code. En ce qui concerne la réutilisation des outils, seule l’interprétation permet d’élaborer des méta-approches pouvant être utilisées pour définir différents langages et exécuter des modèles conformes à ces langages. À notre connaissance, aucun générateur de code ne permet de capturer la sémantique de plusieurs langages. Au mieux, ces générateurs de code (ou compilateurs) utilisent un langage pivot en entrée ce qui permet de traduire des modèles de différents langages vers ce langage pivot. Cependant, pour chaque langage source, cette opération nécessite de construire un outil capturant la sémantique translationnelle du langage source vers ce langage pivot. Par conséquent, cette solution ne peut être qualifiée de méta-approche.

12. On peut noter que même si les outils de génération de code ne sont pas réutilisables, les outils de développement de l’espace technique de LE le sont quant à eux. L’utilisation de transformations et donc de fossés sémantiques peut donc être vue comme un avantage en ce sens.

CHAPITRE 2

# L'ANALYSE ET LA VÉRIFICATION DE PROGRAMMES

---

**Sommaire**

**2.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **31**
**2.2 Panorama des techniques de V&V** . . . . . . . . . . . . . . . . . . . . . . . . . . . **32**
**2.3 Vérification formelle** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **34**
2.3.1 Principes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.3.2 Preuve d'équivalence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
**2.4 Le *model-checking*** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **36**
2.4.1 Principes du *model-checking* . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.4.2 Mise en œuvre du *model-checking* . . . . . . . . . . . . . . . . . . . . . . . 38
2.4.3 Avantages et limites du *model-checking* . . . . . . . . . . . . . . . . . . . . 40
**2.5 Approches pour l'analyse et la vérification logicielle** . . . . . . . . . . . . . **41**
2.5.1 Approches par raffinement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.5.2 Approches avec une transformation vers un langage d'analyse . . . . 44
2.5.3 Approches d'analyse dédiées au langage . . . . . . . . . . . . . . . . . . . 47
2.5.4 Approches d'analyse par API . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
**2.6 Synthèse** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **51**

## 2.1 Introduction

Les langages offrent le moyen de définir des programmes pour implémenter ou modéliser des systèmes numériques. Cependant, la complexité croissante de ces systèmes rend leur développement sujet à davantage de bogues logiciels, d'erreurs de conception ou de comportements indéterminés. Pour détecter et corriger ces erreurs, différentes activités d'analyse peuvent être mises en œuvre. Certaines de ces activités sont dites formelles au sens où elles reposent sur des fondements mathématiques pour prouver qu'un système (ou une abstraction de ce système) satisfait certaines propriétés. Parmi les techniques de vérification formelle,

cette thèse aborde plus en détail le *model-checking* qui permet de vérifier des propriétés sur l'espace d'état d'un modèle du système ou de déceler des contre-exemples les réfutant.

La section 2.2 dresse un panorama des différentes techniques permettant de valider et de vérifier des programmes. La section 2.3 présente la vérification formelle avant de se focaliser sur le *model-checking* dans la section 2.4. Enfin, la section 2.5 identifie les différentes approches permettant d'appliquer ces techniques d'analyse en ciblant particulièrement le *model-checking*.

## 2.2 Panorama des techniques de V&V

Lors du développement d'un logiciel, diverses activités d'analyse peuvent être mises en œuvre par exemple pour évaluer les performances, déterminer la couverture de code, évaluer la qualité de codage ou encore détecter des fuites de mémoire. Parmi toutes ces activités d'analyse, cette thèse s'intéresse particulièrement aux techniques de V&V. Elles permettent de s'assurer que le système satisfait ses exigences et remplit le besoin initial pour lequel il a été élaboré. Informellement, la vérification vérifie que le logiciel a été *bien* conçu alors que la validation s'assure que c'est le *bon* logiciel qui a été développé. La norme ISO 9000 [ISO15], concernant les principes de management de la qualité, donne une définition rigoureuse de ces deux concepts.

**Définition 2.1 (Vérification).** *"Confirmation par des preuves tangibles que les exigences spécifiées ont été satisfaites."* [ISO15]

**Définition 2.2 (Validation).** *"Confirmation par des preuves tangibles que les exigences pour une utilisation spécifique ou une application prévue ont été satisfaites."* [ISO15]

Les activités de V&V permettent de détecter des erreurs. Une erreur correspond à un état d'exécution incorrect atteint par le système. Une erreur entraîne potentiellement des défaillances c.-à-d. des comportements du système non conformes à ses exigences. Une erreur est elle-même la manifestation d'une faute (en générale d'origine humaine) qui a causé l'erreur. Une faute peut donc provoquer des erreurs qui peuvent à leur tour engendrer des défaillances [Lap96].

Pour détecter des erreurs, différentes techniques de V&V peuvent être utilisées tout au long du cycle de développement d'un système ou d'un programme. Durant la phase de conception et de codage, il est intéressant de pouvoir inspecter et visualiser le comportement du programme via les activités de simulation, d'animation ou encore de débogage. La **simulation** permet de construire une trace d'exécution à partir d'un scénario prédéfini ou renseigné par l'utilisateur de façon interactive [Com08]. L'**animation** permet à l'utilisateur de *"voir évoluer son modèle tout au long de l'exécution et ainsi [de] contrôler visuellement le comportement du*

*système pour une exécution donnée"* [Com08]. Le **débogage** est le processus qui essaie de trouver et de corriger l'erreur à l'origine d'une défaillance [Zel05]. Le débogage *interactif* de modèles a également été défini plus récemment dans [Bou15]. Il permet à la fois de contrôler l'exécution du modèle en permettant la mise en pause et la reprise de l'exécution entre les pas, et d'observer le contenu de l'état d'exécution courant du programme pendant les pauses. De plus, les débogueurs modernes peuvent également avoir d'autres caractéristiques intéressantes. Ils peuvent être *omniscients* [Bou+15 ; Bou+17] s'ils implémentent la possibilité de replacer le moteur d'exécution dans un état donné à n'importe quel moment. Cette fonctionnalité peut notamment être utilisée pour reprendre l'exécution depuis un état précédent sans avoir à redémarrer l'exécution depuis son point de départ. Les débogueurs peuvent également être *multivers* [Tor+19] s'ils permettent d'observer et de déboguer tous les chemins d'exécution d'un programme concurrent (p. ex. en plaçant un point d'arrêt (ou en anglais *breakpoint*) sur plusieurs chemins d'exécution en même temps).

Une fois le programme implémenté, d'autres activités de V&V peuvent être menées dont le **test logiciel dynamique** qui permet de vérifier que, pour un ensemble de cas de test donnés, les résultats de l'exécution du programme sont en accord avec les résultats attendus [XRK99]. Pour effectuer du test, il est aussi possible d'appliquer des outils d'exécution symbolique [CS13] permettant d'explorer différents chemins d'exécution en un temps limité. Pour chaque chemin, elle détermine l'ensemble des valeurs d'entrées menant à ce chemin afin de détecter différents types d'erreurs (p. ex. des corruptions de mémoire, des exceptions logicielles non traitées). Les méthodes d'exécution concolique permettent quant à elles de mixer exécution concrète et exécution symbolique pour réaliser des tests [CDE08]. Par ailleurs, il est aussi possible de réaliser du **test logiciel statique**. À partir du code du programme, des revues de code peuvent être effectuées pour détecter des erreurs. Il est aussi possible d'appliquer des outils automatiques permettant de scanner le code source dans le but d'y trouver des vulnérabilités [Ben+19].

Des techniques de vérification formelle peuvent également être utilisées. On distingue généralement quatre activités de vérification formelle : le *model-checking*, l'interprétation abstraite, le *theorem proving* et le *monitoring* (ou *runtime verification*). Le ***model-checking*** est une technique permettant de vérifier de façon exhaustive une propriété sur un modèle abstrait du système [BK08 ; CGL94] (p. ex. SPIN [Hol97], Divine [Bar+17], OBP2 [BGT20]). L'**interprétation abstraite** consiste à utiliser une dénotation qui décrit l'exécution du programme avec des concepts abstraits afin que le résultat de l'exécution abstraite (= symbolique) apporte des informations sur l'exécution concrète [CC77] (p. ex. avec le plugin *Evolved Value Analysis* (EVA) de Frama-C [1] [Kir+15]). Le ***theorem proving*** permet de démontrer la véracité de propositions logiques sur des programmes en se basant sur des concepts mathématiques (p. ex. les types inductifs). Ces preuves sont réalisées via des *theorem provers* automatisés

1. Frama-C : `https://frama-c.com/`.

(p. ex. Z3 [2] [MB08], Alt-Ergo [3] [Con+18]), interactifs (aussi appelés assistants de preuves) (p. ex. Coq [4] [BC10], Isabelle [5] [Pau94], PVS [6] [ORS92]) ou une combinaison de plusieurs d'entre eux (p. ex. Why3 [7] [FP13]). Le **monitoring** (au sens *runtime verification*) permet de surveiller le comportement du programme lors de son exécution (p. ex. Java-MaC [Kim+04], MOP [CDR04], Temporal Rover [Dru00]) afin de déclencher des mécanismes de recouvrement ou de compensation d'erreurs en cas de défaillance.

Toutes ces techniques de V&V permettent d'analyser des programmes de façon statique ou dynamique. Les techniques statiques focalisent l'analyse sur le programme en lui-même (c.-à-d. sans l'exécuter) alors que les techniques dynamiques sont mises en œuvre sur une exécution (symbolique, concolique ou concrète) du programme. Un autre critère de distinction concerne les méthodes dites "formelles" de celles que ne le sont pas. Les techniques de vérification formelle sont basées sur des fondements mathématiques permettant de réaliser des preuves. Elles permettent en général d'obtenir des garanties plus fortes sur le respect des exigences logicielles sous condition que le modèle utilisé soit représentatif du système réel.

Cette thèse s'intéresse aux techniques de V&V pouvant être mises en œuvre dynamiquement sur l'exécution de programmes. La communauté du génie logiciel porte de plus en plus d'intérêt aux techniques de vérification formelle. C'est la raison pour laquelle ce projet de thèse s'intéresse en particulier à une technique dynamique de vérification formelle : le *model-checking*. Nous allons maintenant présenter plus en détail la vérification formelle avant de se focaliser sur les fondements du *model-checking*.

## 2.3 Vérification formelle

La vérification formelle permet de vérifier qu'un système est conforme à ses spécifications en s'appuyant sur des preuves mathématiques. La section 2.3.1 présente les principes de ce domaine puis la notion de preuve d'équivalence est exposée dans la section 2.3.2.

### 2.3.1 Principes

Pour les systèmes logiciels, la vérification formelle est définie comme le fait de prouver ou de réfuter l'exactitude d'un système (p. ex. un programme) par rapport à une spécification ou une propriété, en se basant sur l'utilisation de techniques mathématiques [Bje05]. Parmi les techniques de vérification formelle, on distingue deux grands types d'approches [Fer+92] :

2. Z3 : `https://github.com/Z3Prover/z3`.
3. Alt-Ergo : `http://alt-ergo.lri.fr/`.
4. Coq : `https://coq.inria.fr/`.
5. Isabelle : `https://isabelle.in.tum.de/`.
6. PVS : `https://pvs.csl.sri.com/`.
7. Why3 : `http://why3.lri.fr/`.

— Les méthodes basées sur les preuves (*theorem proving*) consistent à établir des preuves sur le code source des programmes pour vérifier qu'ils sont bien en adéquation avec les spécifications ;
— Les méthodes basées sur les modèles (*model-checking*, interprétation abstraite, *monitoring*) consistent à construire un modèle mathématique du système à vérifier (soit à partir du code source du programme soit en réalisant une abstraction du système) afin de pouvoir vérifier des propriétés sur ce modèle.

Dans les deux cas, ces méthodes formelles permettent d'augmenter la confiance dans le système réel en se basant sur de solides fondements mathématiques. Pour cette thèse, nous allons privilégier les techniques basées sur les modèles car elles peuvent être mises en œuvre sur des traces d'exécution du système. De plus, ces techniques peuvent être facilement automatisées ce qui facilitera leur utilisation par les ingénieurs.

### 2.3.2 Preuve d'équivalence

Une des applications phares de la vérification formelle concerne la preuve d'équivalence [LM17]. Étant donné deux spécifications $Spec_1$ et $Spec_2$ d'un même système, la preuve d'équivalence consiste à prouver que ces deux spécifications sont équivalentes par rapport à un objectif donné.

**Définition 2.3 (Relation d'équivalence).** Si $Spec_1$ et $Spec_2$ sont deux spécifications équivalentes d'un même système, on note :

$$Spec_1 \equiv Spec_2$$

Cette relation d'équivalence peut notamment être établie par bisimulation [Cla+05]. Cette technique consiste à établir une relation, entre deux systèmes de transitions, permettant d'affirmer que les deux systèmes ont le même comportement pour un observateur externe. On parle de simulation lorsque l'un des systèmes est capable d'imiter (c.-à-d. de simuler) l'autre. Si cette relation est réciproque, on dit que les systèmes sont équivalents par bisimulation. Les deux systèmes ont alors les mêmes traces d'exécution. Cette technique peut notamment être utilisée pour prouver que deux implémentations d'un même système sont équivalentes l'une à l'autre.

La relation d'équivalence peut également être établie à un autre niveau c.-à-d. entre un programme et les propriétés qu'il doit satisfaire. Le programme correspond à une implémentation du système à partir des spécifications qui décrivent les comportements possibles de celui-ci. La propriété formelle à vérifier peut-être vue comme une implémentation de la spécification décrivant les comportements à satisfaire (c.-à-d. les "bons" comportements). L'ensemble des traces d'exécution du programme et l'ensemble des traces d'exécution de la propriété forment

chacun un langage [8]. Le problème de la preuve d'équivalence revient donc à un problème d'inclusion de langages c.-à-d. il faut vérifier que le langage décrit par l'implémentation du système est inclus dans le langage défini par la propriété. Autrement dit, l'ensemble de tous les comportements possibles du système est-il inclus dans l'ensemble des "bons" comportements définis par la propriété ? Le *model-checking* résout une variante de ce problème et peut donc être vu comme une technique permettant d'établir la preuve d'équivalence. Nous allons maintenant définir et présenter cette technique de vérification formelle dont l'utilisation est au centre des travaux de cette thèse.

## 2.4 Le *model-checking*

Le *model-checking* est une technique de vérification formelle permettant de vérifier de façon exhaustive qu'une propriété est satisfaite sur un modèle d'un programme donné. Cette section présente les principes du *model-checking* dans la section 2.4.1, sa mise en œuvre dans la section 2.4.2 et enfin ses avantages et ses limites dans la section 2.4.3.

### 2.4.1 Principes du *model-checking*

Historiquement, l'invention du *model-checking* remonte aux travaux de Clarke et Emerson [CE81] et à ceux de Queille et Sifakis [QS08] au début des années 80. Une définition du *model-checking*, donnée dans [BK08], est :

**Définition 2.4 (Model-checking).** *"Model-checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model."* [BK08]

Autrement dit, le *model-checking* est une activité automatisée qui, étant donné un modèle d'états fini du système et une propriété formelle, vérifie de façon exhaustive que la propriété est satisfaite pour (un état donné dans) ce modèle.

Le principe de cette technique est décrit sur la Figure 2.1. On souhaite montrer qu'un *Système* (p. ex. ici la navette spatiale Discovery) satisfait ses *Spécifications*. Cependant, élaborer une telle preuve directement sur le système réel est une tâche trop complexe. Une abstraction du *Système* et des *Spécifications* est donc réalisée afin d'obtenir respectivement un *Modèle* et des *Propriétés*. En *model-checking*, le *Modèle* prend la forme d'un système de transitions dont les nœuds sont les différents états possibles du système et les transitions sont les pas

8. Attention, le sens du terme "langage" utilisé ici n'est pas celui d'un langage de modélisation permettant de décrire le comportement de différents programmes. Il correspond ici au langage (p. ex. une expression régulière pour un système très simple) permettant de décrire l'ensemble des traces d'un programme ou d'une propriété donnée.

d'exécution permettant de passer d'un état à un autre. Les *Propriétés* prennent généralement la forme de propositions logiques (p. ex. LTL, *Computational Tree Logic* (CTL)), de diagrammes de séquence (p. ex. *Property Sequence Chart* (PSC)), ou d'automates (p. ex. des automates de Büchi).

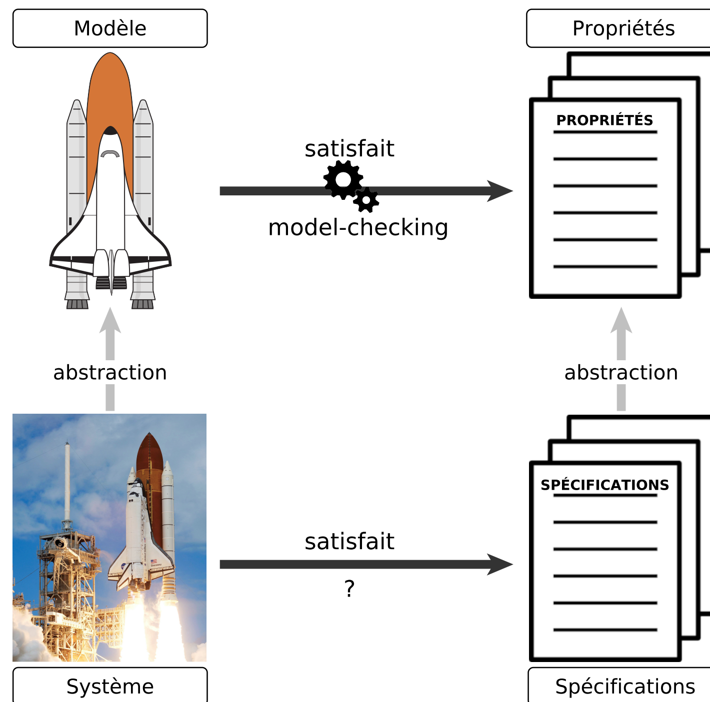



FIGURE 2.1 – Schéma de principe du *model-checking*.

Le problème de vérification formelle est donc maintenant transposé dans le domaine des mathématiques. Des algorithmes de *model-checking* peuvent ainsi être utilisés pour vérifier que le *Modèle* satisfait ses *Propriétés*. Ces algorithmes explorent l'espace d'état du modèle en énumérant tous les états d'exécution possibles et vérifient les propriétés sur l'ensemble des traces d'exécution du système.

**Définition 2.5.** Lorsqu'un *Modèle* satisfait ses *Propriétés*, on note :

$$\textit{Modèle} \models \textit{Propriétés}$$

Cependant, le *model-checking* ne peut être effectué que sur une abstraction du système en boucle fermée (c.-à-d. soumis à des sollicitations). Il est donc nécessaire de modéliser l'environnement dans lequel se trouve le système ainsi que les interactions avec celui-ci [Tka08 ; Dha+12]. Le *Modèle* à vérifier est donc constitué d'un modèle du système composé avec un modèle de son environnement. En *model-checking*, le modèle du système satisfait donc ses propriétés *relativement au modèle d'environnement utilisé* [Abr10].

**Définition 2.6.** Lorsqu'un modèle, constitué d'une abstraction du système, noté *Système*, et d'une abstraction de son environnement, noté *Environnement*, satisfait ses *Propriétés*, on note :

$$\{Système \parallel Environnement\} \models Propriétés$$

### 2.4.2 Mise en œuvre du *model-checking*

L'activité de *model-checking* se décompose en trois phases [BK08] : *(1)* la modélisation, *(2)* la vérification et *(3)* l'analyse des résultats.

**Modélisation.** La phase de modélisation permet de construire une abstraction du système et de son environnement sous la forme d'un modèle décrit dans un langage de modélisation. Cette phase comprend aussi l'écriture des propriétés formelles à partir des exigences à satisfaire. Ces propriétés sont décrites en utilisant un langage de spécification des propriétés. On distingue deux grandes catégories de propriétés formelles : *(i)* les propriétés de *sûreté* qui vérifient que quelque chose de mauvais n'arrivera pas et *(ii)* les propriétés de *vivacité* qui vérifient que quelque chose de bien finira par se produire.

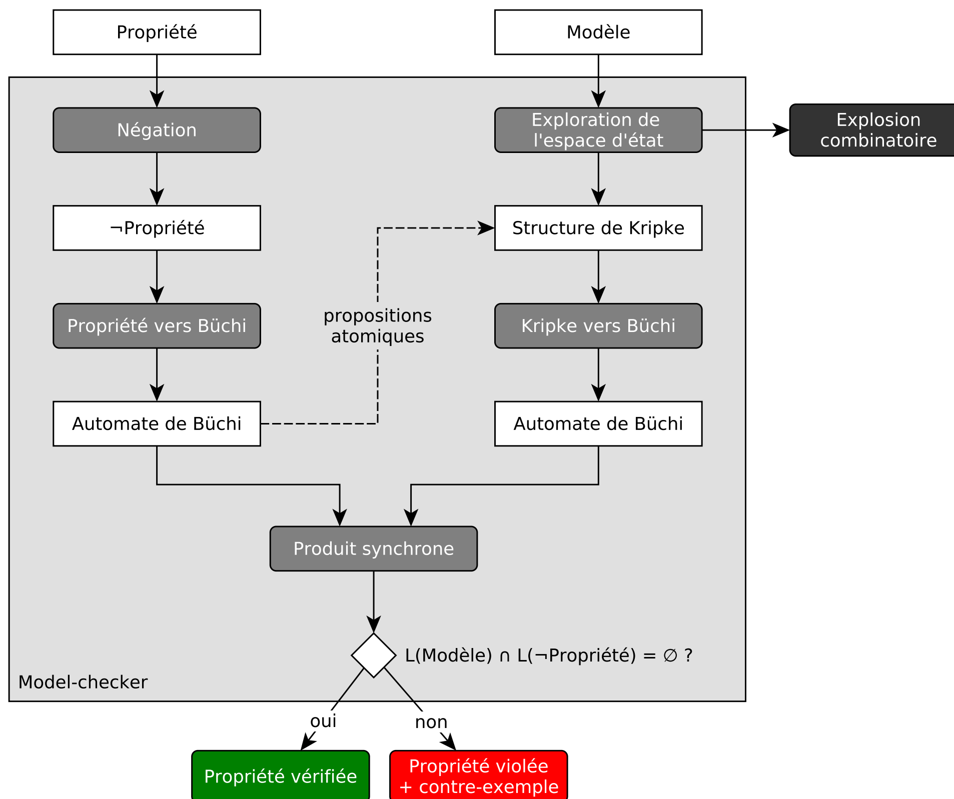



FIGURE 2.2 – Schéma de principe du *model-checking* avec automates de Büchi.

**Vérification.** La phase de vérification permet d'exécuter des algorithmes de *model-checking* afin de vérifier si un modèle satisfait des propriétés formelles. La Figure 2.2 expose le fonctionnement d'un *model-checker* basé sur les automates de Büchi, l'un des formalismes les plus répandus en *model-checking*. Un automate de Büchi est un type d'automates permettant de représenter des traces d'exécution infinies (des $\omega$-mots). On dit qu'il encode un $\omega$-langage.

Conformément à la Figure 2.2, le *model-checker* prend en entrée la *Propriété* et le *Modèle* sur lequel elle doit être vérifiée. On considère ici que la propriété est décrite à l'aide d'un langage de logique temporelle (p. ex. LTL, CTL). Elle est donc constituée de propositions atomiques (c.-à-d. des prédicats dépendants des objets du modèle) reliées entre elles par des opérateurs de logique du premier ordre (p. ex. non, et, ou) et des opérateurs temporels (p. ex. *globally*, *eventually*, *next*, *until*). Pour effectuer la vérification exhaustive, le *model-checker* prend la négation de cette propriété pour obtenir le langage $L(\neg\textit{Propriété})$ décrivant l'ensemble des comportements qui invalident la propriété. Un *Automate de Büchi* est ensuite généré à partir de la négation de cette propriété pour décrire l'ensemble des traces d'exécution redoutées.

En parallèle, le *model-checker* explore l'espace d'état du *Modèle* et, pour chaque état, évalue l'ensemble des *propositions atomiques* de la propriété. Cette opération permet de construire une *Structure de Kripke* [Kri63] représentant une abstraction du comportement du système pour l'ensemble des propositions atomiques considérées. Cette *Structure de Kripke* est ensuite transformée en *Automate de Büchi* afin de réaliser le *Produit synchrone* avec l'*Automate de Büchi* de la propriété. Le produit synchrone permet de calculer l'intersection du langage du système $L(\textit{Modèle})$ (c.-à-d. celui décrivant tous les comportements possibles du système) avec celui de la négation de la propriété $L(\neg\textit{Propriété})$ (c.-à-d. celui décrivant tous les comportements à éviter). Le produit synchrone permet ainsi de déterminer l'ensemble des comportements communs aux deux langages (c.-à-d. les comportements du système qui violent la propriété). En analysant le résultat du produit synchrone ($L(\textit{Modèle}) \cap L(\neg\textit{Propriété}) = \varnothing$ ?), il est possible d'affirmer si la propriété est vérifiée ou violée.

En règle générale, l'exécution de ces algorithmes peut être fait de façon explicite en explorant directement le système de transitions du modèle ou de façon symbolique en explorant une représentation abstraite de ce système de transitions (p. ex. en utilisant des *Binary Decision Diagram* (BDD) [Ake78 ; BK08]). L'exécution peut également être réalisée à la volée (ou en anglais *on-the-fly*) [GS09] afin d'effectuer en même temps la vérification de la propriété et l'exploration de l'espace d'état. Cette technique offre ainsi la possibilité d'arrêter l'algorithme de *model-checking* dès qu'un contre-exemple est trouvé et sans pour autant avoir exploré tout l'espace d'état.

**Analyse des résultats.** Enfin, les résultats retournés par le *model-checker* sont analysés. Ces résultats peuvent être de trois natures différentes :

— La propriété est vérifiée ;
— La propriété est violée et un contre-exemple est retourné par le *model-checker*. Il faut analyser la trace du contre-exemple pour identifier la faute à l'origine de l'erreur puis corriger le modèle en conséquence ;
— Le *model-checker* ne peut pas conclure sur la validité de la propriété car il n'est pas capable d'explorer tous les états du modèle dans un délai imparti et avec une quantité de mémoire donnée. On parle alors d'explosion combinatoire.

### 2.4.3 Avantages et limites du *model-checking*

Le *model-checking* offre de nombreux avantages [BK08] pour la vérification logicielle. Il s'agit d'une technique de vérification générique pouvant être appliquée sur de nombreux systèmes. Les *model-checkers* ont permis d'automatiser le processus de vérification. Ils sont généralement assez simples à utiliser et requièrent peu d'interactions avec l'utilisateur. Ils fournissent également un contre-exemple en cas de violation d'une propriété ce qui est très utile pour pouvoir comprendre, déboguer et corriger les erreurs. De plus, cette technique n'est pas sensible à la probabilité d'occurrence d'une erreur comme un débogueur ou un simulateur. Si une propriété est violée, une nouvelle exécution de l'algorithme de vérification exhaustive aboutira au même résultat.

Cette technique d'analyse possède également des limites [BK08] dont la principale est l'explosion combinatoire. L'apparition de supercalculateurs et l'utilisation d'algorithmes de *model-checking* symbolique ont permis de repousser ce problème mais les ressources de calcul utilisables par les *model-checkers* restent toujours limitées. Pour aller encore plus loin, des techniques de réduction d'ordre partiel, de composition, et de réduction par symétrie sont notamment utilisées [Bar+17 ; Hol97 ; Teo+16 ; Tka08]. Malgré la facilité d'utilisation des *model-checkers*, l'application de cette technique est néanmoins complexe car la spécification des propriétés et la modélisation des systèmes de façon formelle restent des tâches difficiles. De plus, cette technique est appliquée sur une abstraction du système et pas directement sur le système réel ce qui laisse toujours un risque d'erreur. Le modèle utilisé se doit donc d'être le plus représentatif possible du système réel pour garantir les résultats obtenus.

Après avoir présenté les techniques de V&V et notamment le *model-checking*, nous allons maintenant nous intéresser aux différentes approches permettant de mettre en œuvre ces techniques sur des modèles conformes à un langage donné.

## 2.5 Approches pour l'analyse et la vérification logicielle

Lors du développement d'un système logiciel, les activités d'analyse dynamiques sont primordiales pour garantir que le comportement du système satisfait ses exigences. Pour appliquer ces techniques d'analyse et plus généralement pour améliorer la fiabilité des systèmes, différentes approches issues des communautés du génie logiciel et de la vérification formelle ont été conçues et appliquées à travers différents projets de recherche. Ces approches peuvent se regrouper en quatre grandes familles :

1. Les **approches par raffinement** qui produisent des modèles corrects par construction en prouvant chaque modification (ou raffinement) par une preuve ;
2. Les **approches avec une transformation vers un langage d'analyse** bénéficient des outils d'analyse du langage cible ;
3. Les **approches d'analyse dédiées au langage** permettent d'utiliser des outils conçus spécifiquement pour le langage de modélisation ;
4. Les **approches d'analyse par *Application Programming Interface* (API)** permettent de connecter des moteurs d'exécution à des outils génériques d'analyse.

Pour chaque type d'approches, cette section présente une description de la méthode ainsi qu'un état de l'art des différentes techniques et outils pouvant être utilisés. Nous nous focalisons notamment sur les techniques de *model-checking* dont l'utilisation est au cœur de nos travaux.

### 2.5.1 Approches par raffinement

Les approches par raffinement consistent, à partir d'un modèle abstrait d'un système, à obtenir un modèle concret en procédant par dérivations successives. Chaque étape de raffinement doit être prouvée afin de garantir que le modèle dérivé est conforme au modèle abstrait utilisé. À l'instar du *model-checking*, ces techniques permettent de montrer que le modèle obtenu à chaque itération est un raffinement de la spécification et donc que le système modélisé satisfait ses exigences. Dans cette sous-section, la présentation du raffinement est inspirée de [Abr10 ; GFL05].

#### 2.5.1.1 Description

Les méthodes par raffinement ont émergé suite au constat que les systèmes complexes ne peuvent être conçus parfaitement et sans erreur en une seule itération. La conception de ces systèmes est une tâche difficile car elle requiert à la fois une vision globale de leur fonctionnement et une vision détaillée permettant d'en comprendre tous les rouages. Une technique est

donc de procéder par itérations successives afin d'enrichir progressivement le modèle du système. Pour décrire cette approche, la Figure 2.3 illustre le principe de conception d'un système à l'aide d'une méthode de raffinement.

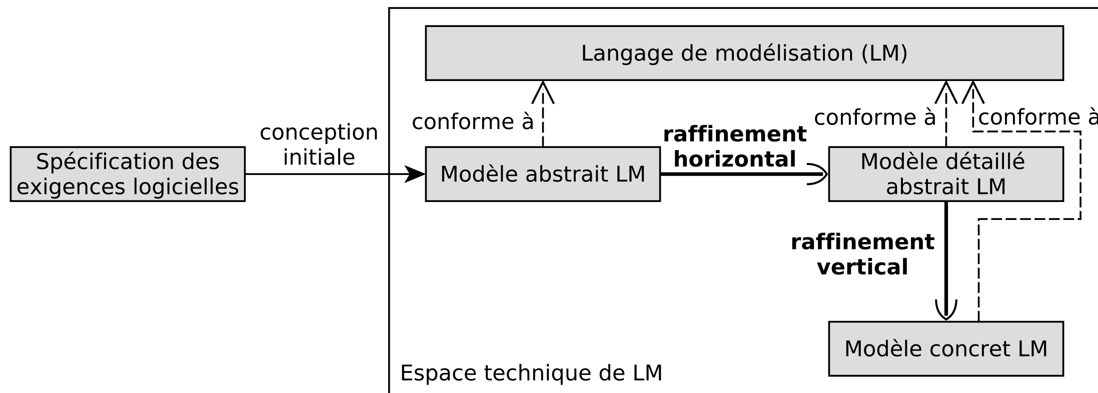



FIGURE 2.3 – Schéma de l'approche par raffinement.

À partir de la *Spécification des exigences logicielles*, un ingénieur conçoit un *Modèle abstrait* du système. Ce modèle est décrit à un haut niveau d'abstraction à l'aide d'un *Langage de modélisation*. Par itérations successives, l'ingénieur va ensuite détailler les différents sous-composants du système afin d'enrichir le modèle. Cette activité est appelée *raffinement horizontal* (ou superposition) [Abr10]. Elle permet de passer d'un *Modèle abstrait* à un *Modèle détaillé abstrait* dans lequel toutes les exigences logicielles du système ont été considérées. Le raffinement horizontal est indépendant de toutes contraintes d'implémentation ou choix technologiques. Il ne se base que sur des outils mathématiques afin d'obtenir une représentation formelle du système modélisé. À chaque étape, des preuves de raffinement, aussi appelées obligations de preuve, doivent être réalisées afin de garantir que le pas de raffinement conserve la cohérence et les exigences prises en compte dans les raffinements précédents. Une obligation de preuve peut être définie de la façon suivante :

**Définition 2.7 (Obligation de preuve).** *"Une obligation de preuve est une formule mathématique à démontrer afin d'assurer qu'un composant [...] est correct."* [Cle09]

Une fois que toutes les exigences logicielles ont été prises en compte et qu'une représentation abstraite du système a été réalisée, l'ingénieur utilise alors le *raffinement vertical* [Abr10] pour transformer certains éléments abstraits en éléments concrets (plus proches des concepts d'implémentation). Les différentes itérations de raffinement vertical permettent ainsi de passer d'un *Modèle détaillé abstrait* à un *Modèle concret* à partir duquel il est facile de générer du code. À chaque étape de raffinement vertical, il faut aussi vérifier des obligations de preuve afin d'assurer que les types de données concrets et les choix d'implémentation du modèle plus

concret restent cohérents avec les exigences logicielles du modèle plus abstrait. C'est pour cette raison que le raffinement vertical est aussi qualifié de raffinement de données (ou en anglais *data refinement*).

En résumé, les approches par raffinement permettent de faire le lien entre les différents niveaux d'abstraction d'un même modèle. Ces techniques permettent ainsi d'assurer qu'un modèle satisfait ses exigences. Elles constituent en ce sens une alternative aux travaux réalisés dans cette thèse. Plus qu'une alternative, elles offrent également une certaine complémentarité par rapport aux autres techniques d'analyse (p. ex. la simulation, le *model-checking*). En effet, l'application de ces outils reste possible (voire primordial) pour valider le comportement du système modélisé dans chacun des niveaux d'abstraction considérés.

#### 2.5.1.2 Langages et outils

Pour mettre en œuvre les approches par raffinement, plusieurs langages formels ont été définis dont Z [WD96 ; Spi92], *Vienna Development Method* (VDM) [Jon95] et B [Abr96 ; Abr+91]. Si la méthode Z ne permet pas de fournir de version exécutable des programmes Z, des outils ont été développés pour supporter les méthodes VDM et B. Pour VDM, le langage de modélisation *VDM Specification Language* (VDM-SL) [PL92] est supporté par SpecBox [FM89] ou encore VDMTools [FLS+08] qui est aujourd'hui l'outil de référence pour VDM. L'un des premiers succès notables de VDM a été de donner une définition formelle de la sémantique du langage PL/I développé par IBM dans les années 60. Pour B, plusieurs outils ont également été développés dont B Toolkit [9] [LH00] ou encore l'Atelier B [10] commercialisé par la société ClearSy [11]. L'un des premiers projets utilisant l'Atelier B visait la vérification de la ligne 14 du métro parisien. Il a depuis été utilisé par de nombreux projets industriels dont les lignes de métro automatique de Pékin et São Paulo. De plus, pour valider le comportement des systèmes modélisés en B, il est également possible d'appliquer des outils d'analyse comme le *model-checker* ProB [LB08] sur les modèles de chaque niveau d'abstraction.

Les langages de raffinement ne se limitent pas à ces trois langages. Des variantes de ces langages utilisent le paradigme objet comme VDM++ [DK92], Z++ [Lan90] ou ObjectZ [Smi12]. Pour la méthode B, un profil nommé UML-B a également été défini pour permettre l'utilisation des principes de B avec UML (cf. section A.3.4). Si B est particulièrement adapté pour la conception logicielle, son homologue Event-B [Abr10] basé sur le paradigme évènementiel est dédié à la conception système.

Les langages utilisés pour les approches par raffinement sont riches et expressifs permettant ainsi la modélisation de systèmes complexes. L'outillage supportant ces langages contri-

9. B Toolkit : `https://github.com/edwardcrichton/BToolkit`.
10. Atelier B : `https://www.atelierb.eu/`.
11. ClearSy System Engineering : `https://www.clearsy.com/`.

bue aussi à l'émergence de ces techniques et aux succès industriels qui ont été accomplis. Ces approches utilisent néanmoins des langages spécifiques ce qui les rend difficilement applicables pour le développement de systèmes avec d'autres langages de modélisation.

### 2.5.2 Approches avec une transformation vers un langage d'analyse

Pour assurer qu'un modèle de conception satisfait ses exigences, une autre stratégie consiste à transformer ce modèle dans un langage cible afin de pouvoir réutiliser tous les outils d'analyse disponibles dans l'espace technique cible.

#### 2.5.2.1 Description

Le principe général de cette approche est illustré sur la Figure 2.4. Un *Modèle LM* décrivant le système à concevoir est conçu avec le *Langage de modélisation*. Cependant, l'*Espace technique de LM* ne possède pas d'outils d'analyse permettant l'analyse des modèles décrits dans LM. Pour résoudre ce problème, un *Traducteur* capture la sémantique de LM afin de traduire le *Modèle LM* dans un autre langage que l'on qualifiera de *Langage d'analyse* noté LA. Cette *transformation* permet ainsi d'obtenir le *Modèle LA*, c.-à-d. une représentation du modèle de conception dans les termes de LA. L'*Espace technique de LA* possède divers *Outils d'analyse* implémentant la sémantique de LA et pouvant s'appliquer aux modèles décrits dans LA. L'utilisation d'une transformation permet ainsi de réutiliser les outils d'analyse existants sans avoir à repayer le coût de leurs développements pour LM. De plus, un changement de la syntaxe abstraite ou de la sémantique de LM nécessite seulement de mettre à jour la transformation sans avoir besoin de changer les outils d'analyse de LA. La transformation permet ainsi de faire le pont entre le langage de modélisation LM et un langage d'analyse LA.

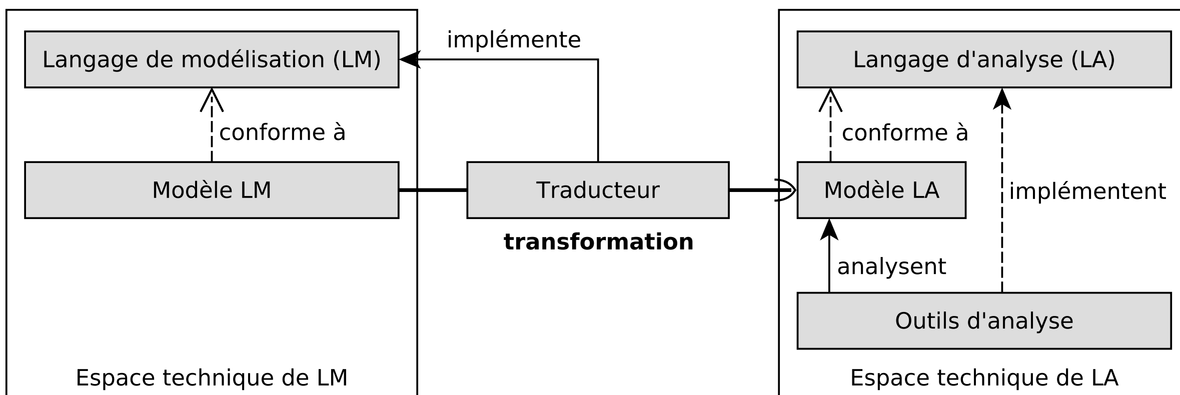



FIGURE 2.4 – Schéma de l'approche utilisant une transformation vers un langage d'analyse.

Même si cette technique offre de nombreux avantages, l'utilisation d'une transformation de modèles fait apparaître un fossé sémantique entre le modèle de conception et le modèle d'analyse. Ce fossé sémantique complexifie notamment la compréhension des résultats d'analyse d'un point de vue de LM. En effet, les langages LM et LA reposent généralement sur des concepts différents rendant ainsi difficile d'établir une correspondance entre les concepts de ces deux langages.

Pour résoudre ce problème, plusieurs techniques peuvent être utilisées. Le *round-trip engineering* [Cic14] consiste à annoter le modèle de conception avec des *back-annotations* [Heg+10] permettant ainsi de traduire les résultats d'analyse en termes des concepts de LM. Une autre méthode est de construire des liens de traçabilité [Jou05 ; Bou15] lors de la transformation du modèle de conception vers le modèle d'analyse. Ces liens de traçabilité peuvent ensuite être utilisés a posteriori pour exprimer les résultats d'analyse avec les concepts de LM. Ces techniques permettent ainsi de résoudre un des problèmes majeurs causés par le fossé sémantique.

L'un des points clés de cette approche par transformation vers un langage d'analyse réside dans le choix de LA. Les critères de choix dépendent notamment de la variété des outils disponibles dans le langage cible et de la facilité à implémenter la transformation de LM vers LA.

#### 2.5.2.2 Langages et outils

Pour des raisons historiques, cette approche par traduction est typiquement employée pour transformer des modèles vers des langages formels et ainsi appliquer des algorithmes de *model-checking*. Il existe en effet un certain nombre de *model-checkers* dont le formalisme d'entrée est un langage formel conçu uniquement dans l'objectif de pouvoir modéliser et vérifier formellement des systèmes logiciels. En général, ces outils ne peuvent s'appliquer qu'à des modèles spécifiés dans un seul et unique langage d'entrée. Ce langage est alors qualifié de langage pivot car des modèles écrits dans divers langages peuvent être transformés vers ce langage unique afin d'appliquer des outils d'analyse.

**Model-checking.** Parmi ces outils, le *model-checker* Spin [12] [Hol97] est utilisé pour vérifier des applications logicielles concurrentes spécifiées dans le langage Promela. Cet outil a notamment été utilisé pour vérifier des systèmes industriels dont des algorithmes pour des missions sur Mars et certains protocoles de transmission de données dans le domaine médical. De manière similaire, le *model-checker* OBP1 [Dha+12 ; Teo+16 ; TDL17] a été développé à l'ENSTA Bretagne pour permettre la vérification de modèles conformes au langage

12. Spin : `http://spinroot.com/spin/whatispin.html`.

Fiacre [Ber+08]. Ce langage a initialement été développé pour l'atelier *Toolkit in Open Source for Critical Applications & Systems Development* (TOPCASED) [Far+06] comme un langage pivot entre différents langages de modélisation (p. ex. UML, *Specification and Description Language* (SDL), *Architecture Analysis and Design Language* (AADL)) et les langages formels des *model-checkers*. D'autres *model-checkers* ont aussi été développés avec un langage d'entrée spécifique dont CADP [Gar+13] pour la vérification de programmes écrits en LOTOS, Tina-selt [BV06] et Romeo [Lim+09 ; Gar+05] pour les modèles de Petri-Nets temporisés, Mocha [Alu+98] et PRISM [KNP09] pour les modèles utilisant des variantes des modules réactifs, TLC [YML99] pour les modèles en TLA+ ou encore ProB [LB08] pour des modèles écrits en B. D'autres outils ont également leur propre langage d'entrée comme SMV [McM92], NuSMV [Cim+02], NuXmv [Cav+14], Maude [Cla+03] ou Uppaal [LPY97]. FDR [FW99] permet quant à lui de vérifier les raffinements des modèles de l'algèbre des processus *Communicating Sequential Processes* (CSP). Alcoa [JSS00] réalise des analyses sur des modèles Alloy en se basant sur des solveurs de contraintes. Enfin, Bogor [RDH03] est un *model-checker* modulaire ayant un langage d'entrée spécifique mais extensible pour s'adapter aux constructions des *Domain Specific Language*s (DSLs).

**Outils de transformation.** Pour traduire les modèles dans des langages formels, il est possible d'utiliser les outils génériques de transformation de modèles (cf. section A.2.1.3) comme *ATLAS Transformation Language* (ATL) [JK06], *Query/View/Transformation* (QVT™) [OMG16] ou *Epsilon Transformation Language* (ETL) [KPP08]. D'autres outils permettent d'automatiser la phase de transformation mais ils sont spécifiques à un seul formalisme d'entrée. Ces outils intègrent généralement le *model-checker* directement comme *back-end* afin de proposer un environnement complet de vérification pour un langage donné. Parmi ces outils, on trouve notamment les outils Hugo/RT [KMR02 ; KW06] et vUML [LP99] qui permettent d'appliquer le *model-checker* Spin en transformant les modèles UML vers Promela (description plus détaillée en section A.3.4). On peut également citer Java PathFinder v1 [HP00] et Bandera [Cor+00] pour la vérification de programmes Java avec Spin, Cadena [Hat+03] pour la vérification de modèles Corba avec Spin ou Mbeddr [Voe+12] pour la vérification de programmes C avec NuSMV.

Certains de ces outils implémentent également une transformation des résultats d'analyse en termes des concepts du langage d'entrée. C'est par exemple le cas des outils Hugo/RT et vUML qui fournissent une représentation des résultats en UML notamment sous forme de diagrammes de séquence. En dehors du *model-checking*, d'autres travaux offrent des exemples d'application du *round-trip engineering* pour monitorer des propriétés non-fonctionnelles en [Cic14], ou de l'utilisation des liens de traçabilité pour analyser des traces [Sad+19 ; CGR11] ou faire du débogage [BMW17 ; BW19].

### 2.5.3 Approches d'analyse dédiées au langage

Pour analyser et vérifier des modèles conformes à un langage, il est possible de concevoir des outils d'analyse spécifiquement pour ce langage.

#### 2.5.3.1 Description

Le schéma conceptuel de cette méthode est présenté en Figure 2.5. Un *Modèle* de conception du logiciel à développer se conforme au *Langage de modélisation*. Pour analyser ce modèle, l'approche employée consiste à développer des *Outils d'Analyse* applicables directement (et uniquement) dans l'*Espace technique de LM*. Pour concevoir ces outils, deux types d'approches peuvent être utilisées : *(1)* des approches ad-hoc applicables à un seul langage et *(2)* des approches systématiques permettant de générer (semi-)automatiquement les outils à partir des caractéristiques (p. ex. la syntaxe abstraite) de LM.

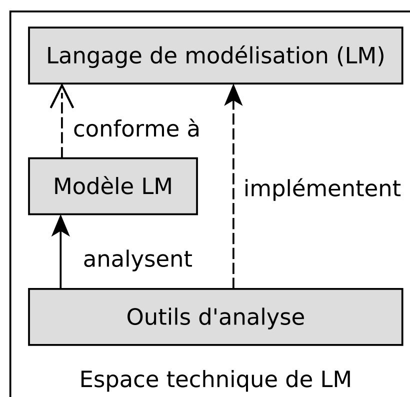



FIGURE 2.5 – Schéma de l'approche d'analyse dédiée au langage.

Lorsque les ingénieurs travaillent avec un seul langage, la conception d'un outil d'analyse est souvent réalisée de façon ad-hoc. Dans ce cas, l'outil peut facilement être personnalisé et optimisé pour s'adapter aux spécificités du langage. L'inconvénient est que ce type de développement ad-hoc doit être réalisé pour chaque nouveau langage.

Pour résoudre ce problème, des approches systématiques permettent de générer (semi-) automatiquement des outils d'analyse pour un langage donné. Ces approches utilisent des méta-outils permettant de générer un ou des outils d'analyse à partir du métamodèle du langage considéré. L'avantage de cette technique est que les méta-outils peuvent être facilement réutilisés pour être appliqués à un autre langage. En revanche, un des challenges est de réussir à prendre en compte les spécificités du langage pour d'une part optimiser le fonctionnement de l'outil et d'autre part obtenir un environnement d'analyse dédié au langage. À titre d'exemple,

les résultats d'analyse ou encore la vue graphique de l'outil doivent être exposés à l'utilisateur avec des concepts de LM et non pas avec des concepts génériques dénués de sens.

#### 2.5.3.2 Outils

De nombreux outils d'analyse ont été construits spécifiquement pour un langage de modélisation notamment pour faire du débogage et du *model-checking*. On distingue ici les outils utilisant une approche ad-hoc des outils utilisant une approche systématique.

**Ad-hoc.** Pour le débogage de programmes, plusieurs outils ont été développés comme GDB [13] pour les programmes C ou JDB [14] pour les programmes Java. Ces techniques d'analyse spécifiques sont aussi plébiscitées par les langages de modélisation. MDebugger [BHD17] est intégré à l'EDI Papyrus-RT [HDB17] pour déboguer des modèles UML-RT. Voyageur [Tor+19] est un débogueur multivers permettant de mettre des points d'arrêts sur l'ensemble des chemins d'exécution d'un programme concurrent AmbientTalk.

Pour le *model-checking*, des outils ont également permis l'adaptation des algorithmes de vérification formelle à un langage donné. Les *model-checkers* USMMC [Liu+13b] et UMC [GM04 ; GM05] ont été conçus pour vérifier des modèles UML dont le comportement est décrit à l'aide de machines à états (description plus détaillée en section A.3.4). À un niveau d'abstraction plus faible, Java PathFinder (v2 et supérieures) (JPF) [Bra+00] permet de *model-checker* des programmes concurrents écrits en Java. L'exploration de l'espace d'état est réalisée de façon explicite dans une machine virtuelle Java (ou en anglais *Java Virtual Machine* (JVM)) personnalisée. Une extension appelée JPF-SE [PR10] permet aussi de réaliser l'exploration de façon symbolique et donc de solliciter les programmes Java avec des entrées de domaines non bornés lors de la vérification (p. ex. un entier appartenant à l'intervalle $[0; +\infty[$).

**Systématique.** Pour les techniques de génération automatique d'outils, les travaux d'Erwan Bousse [Bou+15 ; Bou+17] proposent un méta-outil permettant de générer un débogueur omniscient pour n'importe quel langage de modélisation. À partir de la syntaxe abstraite du langage, un débogueur est généré automatiquement et intégré comme addon au sein de Gemoc Studio [Bou+16]. Il est également intéressant de noter les travaux autour de K framework [RŞ10] qui est utilisé pour décrire la sémantique des langages de programmation en s'appuyant sur des techniques de réécriture. K framework fournit des sémantiques exécutables et permet d'obtenir automatiquement des outils d'analyse comme un débogueur ou un *model-checker* LTL en s'appuyant sur les outils génériques de réécriture fournis par Maude [Cla+03]. Des expérimen-

13. GDB : `https://www.gnu.org/software/gdb/`.
14. JDB : `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`.

tations utilisant ces outils ont notamment été menées pour analyser des programmes C [ER12] et Java [BR15].

### 2.5.4 Approches d'analyse par API

Pour analyser et vérifier des modèles conformes à un langage, une autre démarche possible est d'utiliser une API pour connecter un outil d'analyse générique à un moteur d'exécution du modèle. L'outil d'analyse peut ainsi contrôler et analyser l'exécution du modèle à travers les différentes requêtes de l'API.

#### 2.5.4.1 Description

Le schéma conceptuel de cette approche est illustré sur la Figure 2.6. Un *Modèle* du système à concevoir, conforme au *Langage de modélisation*, est exécuté par un *Moteur d'exécution*. Ce moteur permet d'exécuter des modèles conformes à LM. Pour vérifier que le modèle satisfait ses exigences, des *Outils d'analyse* génériques peuvent se connecter au *Moteur d'exécution* via un lien de communication (*lien de com*). Ces outils d'analyse peuvent ainsi contrôler, observer et analyser l'exécution du modèle via une API ou un protocole de communication. Cette interface de communication doit être la moins contraignante possible pour le moteur d'exécution tout en offrant suffisamment de services aux outils d'analyse. Cette technique offre l'avantage de réutiliser le moteur d'exécution de LM et de découpler les algorithmes de vérification des préoccupations spécifiques au langage de modélisation. Elle facilite ainsi l'interopérabilité entre les outils et la réutilisation des algorithmes de vérification.

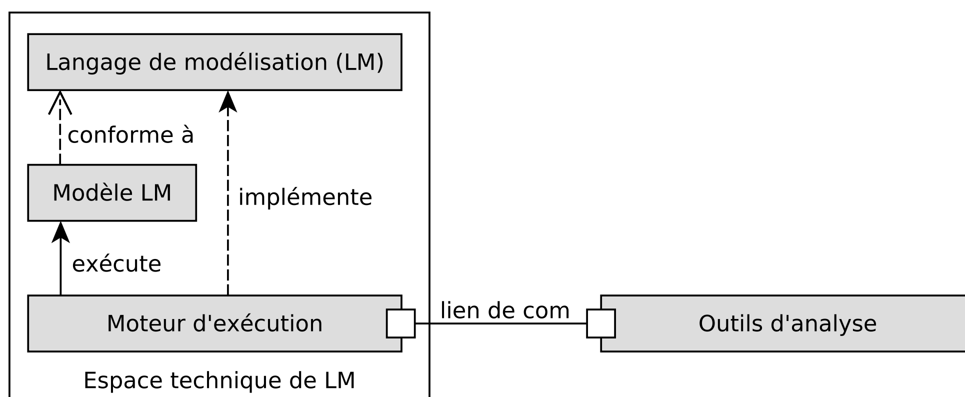



FIGURE 2.6 – Schéma de l'approche d'analyse par API.

#### 2.5.4.2 Outils et protocoles

L'émergence de cette technique a notamment été facilitée par la création de nouveaux protocoles de communication et d'outils d'analyse modulaires.

**Protocoles de Communication.** Des protocoles ont été créés afin de faciliter la manipulation des langages et les interactions avec leurs outils de développement. On trouve notamment le *Langage Server Protocol* (LSP) [15] qui permet d'interfacer un EDI avec un serveur de langages regroupant les différents services propres à un langage (p. ex. l'autocomplétion, l'accès à la documentation). Un même serveur de langages peut ainsi être utilisé par différents éditeurs. Une extension de ce protocole est notamment utilisée par Gemoc Studio [Bou+16] pour connecter des addons (p. ex. des débogueurs) à différents moteurs d'exécution.

De façon similaire à LSP, le *Debug Adapter Protocol* (DAP) [16] permet de connecter des débogueurs à des outils de développement afin de fournir les fonctionnalités usuelles au débogage (p. ex. l'ajout de points d'arrêt). Il vise ainsi à uniformiser les APIs de débogage propres à chaque EDI (p. ex. Eclipse, IntelliJ IDEA).

Le standard *Functional Mockup Interface* (FMI) [17] définit quant à lui un protocole permettant à la fois de faire communiquer et de co-simuler des modèles de natures différentes (p. ex. informatique, électronique, mécanique, thermodynamique).

**Model-checking.** Concernant le *model-checking*, aucun protocole dédié n'a encore été standardisé au même titre que le DAP pour le débogage. Le LSP paraît néanmoins déjà pouvoir être adapté à cet usage même s'il ne semble pas encore être utilisé dans ce but. Plusieurs *model-checkers* modulaires ont tout de même été développés en incluant la possibilité d'interagir avec des moteurs d'exécution externes. Cette approche offre ainsi l'avantage de pouvoir appliquer un même *model-checker* sur des modèles conformes à des langages différents.

En termes d'API, l'exploration de l'espace d'état d'un modèle nécessite trois requêtes : *(i)* une pour récupérer la configuration (c.-à-d. les données d'exécution) initiale du modèle, *(ii)* une pour récupérer la liste des transitions tirables à partir d'une configuration donnée et *(iii)* une pour tirer une transition (c.-à-d. pour exécuter un pas d'exécution). Cette API est exactement celle qui est implémentée par les *model-checkers* LTSmin [Kan+15] et OBP2 [18]. Les outils DIVINE [Bar+17] et SPOT [DP04] implémentent quant à eux une variante de cette interface avec seulement deux requêtes : une pour récupérer la configuration initiale et une pour récupérer la liste des successeurs immédiats d'une configuration donnée. Les requêtes *(ii)* et *(iii)* sont donc

15. Language Server Protocol : `https://microsoft.github.io/language-server-protocol/`.
16. Debug Adapter Protocol : `https://microsoft.github.io/debug-adapter-protocol/`.
17. Functional Mock-up Interface : `https://fmi-standard.org/`.
18. OBP2 : `http://www.obpcdl.org/`.

regroupées en une seule, ce qui a pour effet de masquer les pas d'exécution.

Pour évaluer des propriétés formelles, il est également nécessaire d'évaluer des propositions atomiques (aussi appelées atomes). Si un consensus semble avoir été trouvé sur l'API d'exploration de l'espace d'état, l'évaluation des propositions atomiques est toujours faite de façon ad-hoc sur chaque outil. Sur LTSmin, DIVINE et SPOT, les atomes sont évalués directement par le *model-checker* sur l'ensemble des configurations explorées. Dans ce cas, le *model-checker* et le moteur d'exécution doivent utiliser une structuration unifiée des données pour la configuration. Par exemple pour LTSmin, la configuration est composée de vecteurs d'entiers de taille fixe permettant ainsi au *model-checker* de lire et d'interpréter les données. Pour éviter de contraindre la structure de la configuration, OBP2 utilise une requête de communication dédiée permettant de déléguer l'évaluation des propositions atomiques au moteur d'exécution.

## 2.6 Synthèse

Dans ce chapitre, nous avons étudié quatre types d'approches permettant l'analyse de modèles conformes à un langage donné : *(1)* les approches par raffinement, *(2)* les approches avec une transformation vers un langage d'analyse, *(3)* les approches d'analyse dédiées au langage et *(4)* les approches d'analyse par API.

Le Tableau 2.1 synthétise les avantages et inconvénients de chaque approche selon les différents critères de comparaison : (LM == LA) indiquant si le formalisme d'analyse est identique au formalisme de modélisation, (Résultats) indiquant si les résultats d'analyse sont facilement compréhensibles, et (Réutilisabilité outils) indiquant si les outils d'analyse sont applicables à d'autres langages.

<table>
<tr><th colspan="2">Types d'approches</th><th>LM == LA</th><th>Résultats [19]</th><th>Réutilisabilité outils</th></tr>
<tr><td colspan="2">Par raffinement</td><td>oui</td><td>+</td><td>non</td></tr>
<tr><td rowspan="2">Avec transformation vers LA</td><td>sans traçabilité</td><td rowspan="2">non</td><td>-</td><td rowspan="2">oui</td></tr>
<tr><td>avec traçabilité</td><td>+</td></tr>
<tr><td rowspan="2">Dédiée au langage</td><td>ad-hoc</td><td rowspan="2">oui</td><td rowspan="2">+</td><td>non</td></tr>
<tr><td>systématique</td><td>oui (méta-outils)</td></tr>
<tr><td colspan="2">Par API</td><td>oui</td><td>+</td><td>oui</td></tr>
</table>

TABLE 2.1 – Synthèse des approches d'analyse de modèles.

On peut remarquer que les approches avec une transformation vers un langage d'analyse sont les seules pour lesquelles les outils d'analyse ne sont pas appliqués directement sur le

19. Résultats : + (exprimés en termes de LM), - (exprimés en termes de LA)

modèle de conception mais sur une traduction de ce modèle dans un autre langage. L'utilisation d'une transformation crée un fossé sémantique entre le modèle de conception et le modèle d'analyse. En conséquence, les résultats d'analyse ne sont plus exprimés en termes des concepts de LM. L'ajout de liens de traçabilité permet néanmoins de résoudre ce problème sans pour autant résoudre le fossé sémantique.

Les approches par raffinement et les approches dédiées ad-hoc ne permettent pas de réutiliser les outils d'analyse pour d'autres langages que celui pour lequel ils ont été conçus. Les approches dédiées systématiques et les approches d'analyse par API semblent donc les plus pertinentes car leurs outils sont applicables directement sur le modèle de conception tout en restant réutilisables pour d'autres langages.

CHAPITRE 3

# APPROCHES POUR L'EXÉCUTION ET L'ANALYSE DE MODÈLES

---

**Sommaire**


**3.1 Introduction** . . . . . . . . . . . . . . . . . . . . **53**
**3.2 Contexte de développement** . . . . . . . . . . . . . . **54**
**3.3 Problèmes pour l'analyse et l'exécution réelle** . . . . . . . . **55**
3.3.1 Fossés sémantiques . . . . . . . . . . . . . . . . 55
3.3.2 Problème d'équivalence . . . . . . . . . . . . . . 56
**3.4 Approche 1 : traduction vers modèle vérifiable et code** . . . . . **58**
**3.5 Approche 2 : traduction vers modèle vérifiable puis code** . . . . **59**
**3.6 Approche 3 : traduction vers code vérifiable** . . . . . . . . . **60**
**3.7 Approche 4 : traduction modèle vérifiable vers code** . . . . . . **61**
**3.8 Approche 5 : raffinement de modèles jusqu'au code** . . . . . . **62**
**3.9 Approche 6 : interprétations spécifiques pour vérification et exécution réelle** . . . . . . . . . . . . . . . . . . . . **63**
**3.10 Analyse des six approches** . . . . . . . . . . . . . . **64**
**3.11 Approche X : interprétation unifiée pour vérification et exécution réelle** . **65**
**3.12 Objectifs de recherche** . . . . . . . . . . . . . . . **67**
**3.13 Synthèse** . . . . . . . . . . . . . . . . . . . . **68**


## 3.1 Introduction

Le chapitre 1 a présenté les deux principales techniques permettant d'exécuter des modèles : l'interprétation et la génération de code. Le chapitre 2 a décrit les différentes méthodes permettant d'appliquer des outils de V&V dynamiques sur ces modèles. Ce chapitre présente maintenant les différentes approches permettant de combiner ces différentes techniques d'exécution et d'analyse. En ce qui concerne l'exécution, il faut distinguer l'exécution du modèle pour la mise en œuvre des techniques d'analyse et l'exécution au sens déploiement du modèle sur la plateforme d'exécution réelle. Ce sont ces deux aspects que l'on considère ici. Ce chapitre

distingue six approches permettant à la fois d'appliquer des techniques de V&V sur des modèles et de déployer ces modèles sur une plateforme d'exécution.

Le Tableau 3.1 expose une classification de ces six approches (A#1 à A#6) en fonction des techniques d'analyse de modèles et des techniques d'exécution réelle. Il permet de positionner ces approches les unes par rapport aux autres mais offre également une vue d'ensemble des méthodes actuellement utilisées dans la littérature pour aborder ce sujet. Il positionne également l'approche A#X utilisée dans cette thèse pour concevoir l'approche EMI.

| Analyse / Exécution | Raffinement | Transformation | Dédiée | API |
|---|---|---|---|---|
| Compilation | AC#5 | AC#1<br>AC#2 | AC#3<br>AC#4 | AC#4 |
| Interprétation | - | - | AC#6 | AC#X |

TABLE 3.1 – Positionnement des approches par rapport aux techniques d'analyse et d'exécution de modèles.

La section 3.2 présente le contexte de développement considéré dans cette thèse. La section 3.3 s'intéresse aux fossés sémantiques et au problème d'équivalence communément rencontrés dans la littérature. Les sections 3.4 à 3.9 décrivent chacune des six approches d'analyse et d'exécution de modèles. La section 3.10 identifie les trois problèmes majeurs auxquels elles sont confrontées puis la section 3.11 propose une nouvelle architecture permettant d'éviter ces problèmes. Pour mettre en œuvre cette architecture, des objectifs de recherche ont été identifiés en section 3.12. Enfin, la section 3.13 présente une synthèse de cet état de l'art.

## 3.2 Contexte de développement

Afin de mieux comprendre les besoins pour le développement d'un système logiciel, cette section propose un cycle de développement itératif permettant de modéliser, d'exécuter et d'analyser des systèmes. Ce cycle itératif ne constitue qu'une proposition essayant d'exploiter au mieux différentes techniques de V&V tout au long du cycle de développement. Les différentes phases de ce cycle de développement sont :

1. **Modélisation du système.** Cette phase consiste à concevoir le modèle du système ainsi que le ou les modèles d'environnement abstraits utilisés pour la phase d'analyse. Pour aider le concepteur dans cette tâche, le modèle peut être chargé dans le moteur d'exécution à des fins de simulation ou de débogage. Ces activités d'analyse interactives permettent en général d'identifier les principales erreurs de conception et inconsistances du modèle.

2. **Vérification formelle du modèle.** Les propriétés formelles issues des exigences sont vérifiées sur le modèle grâce au *model-checking*. Cette phase permet alors de détecter des *deadlocks* ou des erreurs de conception plus subtiles.
3. **Déploiement du modèle sur la plateforme d'exécution réelle.** Le modèle est déployé sur la plateforme d'exécution réelle du système (p. ex. une cible embarquée). On parle alors d'*exécution réelle* lorsque le système s'exécute afin de satisfaire le besoin pour lequel il a été conçu.
4. ***Monitoring* du système à l'exécution.** Il est possible de continuer la vérification des propriétés formelles à l'exécution en utilisant des moniteurs pour surveiller la trace d'exécution courante.

Ce cycle de développement est itératif. À chaque fois qu'une erreur de conception ou qu'un bogue logiciel est détecté, l'utilisateur doit mettre à jour le modèle de conception en proposant une correction. La modélisation du système bénéficie ainsi du retour des activités d'analyse pour pouvoir être améliorée d'itération en itération.

Grâce à ce cycle de développement, on constate que l'approche recherchée doit satisfaire trois besoins : *(1)* la possibilité d'exécuter un modèle conforme à n'importe quel langage, *(2)* le fait de pouvoir analyser l'exécution d'un modèle avec différents outils et différentes techniques d'analyse dynamiques (p. ex. simulation, débogage, *model-checking*) et *(3)* la possibilité de déployer un modèle sur une plateforme d'exécution réelle.

## 3.3 Problèmes pour l'analyse et l'exécution réelle

Dans la littérature, plusieurs approches permettent de répondre à ces différents besoins en combinant les techniques d'exécution et les techniques d'analyse de modèles. Ces approches reposent en général sur des transformations non prouvées qui sont à l'origine de plusieurs problèmes : des fossés sémantiques et un problème d'équivalence. Pour comprendre pourquoi ces problèmes se manifestent, cette section les présente plus en détail.

### 3.3.1 Fossés sémantiques

Un fossé sémantique apparaît lorsqu'un même modèle est exprimé dans deux langages différents. L'utilisation de transformations exogènes (c.-à-d. des transformations pour lesquelles le langage source est différent du langage cible) est typiquement à l'origine de fossés sémantiques. En effet, l'outil servant à la transformation capture (tout ou partie de) la sémantique du langage source pour exprimer les éléments du modèle source en termes des concepts du langage cible. Ces transformations (p. ex. transformations de modèles, génération de code) sont souvent vues comme bénéfiques car elles permettent d'utiliser les outils d'analyse du langage

cible pour vérifier des modèles sources. Néanmoins, l'apparition d'un fossé sémantique rend l'établissement de liens entre les concepts du modèle source et ceux du modèle cible plus complexe. L'utilisation de transformations crée un risque d'erreur supplémentaire puisque sans preuve formelle, rien ne garantit que la correspondance entre les éléments sources et les éléments cibles est correcte. Dans certains cas, le fossé sémantique impacte également les outils d'analyse en les séparant en deux groupes : ceux s'appliquant au niveau du modèle et ceux s'appliquant au niveau du code exécutable. Ce problème de fossé sémantique est mentionné dans plusieurs travaux de la littérature [DT16 ; RDH03] et cité dans [Dub+13] comme l'un des challenges scientifiques clés pour la vérification et la validation des modèles.

Pour éviter les problèmes causés par les fossés sémantiques, une solution est de prouver formellement chacune des transformations utilisées mais cette tâche reste complexe. De nouvelles preuves doivent être établies pour chaque langage de modélisation pour lequel on souhaite analyser ou exécuter des modèles. Il est également possible d'utiliser des liens de traçabilité mais ceux-ci ne peuvent se substituer à des preuves formelles. Une autre solution est tout simplement d'éviter l'utilisation des transformations afin de conserver les concepts du langage de modélisation pour l'analyse et l'exécution.

### 3.3.2 Problème d'équivalence

Le problème d'équivalence se manifeste dès que le modèle d'analyse servant à la mise en œuvre des activités de V&V est différent du modèle exécutable (c.-à-d. du code exécutable déployé sur le système réel) et qu'aucune relation d'équivalence entre les deux n'a été établie et prouvée. Dès qu'une transformation non prouvée est utilisée, un problème d'équivalence apparaît en plus d'un fossé sémantique. Dans ce cas, il suffit de prouver la transformation pour résoudre le problème. En revanche, si le flot de développement se sépare en plusieurs branches, le problème d'équivalence devient plus complexe à résoudre. Par exemple, si deux transformations sont appliquées à partir du modèle de conception pour obtenir à la fois des modèles d'analyse et le code exécutable du système, prouver indépendamment ces deux transformations n'est plus suffisant. Il faut construire une preuve d'équivalence (cf. section 2.3.2). Cette preuve (ou relation) d'équivalence peut être construite entre différents artefacts :

1. Entre les modèles d'analyse et le code exécutable ;
2. Entre les différents outils qui capturent la sémantique du langage de conception (c.-à-d. entre l'outil de transformation de modèles qui permet d'obtenir un modèle d'analyse et l'outil de génération de code) ;
3. Entre la plateforme d'exécution réelle et la plateforme d'exécution utilisée pour les activités d'analyse de modèles.

Dans le premier cas, il faut prouver que les modèles sont équivalents (à la fois d'un point de vue structurel et comportemental). Dans le second cas, la preuve doit garantir que les différents outils de transformation capturent exactement la même définition de la sémantique du langage de conception. Enfin, dans le dernier cas, il faut prouver que la plateforme d'exécution dédiée à l'analyse du modèle se comporte comme la plateforme réelle pour l'utilisation qui en est faite.

Par ailleurs, le problème d'équivalence peut se manifester par des différences à plusieurs niveaux :

**Au niveau de l'interprétation de la sémantique** : Un même concept peut être interprété différemment par des outils différents. Ce problème peut se produire lorsqu'un concept a une signification complexe ou lorsque la description de sa sémantique est ambiguë. Par conséquent, ce concept devient la source d'interprétations potentiellement erronées menant ainsi à des exécutions discordantes entre les différents outils (p. ex. pour *Business Process Model And Notation* (BPMN™) en [BGT20] ou pour UML-RT en [PD16]). Ceci est d'autant plus vrai pour les langages non formels ou semi-formels pour lesquels il existe souvent des points de variation sémantique.

**Au niveau du sous-ensemble capturé** : Des traces d'exécution différentes peuvent être obtenues si tous les outils ne capturent pas le même sous-ensemble du langage de conception. Par exemple, si le modèle de conception possède un concept qui est inclus dans le sous-ensemble capturé par l'outil d'analyse mais pas dans celui capturé par le générateur de code, alors le comportement du système obtenu via le code exécutable sera potentiellement différent de celui analysé durant la phase de V&V.

**Au niveau du chargement du modèle** : Chaque outil doit charger le modèle de conception en mémoire avant de pouvoir le transformer ou le manipuler. Le chargement du modèle peut être vu comme une transformation (texte vers représentation binaire) qui ne garantit pas que le modèle chargé en mémoire est fidèle au modèle de conception. Les multiples implémentations des mécanismes de chargement de modèles sont donc une source supplémentaire de dissemblances entre les différents outils (p. ex. débogueurs, *model-checkers*).

**Au niveau de la considération de la plateforme d'exécution** : Les plateformes d'exécution utilisées lors des phases d'analyse considèrent parfois certaines abstractions (p. ex. sur le modèle de concurrence, la communication interprocessus, l'ordonnancement) [Bar+17]. Changer la politique d'ordonnancement peut par exemple changer complètement le comportement du système et l'ordre dans lequel les tâches sont exécutées.

Toutes ces différences peuvent potentiellement impacter le comportement du système lors des phases d'analyse et d'exécution. Une relation d'équivalence doit donc être établie et prouvée afin de garantir que ce qui est exécuté est conforme à ce qui a été vérifié. Cependant, cette relation d'équivalence reste complexe à élaborer dans le cas général d'un langage quelconque.

Pour résoudre le problème d'équivalence, plusieurs solutions peuvent être envisagées. La

première consiste à établir et à prouver formellement la relation d'équivalence en utilisant par exemple du *theorem proving* ou des preuves de bisimulation [Fer90 ; Com+09]. Ce type de preuve reste cependant complexe à mettre en œuvre pour un langage de modélisation quelconque. Une autre solution consiste à utiliser un générateur de code certifié (p. ex. KCG pour SCADE [Ber07]) à partir du modèle d'analyse. Avec cette méthode, l'équivalence n'est prouvée qu'avec le modèle d'analyse ayant servi de référence pour la génération de code. Cependant, plusieurs modèles d'analyse utilisant des formalismes différents sont souvent nécessaires pour appliquer divers outils de V&V. Une autre alternative est de garantir l'équivalence à l'exécution par exemple en utilisant des moniteurs. Cette solution ne permet pas pour autant de se substituer à une preuve formelle. Elle ne peut servir qu'à augmenter la confiance des ingénieurs dans le système.

Les sections qui suivent vont maintenant présenter six approches permettant d'exécuter, d'analyser et de déployer des modèles sur une plateforme d'exécution. L'ordre dans lequel les approches sont présentées n'est pas significatif mais seulement didactique.

## 3.4 Approche 1 : traduction vers modèle vérifiable et code

Le schéma conceptuel de la Figure 3.1 illustre la première approche (A#1) employée pour analyser et exécuter des modèles. Elle est notamment mise en œuvre au sein de l'outil Mbeddr [Voe+12]. Sur cette figure, un *Modèle LM* du système à concevoir est élaboré à partir de la spécification des besoins et des exigences logicielles. Ce modèle est conforme au langage de modélisation dans lequel il est décrit (p. ex. UML, *Systems Modeling Language* (SysML®)). À partir de ce modèle de conception, deux transformations sont utilisées.

FIGURE 3.1 – Schéma de A#1.

**Transformation vers des modèles d'analyse.** Cette première opération permet de transformer le modèle de conception (*Modèle LM*) en un ou plusieurs modèles d'analyse (*Modèles LA*) à l'aide d'outils de transformation de modèles. Les modèles produits peuvent ainsi être exploités par des *Outils d'analyse haut niveau* permettant de simuler, analyser et vérifier ces modèles à un haut niveau d'abstraction. Certains de ces outils d'analyse ont leur propre formalisme d'entrée. Diverses transformations de modèles sont donc nécessaires pour convertir le modèle de conception vers ces différents formalismes.

**Transformation vers du code exécutable.** La seconde opération correspond à la génération de code automatique, semi-automatique, ou manuelle du modèle de conception (*Modèle LM*) en *Code* exécutable. Le code machine obtenu peut alors être déployé sur la *Plateforme d'exécution* réelle (p. ex. un microcontrôleur embarqué) afin de pouvoir s'exécuter. L'exécution du système suit le comportement décrit dans le modèle de conception et peut interagir avec l'*Environnement* physique du système via les entrées/sorties (*I/O*) de la plateforme d'exécution. Ce code exécutable peut également être analysé via des *Outils d'analyse bas niveau* (p. ex. GDB) c.-à-d. directement au niveau du code généré.

Cette approche souffre de trois problèmes majeurs : *(i)* un fossé sémantique entre le modèle de conception et le(s) modèle(s) d'analyse, *(ii)* un fossé sémantique entre le modèle de conception et le code exécutable, et *(iii)* un problème d'équivalence entre le(s) modèle(s) d'analyse et le code exécutable. Le problème d'équivalence se manifeste ici autant entre le(s) modèle(s) d'analyse et le code exécutable qu'entre les outils de transformation qui capturent la sémantique du langage (c.-à-d. entre l'outil de transformation de modèles et le générateur de code).

## 3.5 Approche 2 : traduction vers modèle vérifiable puis code

Cette approche (A#2) est une variante de A#1. Elle consiste à effectuer la génération de code à partir d'un des modèles d'analyse comme dans RoboChart [Miy+19] ou Spin [Hol97] qui possède un générateur de code C. Cette approche se base sur les mêmes techniques de transformations que A#1 (c.-à-d. transformation de modèles et génération de code), ce qui leur permet notamment de manipuler des modèles et donc d'opérer à un haut niveau d'abstraction. Cependant, en partageant les mêmes outils, cette variante souffre aussi des mêmes problèmes que l'approche originale.

Pour le modèle choisi comme référence pour la génération de code, le problème d'équivalence se réduit au problème du fossé sémantique créé par le générateur de code. Cependant, les liens entre les éléments du modèle de conception et les portions de code exécutable deviennent encore plus difficiles à établir car ils sont désormais séparés par deux fossés sémantiques. Pour les autres modèles d'analyse (non choisis comme référence pour la génération de code), une relation d'équivalence doit être établie comme dans A#1. Le problème n'est donc

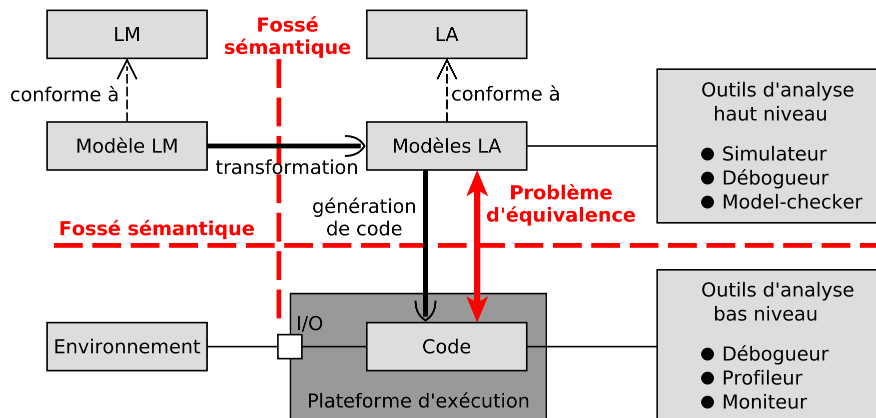



FIGURE 3.2 – Schéma de A#2.

simplifié que si tous les outils d'analyse se basent sur le même formalisme d'entrée, ce qui est rarement le cas en pratique.

## 3.6 Approche 3 : traduction vers code vérifiable

La troisième méthode (A#3) applique les activités d'analyse directement sur le code exécutable. Le code déployé est donc bien celui qui a été vérifié. Néanmoins, cette approche nécessite des outils d'analyse (p. ex. Divine [Bar+17]) applicables au niveau du code et non plus au niveau du modèle de conception ce qui rend l'analyse plus complexe. Cette approche est illustrée sur la Figure 3.3.

Pour appliquer des techniques d'analyse (p. ex. des méthodes de vérification formelle), certains outils (p. ex. Divine [Bar+17]) ont besoin d'une abstraction de la plateforme d'exécution [Roč+19] appelée ici *Plateforme d'exécution pour l'analyse*. D'une part, modéliser une telle abstraction est une tâche complexe et fastidieuse car elle doit abstraire tous les services de la plateforme d'exécution sous-jacente (p. ex. le modèle de concurrence, les mécanismes de communication interprocessus, l'ordonnancement). D'autre part, pour assurer l'équivalence entre l'analyse et l'exécution réelle, il est nécessaire de prouver que l'abstraction modélisée est conforme à la plateforme d'exécution réelle. En résumé, cette approche souffre d'un problème de fossé sémantique (causé par la génération de code) ainsi que d'un problème d'équivalence entre la plateforme d'exécution et son abstraction. De plus, elle ne permet plus d'analyser le modèle à un haut niveau d'abstraction. Elle perd donc certains des avantages offerts par les modèles sans pour autant résoudre les problèmes mentionnés précédemment.

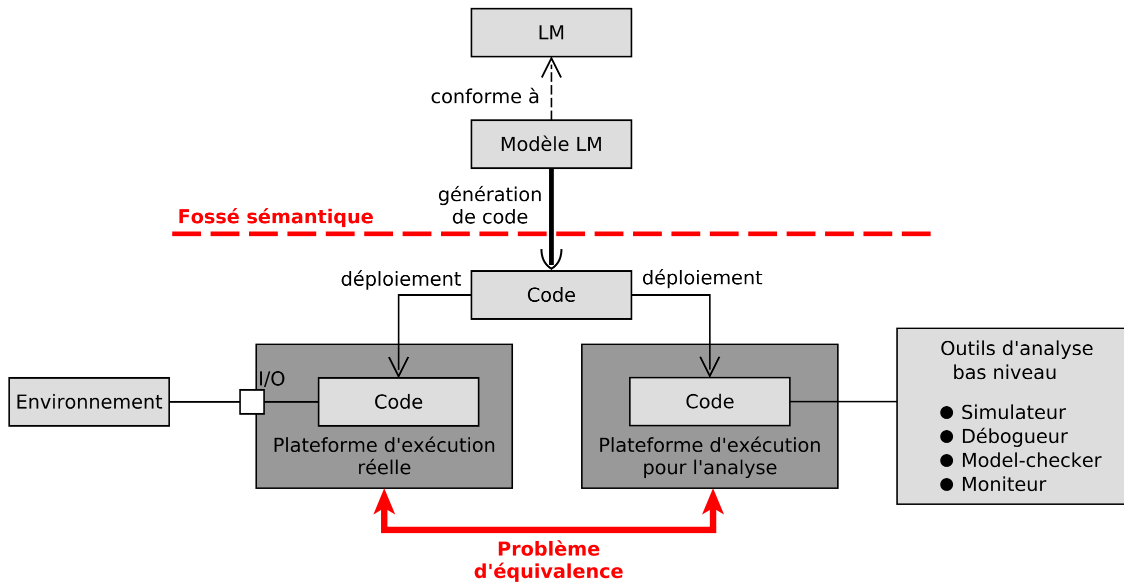



FIGURE 3.3 – Schéma de A#3.

## 3.7 Approche 4 : traduction modèle vérifiable vers code

La quatrième approche (A#4) vise l'utilisation d'outils d'analyse dédiés au langage de modélisation ou d'outils d'analyse par API. Elle utilise ensuite un générateur de code pour déployer le modèle sur une plateforme d'exécution réelle. Cette approche peut notamment être mise en œuvre en combinant des outils d'analyse comme SPOT [DP04] ou LTSmin [Kan+15] avec des générateurs de code. Le schéma conceptuel de cette approche est présenté sur la Figure 3.4.

En utilisant ce type d'outils d'analyse, cette approche n'introduit pas de fossé sémantique entre le modèle de conception et le modèle d'analyse puisque les techniques d'analyse sont

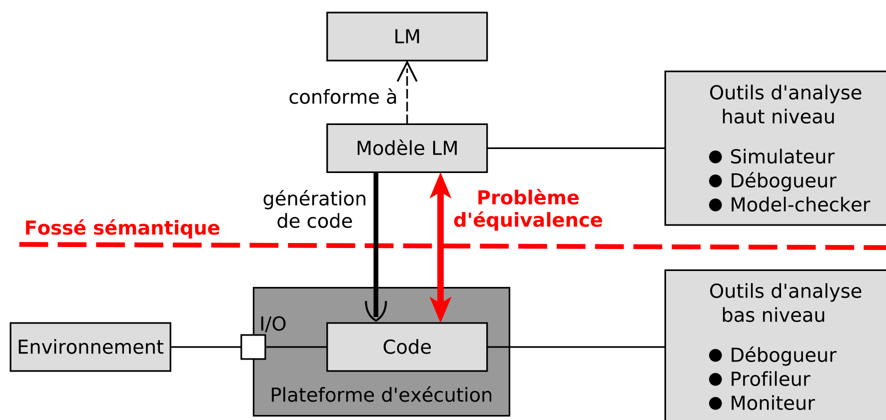



FIGURE 3.4 – Schéma de A#4.

directement mises en œuvre sur le modèle de conception. En revanche, le déploiement du modèle sur la plateforme d'exécution nécessite l'emploi d'un générateur de code. Cette transformation du modèle de conception en code exécutable introduit donc à la fois un fossé sémantique et un problème d'équivalence.

## 3.8 Approche 5 : raffinement de modèles jusqu'au code

L'approche (A#5) utilise les méthodes de raffinement pour construire un modèle de conception permettant ensuite de générer du code. Cette méthode, illustrée sur la Figure 3.5, est celle qui est utilisée dans l'Atelier B [1]. Le modèle de conception est construit par raffinement afin de passer d'un modèle abstrait à un modèle de plus en plus concret (c.-à-d. proche des concepts d'implémentation). Ce modèle peut ensuite être déployé sur une plateforme d'exécution en utilisant de la génération de code.

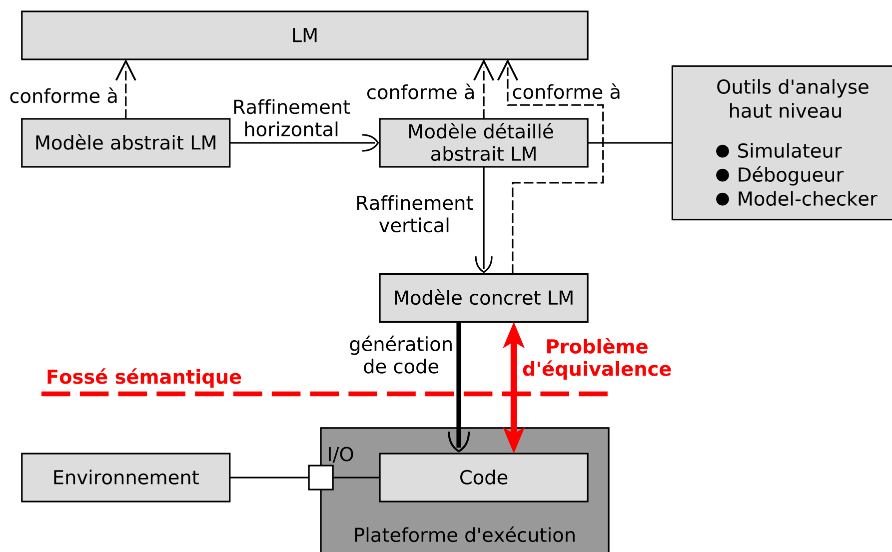



FIGURE 3.5 – Schéma de A#5.

Malgré l'utilisation des techniques de raffinement qui permettent de garantir que le modèle servant de référence pour la génération est conforme aux exigences, l'utilisation d'un générateur de code crée un fossé sémantique et un problème d'équivalence entre le modèle concret et le code exécutable. Une possibilité pour y remédier est d'utiliser des générateurs de code certifiés. Néanmoins, certifier un générateur de code reste une tâche complexe et ce coût doit être assumé intégralement pour chaque langage de modélisation.

1. Atelier B : `https://www.atelierb.eu/`.

## 3.9 Approche 6 : interprétations spécifiques pour vérification et exécution réelle

La sixième approche (A#6) est similaire à A#3 mais elle utilise les techniques d'interprétation pour exécuter le modèle. Le schéma conceptuel de cette approche est illustré sur la Figure 3.6. Pour appliquer des techniques d'analyse, le *Modèle LM* de conception du système est déployé sur une *Plateforme d'exécution pour l'analyse* ayant son propre interpréteur de modèles (*Interpréteur analyse*). Cet environnement d'exécution (plateforme + interpréteur) fournit une abstraction de l'environnement d'exécution réel pour analyser dynamiquement le comportement du système. Lorsque la modélisation du système satisfait ses exigences, le même *Modèle LM* peut être déployé sur la *Plateforme d'exécution réelle* avec un autre interpréteur de modèles (*Interpréteur réel*) offrant souvent de meilleures performances d'exécution.

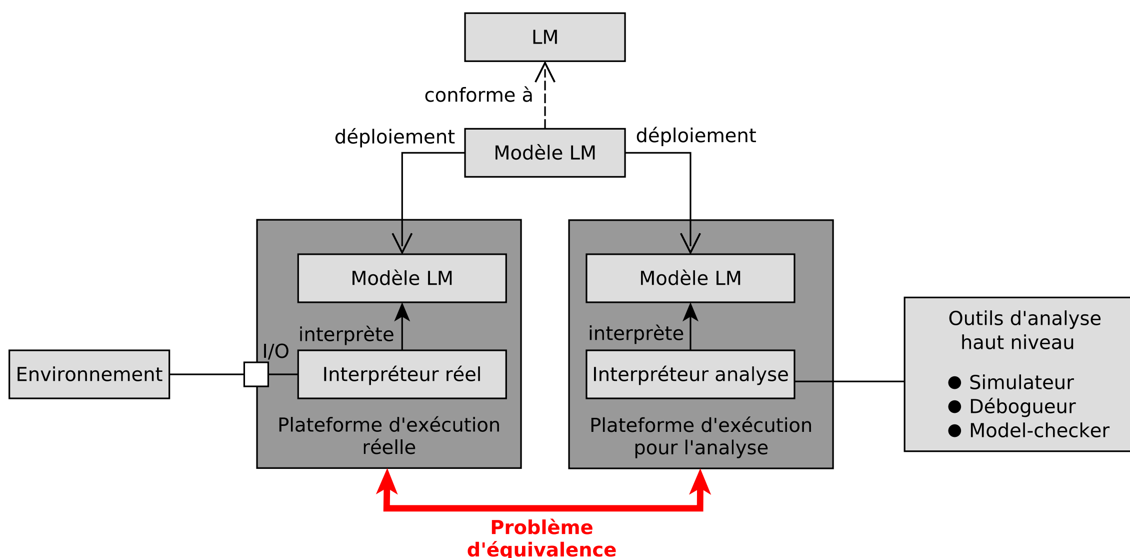



FIGURE 3.6 – Schéma de A#6.

Cette approche est notamment utilisée par le *model-checker* Java PathFinder [Bra+00] qui permet d'appliquer des algorithmes de *model-checking* sur une JVM modifiée dans cet objectif. Les programmes Java vérifiés peuvent ensuite être déployés et exécutés sur une JVM classique.

Comme dans A#3, cette approche souffre d'un problème d'équivalence entre l'environnement d'exécution servant aux activités d'analyse et celui servant à l'exécution réelle. En revanche, contrairement à A#3, c'est directement le modèle de conception qui est exécuté évitant ainsi la création d'un fossé sémantique.

## 3.10 Analyse des six approches

Ces six approches permettent toutes d'analyser des modèles et de les déployer sur des plateformes d'exécution. L'étude de ces solutions a cependant permis de montrer qu'au moins trois problèmes majeurs subsistent.

**P#1 Un fossé sémantique entre le modèle de conception et les modèles d'analyse**. Avec ce premier fossé sémantique, il devient difficile d'établir des liens entre les éléments du modèle de conception et les éléments des modèles d'analyse. Cela complexifie notamment la compréhension des résultats donnés par les outils d'analyse car ces résultats ne sont pas exprimés en termes des concepts du langage de modélisation mais avec ceux du langage d'analyse.

**P#2 Un fossé sémantique entre le modèle de conception et le code exécutable**. Avec ce second fossé sémantique, il devient difficile de relier les éléments du modèle de conception à des portions de code exécutable. Ce fossé sémantique impacte aussi les outils d'analyse, les séparant en deux catégories : ceux agissant directement au niveau du modèle (dits de haut niveau) et ceux agissant au niveau du code exécutable (dits de bas niveau).

**P#3 Un problème d'équivalence.** Il est nécessaire d'établir, de prouver et de maintenir une relation d'équivalence entre chacun des modèles d'analyse et le code exécutable pour assurer que ce qui a été vérifié en phase de V&V reste vrai lors de l'exécution. Ce problème d'équivalence peut aussi se manifester entre les outils de transformation utilisés pour produire les modèles d'analyse et le code exécutable, ou entre les plateformes d'exécution utilisées pour les activités d'analyse et l'exécution réelle.

| Problèmes \ Approches | A#1 | A#2 | A#3 | A#4 | A#5 | A#6 | X |
|---|---|---|---|---|---|---|---|
| P#1 | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| P#2 | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| P#3 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |

TABLE 3.2 – Récapitulatif des problèmes de chaque approche (✗ signifie que le problème est présent et ✓ qu'il est absent).

Le Tableau 3.2 dresse un récapitulatif des problèmes de chacune des six approches. Le problème P#1 n'est présent que sur A#1 et A#2 tandis que le fossé sémantique P#2 se manifeste également sur les approches A#3 à A#5. Le problème d'équivalence est, quant à lui, présent sous une forme ou une autre dans chacune des six approches mentionnées. Ces trois problèmes sont la source potentielle d'erreurs de conception, de bogues, ou de failles de sécurité mais ils complexifient également le développement des logiciels et la mise en œuvre des activités d'analyse.

En somme, l'approche qui semble la plus pertinente par rapport aux objectifs de cette thèse est A#6. Elle reste cependant affectée par le problème d'équivalence. Pour y remédier, l'objectif serait d'unifier l'environnement d'exécution réel avec l'environnement d'analyse afin d'employer la même sémantique du langage de modélisation pour toutes ces activités Avec cette hypothèse, les activités d'analyse et d'exécution réelle utiliseraient ainsi le même interpréteur en plus d'utiliser le même *Modèle LM*. Cette dernière approche figure dans le Tableau 3.2 sous le nom A#X car c'est l'approche qui va être eXaminée dans cette thèse.

Avant de présenter cette approche plus en détail, il convient de rappeler les hypothèses de travail que nous avons considérées dans cette thèse. *(i)* À partir de la documentation du système, les ingénieurs sont capables de modéliser une abstraction de l'environnement du système qui soit exhaustive et conforme à l'environnement réel. Cette abstraction permet de fermer l'exécution du système en phase d'analyse. Sa conformité à l'environnement réel permet de garantir la validité des résultats obtenus par vérification formelle. *(ii)* L'exécution réelle utilise un ordonnancement dans lequel il n'y a pas de création de processus et pas de prise en compte des interruptions.

## 3.11 Approche X : interprétation unifiée pour vérification et exécution réelle

Comme illustrée sur la Figure 3.7, l'approche X consiste à déployer directement le *Modèle LM* sur la *Plateforme d'exécution* afin qu'il soit exécuté par un *Interpréteur* de modèles. La spécificité de cet *Interpréteur* est qu'il repose sur **une unique implémentation de la sémantique de LM** pour toutes les activités de développement logiciel. Pour l'exécution réelle, cet *Interpréteur* peut interagir avec l'*Environnement* du système via les entrées/sorties (*I/O*) de la *Plateforme d'exécution*. Pour les activités d'analyse, les *Outils d'analyse haut niveau* peuvent se connecter directement à ce même *Interpréteur* pour pouvoir piloter et observer l'exécution du modèle. Cette approche permet ainsi d'appliquer les activités d'analyse directement sur le modèle chargé en mémoire par l'*Interpréteur* tout en réutilisant la même implémentation de la sémantique que pour l'exécution réelle.

L'approche X repose sur les techniques d'interprétation (cf. section 1.3.3) pour l'exécution du modèle et sur les techniques d'analyse par API (cf. section 2.5.4) pour les activités d'analyse. Pour centraliser tous les outils de développement logiciel autour de cette même définition de la sémantique, l'interpréteur doit être **pilotable** via une *API* pour permettre aux outils d'analyse de contrôler l'exécution du modèle.

Avec cette approche, il n'existe plus de limites distinctes entre les activités de conception et d'exécution créant ainsi un *continuum* entre ces deux phases. Les mêmes outils d'analyse peuvent ainsi être utilisés tout au long du cycle de développement du système. Grâce à ce

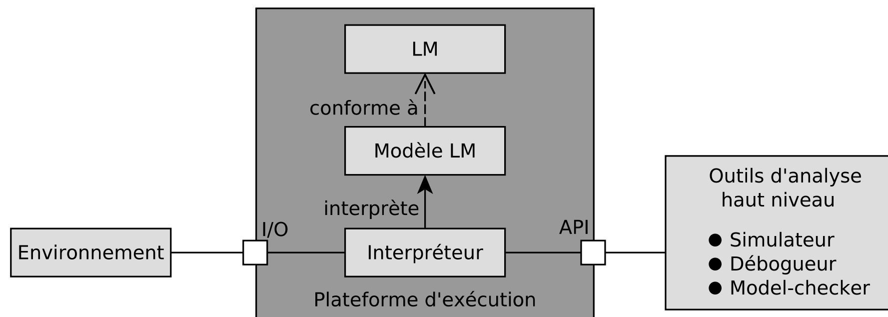



FIGURE 3.7 – Schéma de l'approche X.

continuum, il devient possible de déployer un modèle sur une plateforme d'exécution embarquée durant les phases amont de conception mais aussi d'appliquer des outils d'analyse (p. ex. simulateur, débogueur) sur un modèle déployé sur une plateforme embarquée. Ce continuum permet ainsi d'analyser l'exécution du modèle sur la plateforme réelle à un plus haut niveau d'abstraction.

Grâce à ce cycle de développement, on constate que la conception et la mise au point du système se font toujours dans le même environnement de modélisation. Cette perspective fait sens pour un industriel puisque ses ingénieurs n'auront besoin de connaître qu'un seul formalisme de modélisation. Tous les outils de développement nécessaires à l'analyse du système pourront ensuite venir se connecter à l'interpréteur pour piloter l'exécution du modèle.

Avec cette architecture, le même couple (modèle de conception + sémantique) est utilisé pour toutes les activités de développement du système (c.-à-d. la simulation, la vérification et l'exécution embarquée). En évitant l'utilisation de transformations, cette approche préserve l'unique implémentation de la sémantique du langage de modélisation et apporte des solutions aux problèmes P#1, P#2 et P#3.

**Solution à P#1 (fossé sémantique avec les modèles d'analyse).** L'architecture proposée évite le fossé sémantique P#1 en appliquant les activités d'analyse directement sur le modèle de conception chargé dans l'interpréteur de modèles. Les résultats des activités d'analyse sont donc exprimés en termes des concepts du langage de modélisation facilitant ainsi leur compréhension par les ingénieurs.

**Solution à P#2 (fossé sémantique avec le code exécutable).** Cette approche se soustrait également au fossé sémantique P#2 en déployant et en exécutant directement le modèle de conception sur la plateforme d'exécution réelle. Les concepts utilisés à l'exécution sont donc directement ceux du langage de conception.

**Solution à P#3 (problème d'équivalence).** L'approche X répond au problème d'équivalence P#3 en assurant que ce qui est exécuté est bien ce qui a été vérifié. Le même modèle et la même implémentation de la sémantique sont utilisés en phase de vérification et en phase

d'exécution réelle. De plus, le modèle est chargé une seule et unique fois dans l'interpréteur de modèles évitant ainsi de potentielles dissemblances entre de multiples chargements (cf. section 3.3.2). Avec cette approche, le modèle chargé dans la mémoire de l'interpréteur est le modèle de *référence*. C'est celui qui fait foi pour toutes les activités de développement logiciel.

## 3.12 Objectifs de recherche

Pour mettre en œuvre l'approche X, il faut maintenant identifier plus précisément les exigences auxquelles elle doit répondre afin de définir les objectifs de recherche de cette thèse. Ces exigences peuvent être regroupées en trois groupes : *(1)* langages et interprétation, *(2)* interface, et *(3)* outils d'analyse. Pour chacun de ces groupes, les différentes exigences ont permis d'identifier un objectif de recherche.

**Langages et interprétation.** L'architecture proposée devrait permettre de :

- Unifier l'analyse et l'exécution réelle de modèles en utilisant un même interpréteur pour toutes ces activités ;
- Rendre l'interpréteur pilotable tel que l'exécution de modèles puisse être contrôlée pas à pas (p. ex. par des outils d'analyse) tout en respectant la sémantique du langage de modélisation (notamment l'atomicité des primitives du langage) ;
- Ordonnancer l'exécution pour résoudre le non-déterminisme sur la plateforme d'exécution réelle ;
- Obtenir des performances d'exécution acceptables tout en offrant des services pour l'analyse de modèles. (exigence secondaire)

L'objectif de recherche **O#1** est d'**Identifier les particularités d'un interpréteur permettant d'unifier l'exécution et l'analyse de modèles.**

**Interface.** L'API entre l'interpréteur et les outils d'analyse devrait permettre de :

- Piloter l'exécution du modèle pour parcourir différentes traces d'exécution ;
- Observer l'exécution (p. ex. en évaluant des expressions) afin de fournir aux outils d'analyse les informations dont ils ont besoin ;
- Visualiser l'exécution pour fournir une vue de l'exécution à l'utilisateur ;
- Combler le manque d'outils d'analyse au niveau du langage de modélisation étant donné qu'il n'est plus possible d'utiliser des transformations pour utiliser les outils d'analyse appartenant à d'autres espaces techniques.

L'objectif de recherche **O#2** est de **Définir l'interface nécessaire pour faire interagir des outils d'analyse avec un interpréteur de modèles pilotable.**

**Outils d'analyse.** L'architecture proposée devrait permettre de :
- Utiliser des concepts de modélisation proches de l'utilisateur pour faciliter l'expression des propriétés formelles et la compréhension des résultats d'analyse ;
- Rendre les outils d'analyse indépendants de la sémantique du langage pour gagner en généricité ;
- Rendre les outils plus modulaires pour qu'ils soient davantage configurables et adaptables aux besoins de chaque système.

L'objectif de recherche **O#3** est de **Rendre les outils d'analyse plus génériques, plus modulaires et plus facilement utilisables par les ingénieurs.**

## 3.13 Synthèse

Dans le cadre de ce projet de thèse, nous souhaitons définir des méthodes permettant de concevoir, d'analyser et d'exécuter des systèmes logiciels complexes. Pour y parvenir, cet état de l'art a débuté par l'étude des langages de modélisation afin de définir des modèles de ces systèmes. Pour les exécuter, la sémantique de ces langages est capturée dans des moteurs d'exécution (interpréteurs ou générateurs de code). Pour les analyser, différentes techniques d'analyse dynamique (p. ex. simulation, débogage, *model-checking*) peuvent être appliquées. En pratique, quatre grandes approches d'analyse (par raffinement, avec une transformation vers un langage d'analyse, dédiée, par API) sont utilisées pour vérifier que le comportement du système modélisé satisfait ses exigences.

Dans ce chapitre, nous avons vu comment ces différentes approches d'analyse et d'exécution de modèles peuvent être combinées afin de développer des systèmes logiciels. Les principes de fonctionnement des six approches les plus répandues pour analyser et exécuter des modèles ont été décrits. L'étude de ces différentes approches a permis d'identifier trois problèmes majeurs : P#1 un fossé sémantique entre le modèle de conception et les modèles d'analyse, P#2 un fossé sémantique entre le modèle de conception et le code exécutable, et P#3 un problème d'équivalence entre ce qui est exécuté et ce qui est vérifié.

La **cause principale** de ces trois problèmes est l'**utilisation de multiples implémentations de la sémantique du langage de conception**. En effet, plusieurs outils (p. ex. outils de transformation de modèles, générateurs de code) capturent la sémantique de ce langage afin de traduire le modèle de conception vers différents formalismes (p. ex. langage formel, code exécutable). L'utilisation de transformations de modèles conduit donc à des implémentations plurielles de la sémantique du langage de modélisation. Ces transformations créent ainsi plusieurs instances du même modèle : des modèles d'analyse pouvant être exploités par des outils de V&V et un modèle exécutable pouvant être déployé sur une cible d'exécution réelle.

Pour résoudre ces trois problèmes, cet état de l'art a identifié une approche sans transfor-

mation appelée approche X permettant d'utiliser le même modèle et la même implémentation de la sémantique du langage de modélisation pour l'analyse et l'exécution réelle. L'approche X correspond en fait à l'approche EMI développée dans cette thèse. Pour appliquer cette approche, différents objectifs de recherche ont été identifiés. Afin de tenter de répondre à ces objectifs, la partie suivante décrit les travaux menés pendant cette thèse autour de l'approche EMI.

Deuxième partie

# L’approche EMI

CHAPITRE 4

# PRÉSENTATION DE L'APPROCHE EMI

**Sommaire**


4.1 **Introduction** . . . . . . . . 72
4.2 **Description de l'architecture candidate** . . . . . . . . 73
4.3 **Un contrôleur d'exécution pour chaque activité** . . . . . . . . 73
4.4 **L'interface STR** . . . . . . . . 74
4.5 **Analyse de modèles avec un interpréteur EMI** . . . . . . . . 75
4.6 **Exécution embarquée de modèles avec un interpréteur EMI** . . . . . . . . 77
4.7 **Présentation des différents modes d'exécution** . . . . . . . . 78
4.8 **Synthèse** . . . . . . . . 78


## 4.1 Introduction

Pour traiter les différents problèmes identifiés dans l'état de l'art, cette thèse présente l'approche EMI (pour *Embedded Model Interpreter*) permettant de fédérer la conception, l'exécution et l'analyse des modèles conformes à un langage. Cette approche s'appuie sur une architecture logicielle s'articulant autour d'un interpréteur de modèles qui capture la sémantique du langage de modélisation. Pour mener des activités d'analyse, une interface de contrôle d'exécution permet à de multiples outils (p. ex. , simulateurs, débogueurs, *model-checkers*) de piloter et d'observer l'exécution d'un modèle sur l'interpréteur. L'approche EMI permet de réutiliser la sémantique implémentée dans l'interpréteur pour l'analyse de modèles et ainsi d'unifier les différentes activités de développement logiciel.

Ce chapitre vise à donner un aperçu global de l'approche EMI et de l'architecture logicielle proposée pour l'analyse et l'exécution de modèles. Cette architecture est qualifiée d'architecture "candidate" car elle permet théoriquement de répondre aux besoins énoncés mais son efficacité reste encore à évaluer d'un point de vue pratique. La section 4.2 donne une description de l'architecture candidate puis la section 4.3 présente la notion de contrôleur d'exécution. L'interface de contrôle d'exécution se compose notamment de l'interface STR présentée en section 4.4. La mise en œuvre de cette architecture est décrite en section 4.5 pour l'analyse de

modèles et en section 4.6 pour l'exécution réelle. Enfin, une présentation des caractéristiques d'un interpréteur EMI est réalisée pour chaque mode d'exécution en section 4.7.

## 4.2 Description de l'architecture candidate

L'architecture candidate étudiée dans cette thèse est présentée d'un point de vue conceptuel sur la Figure 4.1. Cette architecture fait apparaître deux composantes du langage de modélisation indispensables à l'exécution des modèles qui s'y conforment : sa syntaxe abstraite modélisée par un *Métamodèle* et sa *Sémantique* capturée par un *Interpréteur*. Le *Modèle* de conception du système étudié se conforme au *Métamodèle* du langage de modélisation. Précédemment, dans le chapitre 3, ce modèle était dit conforme au langage de modélisation mais cette relation est désormais raffinée en se plaçant dans le cadre de l'IDM. Ce *Modèle* est exécuté par un *Interpréteur* de modèles qui encode une *unique* implémentation de la *Sémantique* de ce langage. Cet *Interpréteur* est un moteur d'exécution pouvant interagir avec l'*Environnement* du système (pour l'exécution réelle) ou une abstraction de celui-ci (pour les activités d'analyse). Il est contrôlé par un *Contrôleur d'exécution* à travers une interface de contrôle d'exécution (*CE*). Le *Contrôleur d'exécution* permet de piloter l'exécution du modèle en choisissant les pas d'exécution à effectuer.

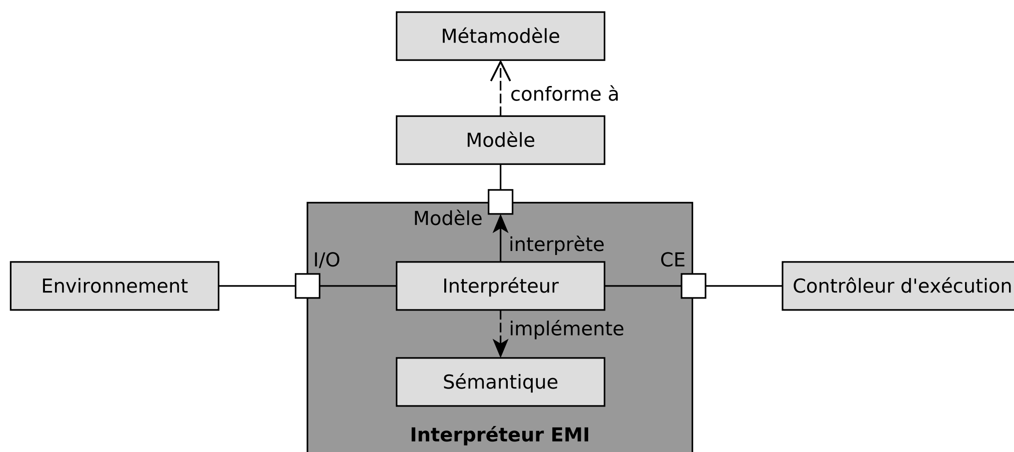



FIGURE 4.1 – Schéma conceptuel de l'architecture candidate pour l'approche EMI.

## 4.3 Un contrôleur d'exécution pour chaque activité

Le contrôleur d'exécution est ce qui donne toute son originalité à cette approche. Ce composant interchangeable permet d'utiliser l'unique implémentation de la sémantique du langage

pour toutes les activités de développement logiciel. Il suffit de choisir un contrôleur d'exécution approprié pour chaque activité afin de répondre aux exigences et aux besoins de chacune d'entre elles :

— Pour la simulation interactive ou l'animation, le rôle du contrôleur d'exécution est joué par l'utilisateur. Il choisit les pas d'exécution à effectuer selon ses propres critères de sélection afin d'explorer différentes traces d'exécution ;

— Pour le *model-checking*, c'est un algorithme d'exploration de l'espace d'état (p. ex. l'algorithme *Depth First Search* (DFS)) qui contrôle l'exécution du modèle. L'automatisation de l'exploration exhaustive est primordiale en *model-checking* afin de pouvoir vérifier, le plus rapidement possible, des propriétés sur des systèmes complexes ;

— Pour l'exécution réelle sur une cible embarquée, le contrôleur d'exécution correspond à la boucle d'exécution principale de la plateforme embarquée. Cette boucle est en charge d'ordonnancer l'exécution du système pour parcourir une trace d'exécution.

Cette architecture procure donc la possibilité de connecter différents types de contrôleurs d'exécution à un interpréteur de modèles. Elle offre ainsi un environnement d'exécution unifié pour l'analyse et l'exécution embarquée de modèles.

L'interface de contrôle d'exécution est un des éléments essentiels de l'approche EMI. Elle doit imposer un minimum de contraintes sur les outils d'analyse et sur l'interpréteur de modèles pour s'adapter au mieux aux besoins de chaque activité de développement logiciel. Cette interface doit être générique afin d'être indépendante du langage et de la façon dont la sémantique est implémentée. L'objectif est de rendre les outils d'analyse plus modulaires afin de les appliquer, à moindre coût, sur des modèles de différents langages.

## 4.4 L'interface STR

Le contrôle de l'exécution est réalisé au moyen de différentes interfaces dont la principale est l'interface *Semantic Transition Relation* (STR) utilisée dans notre approche pour piloter l'exécution du modèle. Cette interface permet d'abstraire la relation de transition de l'exécution d'un modèle en faisant apparaître explicitement la notion de pas d'exécution. Pour définir formellement cette interface, deux types sont nécessaires :

— `C` : le type des configurations. Une configuration correspond à l'ensemble des données dynamiques gérées par un moteur d'exécution (c.-à-d. son état d'exécution) à un instant donné ;

— `A` : le type des actions. Une action est une représentation abstraite d'une transition ou d'un pas d'exécution.

Dans ce manuscrit, toutes les définitions formelles sont données en utilisant le langage Lean [1]

1. Lean : `https://leanprover.github.io/`.

[Mou+15]. Ce langage formel a été privilégié à une notation mathématique car il offre une notation précise et non ambiguë dont le typage peut être vérifié par l'assistant de preuve du même nom. C'est donc le langage Lean qui est utilisé en Listing 4.1 pour définir l'interface STR.

**Définition 4.1.** L'interface STR utilisée pour contrôler l'exécution de modèles est définie avec les trois fonctions suivantes :

```
structure STR (C A : Type) :=
  (initial : set C)
  (actions : C → set A)
  (execute : C → A → set C)
```

Listing 4.1 – Interface STR.

La fonction `initial` retourne toutes les configurations initiales possibles (`set C`) de l'exécution du modèle. La fonction `actions` retourne toutes les actions (`set A`) qui sont exécutables à partir d'une configuration donnée (`C`). Elle expose le non-déterminisme du modèle exécuté car les actions retournées appartiennent à différents processus qui sont concurrents pour leur exécution. La fonction `execute` exécute une action (`A`) dans une configuration source donnée (`C`) et récupère toutes les configurations cibles (`set C`) qui sont obtenues. Même si l'exécution est typiquement déterministe (c.-à-d. seulement une configuration est retournée à un instant donné), l'abstraction fournie par l'interface STR permet de considérer des exécutions non-déterministes. Dans ce cas, il existe potentiellement plusieurs configurations initiales ainsi que plusieurs configurations cibles pour un même pas d'exécution. L'utilisation d'un ensemble de configurations (`set C`) pour `initial` et `execute` permet à l'interface STR de prendre en compte ce type d'exécution, ce qui est parfois utile pour des langages de spécification de haut niveau.

Avec ces trois fonctions, l'interface STR offre ainsi un point de vue sur l'exécution du modèle et plus globalement sur la sémantique du langage de modélisation utilisé pour décrire ce modèle. Ce point de vue est exploité par le contrôleur d'exécution notamment pour mener diverses activités d'analyse.

## 4.5 Analyse de modèles avec un interpréteur EMI

Pour mener des activités d'analyse (Figure 4.2), un *Outil d'analyse* (p. ex. un simulateur, un débogueur, un *model-checker*) est utilisé pour piloter l'exécution du *Modèle*. Cet outil est branché sur le même interpréteur de modèles que celui utilisé lors de l'exécution réelle. Cette approche permet ainsi d'analyser le comportement du modèle exécutable chargé dans l'interpréteur.

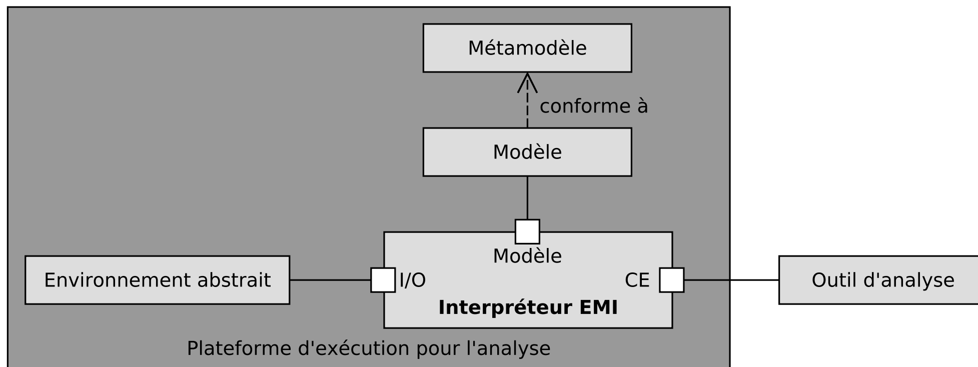



FIGURE 4.2 – Mise en œuvre de l'architecture candidate pour l'analyse de modèles.

Pour effectuer ces activités d'analyse, le *Modèle* du système exécuté par l'interpréteur doit être fermé par une abstraction de l'environnement réel du système. Cette abstraction de l'environnement, appelée *Environnement abstrait*, est également représentée sous forme d'un modèle (au sens de l'IDM) qui modélise les interactions de l'environnement avec le système. Pour réaliser cette abstraction, il est souvent nécessaire de prendre en compte certaines hypothèses afin de simplifier la modélisation et d'éviter l'explosion combinatoire. Ce modèle peut être spécifié avec le même langage de modélisation que le système et peut, dans ce cas, être exécuté par le même interpréteur.

Pour certaines activités d'analyse comme le *model-checking* ou le débogage, il est non seulement utile de contrôler l'exécution du modèle mais il faut aussi pouvoir observer son exécution. L'approche EMI offre pour cela un langage d'observation permettant de spécifier des prédicats, appelés propositions atomiques, qui portent sur l'exécution du système. Ces prédicats peuvent être exprimés directement en termes des concepts du langage de modélisation ce qui facilite le travail des ingénieurs et rend davantage accessible l'utilisation des outils de vérification formelle. Lors de la phase de V&V, l'évaluation de ces prédicats est déléguée à l'interpréteur de modèles qui possède une fonction d'évaluation dédiée.

L'approche EMI fournit également divers opérateurs permettant d'adapter l'architecture logicielle aux besoins de l'activité d'analyse menée. Parmi ceux-ci, un opérateur d'ordonnancement permet de prendre en compte l'ordonnanceur dans la boucle d'analyse pour résoudre le non-déterminisme du système. Pour la vérification formelle, un opérateur de composition synchrone permet de composer l'exécution du modèle avec un automate représentant une propriété formelle. Grâce à cet opérateur, il est possible d'appliquer des algorithmes de *model-checking* mais aussi d'effectuer du *monitoring* afin de poursuivre la vérification formelle à l'exécution. Le *monitoring* est complémentaire au *model-checking* pour la vérification de propriétés monitorables mais aussi pour vérifier que les hypothèses considérées pour réaliser l'abstrac-

tion de l'environnement sont bien satisfaites à l'exécution. La mise en œuvre de ces différentes activités d'analyse sera présentée plus en détail dans les chapitres suivants tout comme le fonctionnement des opérateurs mentionnés.

## 4.6 Exécution embarquée de modèles avec un interpréteur EMI

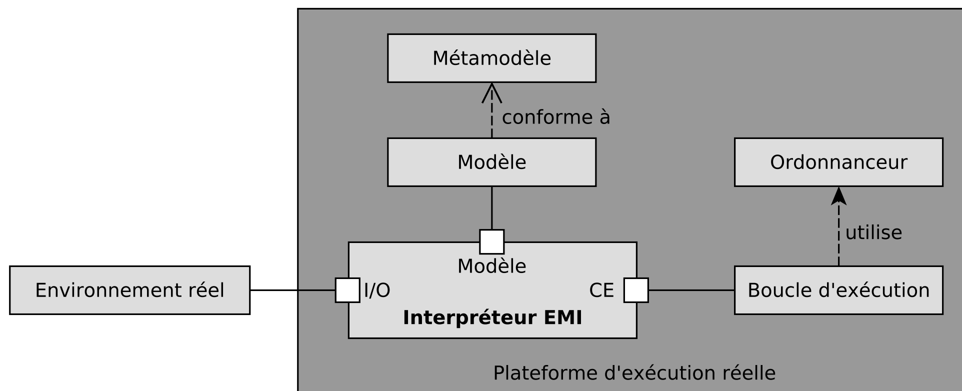



FIGURE 4.3 – Mise en œuvre de l'architecture candidate pour l'exécution embarquée de modèles.

Pour l'exécution réelle (ou exécution embarquée) en Figure 4.3, le *Modèle* du système est déployé sur une *Plateforme d'exécution* embarquée avec l'interpréteur de modèles qui implémente la sémantique opérationnelle du langage de modélisation. L'exécution du système interagit ici avec l'*Environnement réel* via les entrées/sorties (*I/O*) de la plateforme d'exécution. Les périphériques de la cible d'exécution permettent de récupérer les données des capteurs et de piloter les actionneurs du système réel. Les évènements sortants de l'interpréteur sont ainsi convertis en signaux physiques et les signaux physiques entrants sont convertis en évènements pour l'exécution du système.

Pour l'exécution embarquée, le contrôleur d'exécution est joué par la *Boucle d'exécution* principale de la plateforme d'exécution. Une seule trace d'exécution doit ici être parcourue. Un *Ordonnanceur* est donc utilisé pour choisir le prochain pas d'exécution à réaliser. Cet ordonnanceur apparaît ici explicitement sous forme d'un opérateur afin d'ordonnancer les différents processus du système qui sont concurrents pour leur exécution.

## 4.7 Présentation des différents modes d'exécution

Le Tableau 4.1 présente un récapitulatif des différents modes d'exécution d'un interpréteur EMI. Il donne ainsi un aperçu des différentes caractéristiques de l'interpréteur pour chacune des activités de développement mises en œuvre dans ces travaux de doctorat c.-à-d. la simulation interactive, l'animation, la bisimulation, le débogage multivers, la détection de *deadlocks*, le *model-checking*, le *monitoring* et l'exécution réelle. Pour chacune d'entre elles, ce tableau indique si l'environnement réel ou une abstraction est utilisé, si l'activité peut être menée sur une plateforme de développement ou une cible embarquée, et quel est le type du contrôleur d'exécution. Il indique également si l'activité peut être réalisée en prenant en compte l'ordonnanceur ou si le non-déterminisme de l'exécution du système est exposé au contrôleur d'exécution. Ce tableau précise si l'activité en question a besoin d'évaluer des propositions atomiques (ou atomes) et si l'exécution du système doit être composée avec un automate représentant une propriété formelle à l'aide d'un opérateur de composition synchrone. Enfin, pour chaque activité de développement, ce tableau fait référence aux principales sections de ce manuscrit en charge de présenter l'activité ou d'illustrer sa mise en œuvre via des expérimentations.

| Modes | Simulation interactive | Animation | Bisimulation | Débogage multivers |
|---|---|---|---|---|
| Environnement | réel/abstrait | réel/abstrait | abstrait | abstrait |
| Plateforme d'exécution | développement/embarquée | développement/embarquée | développement | développement/embarquée |
| Contrôleur d'exécution | utilisateur | utilisateur | algo d'atteignabilité | algo d'atteignabilité |
| Ordonnanceur | avec/sans | avec/sans | avec/sans | avec/sans |
| Évaluation d'atomes | non | non | non | oui |
| Composition synchrone | non | non | non | non |
| Sections concernées | 5.3.1, 10.4.1 | 10.5.2 | 10.5.2 | 5.3.2, 10.4.2 |

| Modes | Détection de deadlocks | Model-checking | Monitoring | Exécution réelle |
|---|---|---|---|---|
| Environnement | abstrait | abstrait | réel/abstrait | réel |
| Plateforme d'exécution | développement | développement | développement | embarquée |
| Contrôleur d'exécution | algo d'atteignabilité | algo de model-checking [GS09] | boucle d'exécution | boucle d'exécution |
| Ordonnanceur | avec/sans | avec/sans | avec | avec |
| Évaluation d'atomes | non | oui | oui | non |
| Composition synchrone | non | oui | oui | non |
| Sections concernées | 5.3.3, 10.4.3 | 5.4.4, 10.4.3, 10.4.5.2 | 5.4.5, 10.4.7 | 8.6, 10.4.7 |

TABLE 4.1 – Présentation des différents modes d'exécution.

## 4.8 Synthèse

En résumé, l'approche EMI permet d'unifier l'analyse et l'exécution embarquée de modèles en reposant sur une unique implémentation de la sémantique d'exécution. La sémantique du langage de modélisation est capturée sous forme opérationnelle dans un interpréteur de modèles pouvant être piloté par différents contrôleurs d'exécution. Cet interpréteur s'inscrit dans une architecture logicielle modulaire et configurable permettant de s'adapter aux besoins des différentes activités de développement logiciel.

Cette approche sans transformations (c.-à-d. sans transformations de modèles, sans génération de code) permet d'utiliser le même couple (modèle + sémantique du langage) pour toutes les activités de développement logiciel. Elle évite ainsi les fossés sémantiques et répond au problème d'équivalence en exécutant et en vérifiant directement le modèle de conception. Les chapitres suivants vont maintenant détailler les différentes contributions scientifiques au cœur de l'approche EMI.

CHAPITRE 5

# ANALYSE DE L'EXÉCUTION DE MODÈLES

## Sommaire


**5.1 Introduction** . . . . . . . . **80**
**5.2 Abstraction de l'environnement** . . . . . . . . **81**
**5.3 Simulation, débogage, et détection de *deadlocks*** . . . . . . . . **82**
5.3.1 Simulation interactive . . . . . . . . 82
5.3.2 Débogage multivers . . . . . . . . 83
5.3.3 Détection de *deadlocks* . . . . . . . . 87
**5.4 Architecture de vérification formelle** . . . . . . . . **87**
5.4.1 Automates de Büchi et automates observateurs . . . . . . . . 88
5.4.2 Les interfaces APC et $A_{cc}$ . . . . . . . . 88
5.4.3 Description de l'architecture conceptuelle . . . . . . . . 89
5.4.4 Architecture conceptuelle pour le *model-checking* . . . . . . . . 90
5.4.5 Architecture conceptuelle pour le *monitoring* . . . . . . . . 92
**5.5 Composition synchrone** . . . . . . . . **93**
5.5.1 Description théorique . . . . . . . . 93
5.5.2 Optimisation pour le *monitoring* . . . . . . . . 95
**5.6 *Model-checking* avec des propriétés LTL** . . . . . . . . **96**
5.6.1 Spécification de propriétés LTL . . . . . . . . 96
5.6.2 Architecture logicielle pour la vérification LTL . . . . . . . . 97
**5.7 Synthèse** . . . . . . . . **99**


*Ce chapitre est partiellement adapté de nos contributions en [Bes+18c ; Bes+19d].*

## 5.1 Introduction

Avec l'approche EMI, les outils d'analyse peuvent se connecter à l'interpréteur de modèles afin de piloter et d'observer l'exécution du système. Pour mener ces activités d'analyse, une abstraction de l'environnement réel doit être réalisée afin de modéliser les interactions entre le système et son environnement (section 5.2). Avec ce dispositif, diverses activités d'analyse (p.

ex. simulation, débogage, détection de *deadlocks* en section 5.3) peuvent être réalisées dès les phases de conception amont pour aider les ingénieurs à modéliser le système.

Des techniques de vérification formelle (p. ex. *model-checking*, *monitoring* en section 5.4) permettent ensuite de vérifier des propriétés formelles sur l'exécution du système afin d'assurer que son comportement est conforme aux spécifications. Ces techniques reposent sur l'utilisation d'un opérateur de composition synchrone (section 5.5) permettant à l'automate modélisant la propriété formelle de suivre l'exécution du système. Cette architecture de vérification formelle peut notamment être utilisée pour mettre en œuvre des algorithmes de *model-checking* à partir de propriétés LTL (section 5.6).

## 5.2 Abstraction de l'environnement

Avant de pouvoir mener des activités d'analyse, il est nécessaire de fermer l'exécution du système avec une abstraction de l'environnement réel. Cette abstraction modélise les interactions entre l'environnement et le système afin de pouvoir solliciter l'exécution du système. En IDM, cette abstraction est également définie sous forme d'un modèle. Dans notre approche, le même langage de modélisation que celui ayant servi pour le système est utilisé pour décrire le modèle abstrait de l'environnement. Ainsi, ce modèle peut être exécuté avec le même interpréteur que le modèle du système.

L'élaboration du modèle de l'environnement est une étape primordiale pour pouvoir effectuer des activités d'analyse et notamment du *model-checking*. Plus ce modèle est une représentation fidèle de l'environnement réel, plus les activités d'analyse seront efficaces et pourront garantir le bon fonctionnement du système. Il est donc important de bien comprendre l'environnement réel dans lequel le système est plongé pour en définir une abstraction conforme.

En pratique, il est courant de modéliser différentes versions (plus ou moins raffinées) de l'environnement selon les besoins de chaque activité d'analyse. *(i)* Pour la simulation interactive, il est possible de définir un modèle très générique composé seulement d'un automate pâquerette permettant d'envoyer ou de recevoir n'importe quel évènement à chaque pas d'exécution. Ce modèle très abstrait considère ainsi tous les entrelacements possibles d'évènements. *(ii)* Pour le *model-checking*, des raffinements doivent généralement être effectués pour définir plus finement les interactions de l'environnement avec le système. En effet, pour simplifier la modélisation et éviter une explosion combinatoire de l'espace d'état, il est souvent nécessaire de prendre en compte certaines hypothèses par rapport au comportement de l'environnement réel. Ces hypothèses permettent de réduire les entrelacements d'évènements en éliminant certains chemins d'exécution qui sont impossibles dans le monde réel. Le modèle résultant doit en théorie toujours couvrir un sur-ensemble de l'ensemble des scénarios possibles afin de garantir une couverture complète de l'exécution du système. En pratique, si le système réel est sollicité

avec une séquence d'évènements non prévue par le modèle abstrait utilisé en phase de V&V, alors un risque de défaillance est possible. C'est la raison pour laquelle le *model-checking* ne garantit le comportement du système que par rapport à un modèle d'environnement donné. Le risque de défaillance n'est donc pas nul même avec une technique de vérification formelle exhaustive.

Par conséquent, il est essentiel de s'assurer que l'abstraction de l'environnement prise en compte en phase d'analyse est bien conforme à l'environnement réel du système. C'est la raison pour laquelle cette thèse considère l'hypothèse de travail suivante : "À partir de la documentation du système, les ingénieurs sont capables de produire un modèle d'environnement exhaustif (c.-à-d. qui prend en compte tous les scénarios possibles) et conforme à l'environnement réel". Cette hypothèse est particulièrement réaliste dans le cas des systèmes embarqués car ce type de systèmes possède souvent des fonctionnalités critiques qui sont rigoureusement spécifiées et documentées. En particulier, les interactions du système avec son environnement sont souvent bien décrites et caractérisées dans les documents de spécification, ce qui permet de capturer précisément le comportement de l'environnement.

De plus, pour prendre en compte les limitations identifiées et les hypothèses considérées pour modéliser l'environnement, il semble intéressant de combiner l'usage du *model-checking* avec une solution de *monitoring*. Celle-ci permettra d'une part de continuer la vérification des propriétés formelles à l'exécution et d'autre part de vérifier que la trace d'exécution courante ne sort pas de l'espace d'état considéré lors de la phase de V&V.

## 5.3 Simulation, débogage, et détection de *deadlocks*

Pour aider à la mise au point du modèle du système, diverses activités d'analyse peuvent être menées durant les phases de conception amont dont la simulation interactive, le débogage multivers et la détection de *deadlocks*. Ces activités permettent en général de détecter des erreurs de conception et de mettre en évidence certaines incohérences lors de l'exécution du modèle.

### 5.3.1 Simulation interactive

La simulation interactive permet d'explorer différentes traces d'exécution et de visualiser l'évolution des données d'exécution. Pour y parvenir, il est nécessaire de pouvoir piloter l'exécution du modèle en demandant dans chaque configuration la liste des pas d'exécution disponibles. Cette liste est ensuite exposée à l'utilisateur qui choisit le pas d'exécution devant être exécuté. Avec la fonctionnalité de retour en arrière (ou en anglais *back-in-time*), il est également possible de replacer l'interpréteur dans une configuration donnée et de reprendre

l'exécution à partir de ce point. Ce mécanisme permet de facilement revenir en arrière sans avoir à redémarrer le moteur d'exécution.

En ce qui concerne la visualisation, il est nécessaire de construire une projection de la configuration et des pas d'exécution disponibles sur l'interface graphique. Dans ce but, une nouvelle interface P a été définie en Lean. Elle se base sur deux nouveaux types : `Vc` la vue d'une configuration et `Va` la vue d'une action.

**Définition 5.1.** L'interface P (*Projection*) permet d'obtenir une projection des configurations et des actions dans les termes du langage de modélisation. Elle est définie par :

```
structure P (C A : Type) :=
  (projectC : C → Vc)
  (projectA : A → Va)
```

Listing 5.1 – Interface P.

Dans cette interface P, **`projectC`** permet de projeter une configuration `C` sur une vue `Vc` et **`projectA`** permet de projeter une action `A` sur une vue `Va` avec `Vc` et `Va` des vues de l'interface graphique de l'outil d'analyse. Étant donné que l'approche EMI vise la conception d'un interpréteur embarqué, la construction de ces vues doit être faite hors de l'interpréteur afin de ne pas accroître l'empreinte mémoire ou d'impacter les performances d'exécution. Dans notre approche, cette tâche est réalisée par un serveur de langages qui servira d'intermédiaire entre l'outil d'analyse et l'interpréteur de modèles.

### 5.3.2 Débogage multivers

Le débogage multivers a été introduit par [Tor+19] comme la possibilité de définir des points d'arrêts qui peuvent mettre en pause l'exécution du système dans différents chemins d'exécution, appelés univers. Cette fonctionnalité est intéressante pour les interpréteurs EMI afin de permettre le débogage de modèles ayant des exécutions concurrentes.

Pour mener cette activité, des points d'arrêts conditionnels doivent être définis. Les conditions de ces points d'arrêt sont définies comme des prédicats portant sur l'exécution du système. Un langage, que nous appellerons langage d'observation, est alors nécessaire pour pouvoir spécifier ces prédicats aussi appelés propositions atomiques. Une fois exprimés, ces prédicats sont ensuite évalués dans chaque nouvelle configuration explorée et le résultat de leur évaluation est indiqué à l'utilisateur. L'évaluation de ces expressions nécessite non seulement l'accès aux données d'exécution du modèle mais aussi la connaissance de la sémantique du langage (p. ex. pour appeler une fonction). Pour effectuer cette tâche, une fonction d'évaluation est donc également nécessaire pour déléguer l'évaluation d'une proposition atomique

à l'interpréteur EMI. Cette technique permet ainsi de réutiliser la même sémantique opérationnelle que pour l'exécution de modèles.

Pour définir cette fonction d'évaluation, l'interface de contrôle d'exécution doit être enrichie avec une nouvelle interface appelée APE. Pour cette définition, `L` est le type des étiquettes (en anglais *labels*) aussi appelées propositions atomiques ou atomes.

**Définition 5.2.** L'interface APE (*Atomic Proposition Evaluator*) qui permet d'évaluer une proposition atomique (`L`) sur un pas d'exécution (`C → A → C`) est définie par :

```
structure APE (C A L : Type) :=
  (eval : L → C → A → C → bool)
```

Listing 5.2 – Interface APE

Avec ce mécanisme, le débogage multivers peut facilement être supporté en utilisant un algorithme d'atteignabilité afin d'explorer l'espace d'état jusqu'à ce que l'un des prédicats devienne vrai (ou faux). Tous les chemins issus de la configuration courante sont donc explorés simultanément. Le premier qui atteint une configuration déclenchant l'un des points d'arrêt conditionnels est exposé à l'utilisateur afin qu'il visualise ce qu'il cherche à déboguer. Si les points d'arrêts ne sont jamais déclenchés, il faut attendre l'exploration complète de l'espace d'état pour le notifier à l'utilisateur. En considérant `p` la condition d'un point d'arrêt conditionnel, l'algorithme de débogage multivers est équivalent à vérifier par *model-checking* la propriété LTL `"[] !p"` indiquant que globalement `p` est faux. Dès que `p` devient vrai, un contre-exemple est trouvé et l'état dans lequel se trouve l'exécution du modèle correspond à l'état recherché.

D'un point de vue plus théorique, ces travaux de thèse proposent une formalisation du débogage multivers interactif via les Listings 5.3 et 5.4. Cette définition formelle s'appuie sur une configuration de débogage (`DebugConfig`) et des actions de débogage (`DebugAction`) comme le montre le Listing 5.3. La configuration de débogage correspond aux données d'exécution stockées par le débogueur. Elle contient la configuration courante (`current`), l'ensemble des configurations déjà explorées (`trace`), et l'ensemble des successeurs (`options`) c.-à-d. l'ensemble des configurations pouvant être atteintes à partir de la configuration courante avec un seul pas d'exécution. Grâce à ces informations, l'utilisateur peut réaliser différentes actions de débogage permettant d'exécuter un pas à partir de la configuration courante (`step`), de sélectionner une configuration parmi l'ensemble des successeurs (`select`), de retourner dans une configuration déjà explorée (`jump`) ou de lancer un algorithme de débogage multivers jusqu'à un point d'arrêt (`run_to_breakpoint`). C'est la raison pour laquelle on parle de débogage multivers *interactif* car c'est l'utilisateur qui joue le rôle du contrôleur d'exécution mais il peut déléguer ce rôle à l'algorithme de débogage multivers pour atteindre des points d'arrêt.

```
structure DebugConfig (C : Type) :=
  (current : option C)
  (trace : set C)
  (options : set C)
inductive DebugAction {C A : Type}
  | step : A → DebugAction
  | select : C → DebugAction
  | jump : C → DebugAction
  | run_to_breakpoint : DebugAction
```

Listing 5.3 – Définition des types utilisés en débogage multivers.

À partir de ces définitions formelles, le Listing 5.4 définit un opérateur appelé `interactive_multivers_debugging` permettant de transformer une STR en une STR pour le débogage multivers interactif. Cet opérateur prend en entrée la STR concernée (`str`), la fonction permettant de retrouver la trace du contre-exemple (`find_counter example`) et la liste des points d'arrêt considérés (`breakpoints`).

La STR résultante contient un unique état de débogage `initial` dans lequel l'état courant n'est pas défini (`current := none`), la trace est vide (`trace := ∅`) et l'ensemble des configurations initiales de la STR est placé dans le champ `options` (`options := str.initial`). À partir de cet état initial, les seules `actions` possibles correspondent à la sélection de l'une des configurations initiales de la STR (cas où `dc.current = none`). Si une configuration courante est déjà sélectionnée (cas où `dc.current = (some c)`), diverses actions de débogage sont possibles. La Figure 5.1 illustre ces différentes actions sur un automate très simple ayant sa configuration initiale en noir. À partir de la configuration courante (en gris foncé), il est possible de faire un pas (`oa := {step1, step2}`), de retourner sur l'une des configurations appartenant à la trace en faisant un `jump` (`ja := {jump1, jump2}`) ou de lancer l'algorithme de débogage multivers (`run_to_breakpoint`). En résumé, l'ensemble des `actions` ne correspond plus seulement aux pas exécutables (`step`) mais à l'ensemble des actions pouvant être réalisées par l'utilisateur (`step`, `select`, `jump` et `run_to_breakpoint`). De même, le champ `execute` de la nouvelle STR permet non seulement d'exécuter des pas mais aussi d'effectuer différentes actions de débogage tout en mettant à jour la configuration courante de débogage et la trace. Le Listing 5.4 présente la combinatoire des différents cas possibles selon l'action de débogage devant être exécutée et si la configuration courante de débogage est définie ou non. On peut notamment remarquer que l'exécution de l'algorithme de débogage multivers doit reconstruire la trace du contre-exemple si un point d'arrêt est atteint.

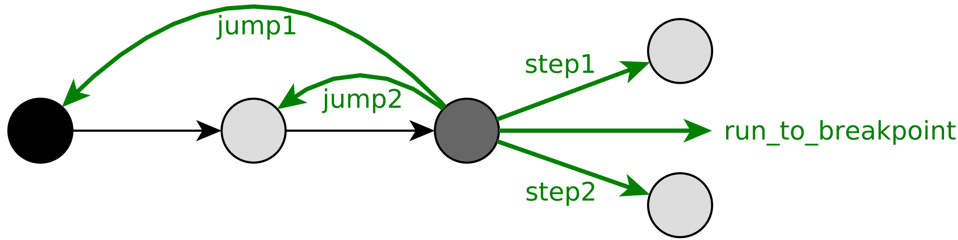



FIGURE 5.1 – Présentation des actions de débogage disponibles.

```
def interactive_multiverse_debugging
    (str : STR C A)
    (find_counterexample : set C → STR C A → set C → list C)
    (breakpoints : set C)
: STR (DebugConfig C) (@DebugAction C A) := {
initial := { { DebugConfig . current := none,
    trace := ∅, options := str.initial } },
actions := λ dc, match dc.current with
    | (some c) :=
        let
            oa := { x : @DebugAction C A |
                ∀ a ∈ str.actions c, x = DebugAction.step a },
            ja := { x : @DebugAction C A |
                ∀ c ∈ dc.trace,     x = DebugAction.jump c }
        in
            oa ∪ ja ∪ { DebugAction.run_to_breakpoint }
    | none := { sa | ∀ c ∈ dc.options, sa = DebugAction.select c }
    end,
execute := λ dc da, match dc.current, da with
    | (some c), (DebugAction.step a) := {{ DebugConfig . current := none,
        trace := dc.trace, options := str.execute c a}}
    | (none)  , (DebugAction.step a) := ∅
    | _       , (DebugAction.select c) := {{ DebugConfig . current := c,
        trace := {c} ∪ dc.trace, options := ∅ }}
    | _       , (DebugAction.jump c) := {{ DebugConfig . current := c,
        trace := {c} ∪ dc.trace, options := ∅ }}
    | (some c), (DebugAction.run_to_breakpoint) :=
        match (find_counterexample { c } str breakpoints) with
        | []        := {{ DebugConfig . current := dc.current,
```

```
            trace := dc.trace, options := ∅ }}
        | a::ctrace := {{ DebugConfig . current := a,
            trace := {a} ∪ { x | x ∈ ctrace } ∪ dc.trace, options := ∅ }}
        end
    | none     , (DebugAction.run_to_breakpoint) :=
        match (find_counterexample dc.options str breakpoints) with
        | []         := { { DebugConfig . current := dc.current,
            trace := dc.trace, options := ∅ } }
        | a::ctrace := { { DebugConfig . current := a,
            trace := {a} ∪ { x | x ∈ ctrace } ∪ dc.trace, options := ∅ } }
        end
    end
}
```

Listing 5.4 – Définition du débogage multivers interactif.

### 5.3.3 Détection de *deadlocks*

La détection de *deadlocks* utilise également un algorithme d'atteignabilité pour rechercher des *deadlocks* en explorant l'espace d'état du modèle. Un *deadlock* est détecté dans une configuration donnée si aucun pas d'exécution ne peut être effectué à partir de cette configuration ou si l'exécution d'un pas donné ne permet pas d'obtenir de configuration cible. L'exécution du système se retrouve donc bloquée. Si un *deadlock* est détecté, le chemin d'exécution menant à l'état d'exécution redouté depuis la configuration initiale est affiché à l'utilisateur afin qu'il visualise le scénario ayant conduit à cette situation. Cet algorithme de détection de *deadlocks* requiert uniquement le pilotage de l'exécution du modèle mais pas l'évaluation de propositions atomiques. Il est équivalent à la propriété LTL suivante : `"[] !|DEADLOCK|"` où `|DEADLOCK|` est la proposition atomique qui s'évalue à $true$ lors de la présence d'un *deadlock*.

## 5.4 Architecture de vérification formelle

En plus des activités de simulation et de débogage, il est possible de mettre en œuvre des activités de vérification formelle en pilotant l'exécution du modèle sur un interpréteur EMI. Cette section présente l'architecture utilisée pour réaliser de la vérification formelle hors-ligne via *model-checking* ou en ligne via *monitoring*.

### 5.4.1 Automates de Büchi et automates observateurs

Pour appliquer cette architecture de vérification formelle, les deux formalismes d'automates utilisés dans cette thèse sont les automates de Büchi et les automates observateurs.

Les automates de Büchi sont couramment utilisés en *model-checking* pour vérifier des propriétés sur l'espace d'état complet d'un modèle [BK08]. Ces automates considèrent des traces infinies et peuvent exprimer n'importe quel $\omega$-langage (p. ex. des propriétés de sûreté ou de vivacité).

Les automates observateurs sont définis en [OGO06] comme des processus spéciaux qui surveillent les changements dans l'état d'un modèle (p. ex. l'état courant d'une machine à états, le contenu des piles d'exécution) et les évènements qui surviennent dans celui-ci (p. ex. des envois de messages ou de signaux). Ces automates opèrent sur des traces finies ce qui leur permet d'être utilisés en *monitoring* mais leur expressivité se limite à des propriétés monitorables [BLS11].

En accord avec la littérature [BK08], nous pouvons donner une définition commune pour ces deux types d'automates.

**Définition 5.3.** Un automate est défini par le tuple $A = (\Sigma, Q, \delta, q_0, F)$ où :
— $\Sigma$ est un ensemble fini d'étiquettes appelé l'alphabet de $A$ ;
— $Q$ est un ensemble fini d'états de $A$ ;
— $\delta \subseteq Q \times \Sigma \times Q$ est la relation de transition de $A$ ;
— $q_0 \in Q$ est l'état initial de $A$ ;
— $F \subseteq Q$ est un ensemble fini d'états d'acceptation de $A$.
La principale différence entre les automates de Büchi et les automates observateurs concerne leur *condition d'acceptation*. Pour les automates de Büchi, un mot (ou une trace) est accepté si un état d'acceptation est atteint *infiniment souvent*, alors que pour les automates observateurs, un mot est accepté dès qu'un état d'acceptation est atteint au moins *une fois*.

L'architecture de vérification formelle présentée dans cette thèse n'est en aucun cas spécifique à ces deux formalismes d'automates. D'autres types d'automates pourraient être utilisés (p. ex. les automates de Büchi généralisés). La principale différence concerne la condition d'acceptation qui est spécifique à chaque formalisme.

### 5.4.2 Les interfaces APC et $A_{cc}$

Pour effectuer des activités de vérification formelle, l'interface de contrôle d'exécution doit être enrichie avec les interfaces APC et $A_{cc}$ en plus des interfaces STR et APE définies précédemment.

**Définition 5.4.** L'interface APC (*Atomic Propositions Collector*) permet de récupérer les propositions atomiques nécessaires à l'évaluation d'une action (A) dans une configuration donnée (C). Elle est définie par :

```
structure APC (C A L : Type) :=
  (eval : C → A → set L)
```

Listing 5.5 – Interface APC.

**Définition 5.5.** L'interface $A_{cc}$ (*Acceptance*) permet de savoir si une configuration donnée (C) est un état d'acceptation. Elle est définie par :

```
structure Acc (C : Type) :=
  (accepting : set C)
```

Listing 5.6 – Interface $A_{cc}$.

### 5.4.3 Description de l'architecture conceptuelle

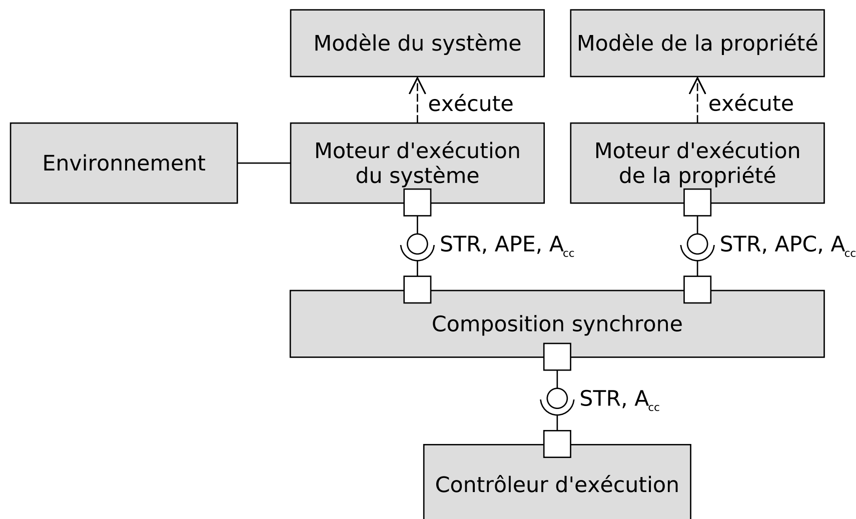



FIGURE 5.2 – Architecture utilisée pour la vérification formelle.

La Figure 5.2 présente, sous forme d'un diagramme de composants UML, l'architecture conceptuelle permettant de vérifier des propriétés formelles sur l'exécution du système. D'un côté, le *Modèle du système* est exécuté par le *Moteur d'exécution du système* (p. ex. un interpréteur EMI) dont la sémantique opérationnelle est celle du langage de modélisation du système. Cet outil interagit avec l'*Environnement* réel dans lequel le système évolue ou une abstraction de celui-ci selon l'activité réalisée. De l'autre, le *Modèle de la propriété* est interprété

par le *Moteur d'exécution de la propriété* qui est spécifique au formalisme utilisé pour la spécification de propriétés. Quel que soit le langage utilisé, les propriétés formelles peuvent toujours s'exprimer sous forme d'automates afin d'appliquer des techniques de *model-checking*. Divers formalismes d'automates peuvent être utilisés dont ceux des automates de Büchi et des automates observateurs qui font partie des plus répandus dans la littérature [Hol97 ; Dha+12 ; DP04 ; Bar+17 ; OGO06].

Chacun des moteurs d'exécution implémente l'interface STR ce qui permet de piloter l'exécution des modèles du système et de la propriété. Le pilotage est réalisé par notre opérateur de composition synchrone dont le rôle est de synchroniser l'exécution du modèle de la propriété avec l'exécution du système. L'automate de la propriété peut ainsi suivre l'exécution du système afin de détecter des défaillances. La composition synchrone repose sur l'utilisation de propositions atomiques. Celles-ci sont spécifiées dans le modèle de la propriété et portent sur l'exécution du système. À chaque pas d'exécution du système, l'opérateur de composition synchrone doit déterminer quels sont les pas de l'automate de la propriété qui doivent être synchronisés avec celui-ci. Pour cela, il récupère les propositions atomiques nécessaires à l'évaluation des pas du modèle de la propriété via l'interface APC puis il délègue l'évaluation de ces propositions atomiques au *Moteur d'exécution du système* via l'interface APE. Le vecteur de booléens résultant de l'évaluation de ces atomes est ensuite renvoyé au *Moteur d'exécution de la propriété*. Ce vecteur donne une vue Kripke sur l'exécution du système (c.-à-d. sous forme d'une structure de Kripke [Kri63]) permettant au *Moteur d'exécution de la propriété* de choisir les pas d'exécution de la propriété qui doivent être synchronisés avec le pas d'exécution du système. L'opérateur de composition synchrone permet ainsi de construire le produit synchrone de l'exécution de la propriété avec l'exécution du système. Ce produit synchrone est également un automate dont l'exécution est exposée par l'opérateur de composition synchrone sous forme d'une STR.

La STR résultante est pilotée par un *Contrôleur d'exécution* afin d'appliquer des algorithmes de vérification formelle. Le *Contrôleur d'exécution* diffère suivant le type d'activité réalisée (c.-à-d. *model-checking* ou *monitoring*). Celui-ci peut également connaitre les états d'acceptation du produit synchrone à travers l'interface $A_{cc}$ fournie par l'opérateur de composition synchrone.

### 5.4.4 Architecture conceptuelle pour le *model-checking*

La Figure 5.3 présente l'application de l'architecture conceptuelle de vérification formelle au *model-checking*. Le contrôleur d'exécution est ici remplacé par un *Algorithme de model-checking* qui donne en sortie des *Résultats de vérification* par rapport à l'*Environnement abstrait* considéré. Cet algorithme permet de vérifier que le produit synchrone de l'exécution du système et de la propriété est vide c.-à-d. qu'aucun des comportements indésirables encodés par la propriété n'est contenu dans ceux du système. Le choix de cet algorithme dépend du

formalisme d'automates choisi et du type de propriétés vérifiées (p. ex. sûreté, vivacité). Il peut par exemple s'agir d'un algorithme d'atteignabilité (p. ex. pour les automates observateurs) ou encore d'un algorithme de détection de boucle d'acceptabilité (p. ex. pour les automates de Büchi). Chacun de ces algorithmes doit également stocker l'ensemble des configurations déjà explorées pour assurer la terminaison c.-à-d. pour savoir à quel instant l'espace d'état du produit synchrone a été complètement exploré.

Pour réaliser l'exploration de l'espace d'état, l'*Algorithme de model-checking* utilise le composant *STR2TR* pour se connecter à l'opérateur de *Composition synchrone*. Le composant *STR2TR* assure la conversion d'interface STR vers TR, l'interface classiquement utilisée par les algorithmes de *model-checking* (cf. Listing 5.7).

**Définition 5.6.** L'interface TR (*Transition Relation*) utilisée pour appliquer des algorithmes de *model-checking* est définie avec les trois fonctions suivantes :

```
structure TR (C : Type) :=
  (initial : set C)
  (next : C → set C)
  (accepting : set C)
```

Listing 5.7 – Interface TR.

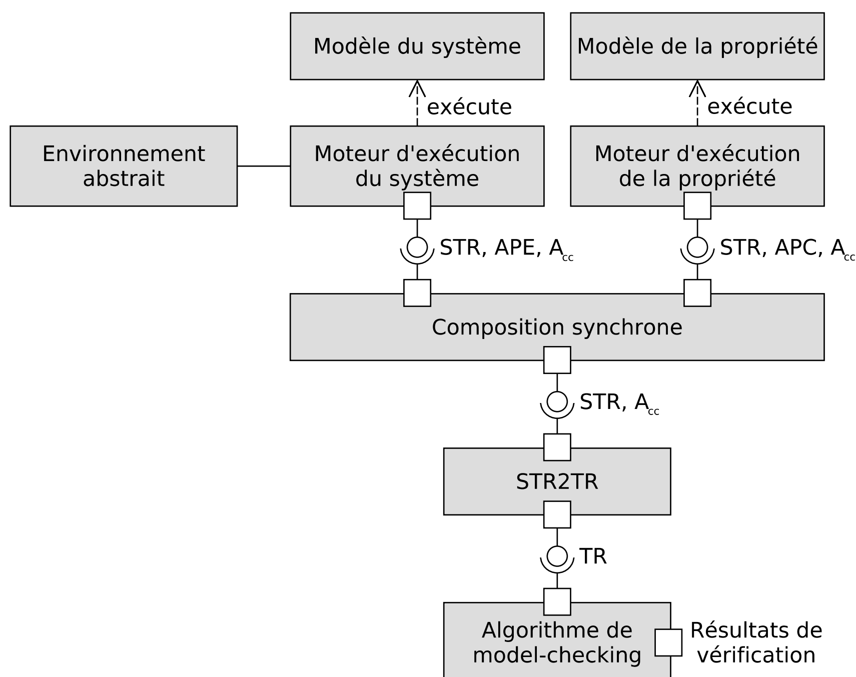



FIGURE 5.3 – Architecture utilisée pour le *model-checking*.

La fonction `initial` récupère l'ensemble des configurations initiales. La fonction `next` est un itérateur sur l'ensemble des configurations de l'espace d'état. Il fournit toutes les configurations qui peuvent être atteintes en exécutant un seul pas d'exécution à partir d'une configuration donnée. La fonction `accepting` permet de savoir si une configuration donnée est un état d'acceptation.

Étant donnée cette définition, la conversion d'interface STR vers TR peut désormais être formellement définie.

**Définition 5.7.** La conversion d'interface STR vers TR est définie par :

```
def STR2TR
   (str : STR C A)
   (acc : Acc C)
: @TR C := {
initial := str.initial,
next := λ c, { n | ∀ a ∈ str.actions c, ∀ t ∈ str.execute c a, n = t },
accepting := acc.accepting }
```

Listing 5.8 – Conversion de l'interface STR en TR.

La fonction `initial` correspond exactement à la fonction `initial` de la STR. L'implémentation de la fonction `next` consiste à prendre toutes les actions disponibles dans la configuration donnée puis à exécuter chacune de ces actions afin de récupérer toutes les configurations pouvant être atteintes. Enfin, la fonction `accepting` est implémentée en utilisant la fonction `accepting` de l'interface $A_{cc}$ qui indique pour une configuration donnée s'il s'agit d'un état d'acceptation. Grâce à cette conversion d'interface, notre architecture logicielle est compatible avec les algorithmes de *model-checking* classiquement utilisés dans la littérature [Hol97 ; Dha+12 ; DP04 ; BK08 ; Bar+17].

### 5.4.5 Architecture conceptuelle pour le *monitoring*

La Figure 5.4 présente l'application de l'architecture de vérification formelle au *monitoring* afin de surveiller le comportement du système à l'exécution. Dans la communauté formelle, le formalisme typiquement utilisé pour cette activité est celui des automates observateurs. Grâce à l'approche EMI, les mêmes automates observateurs et la même architecture de vérification formelle (excepté le *Contrôleur d'exécution*) peuvent donc être utilisés en *model-checking* et en *monitoring*.

En termes d'architecture, la *Boucle d'exécution* principale de la plateforme embarquée permet de piloter l'exécution du produit synchrone et fait office de contrôleur d'exécution. Contrairement au *model-checking* qui réalise une exploration exhaustive, ici seule la trace d'exécution

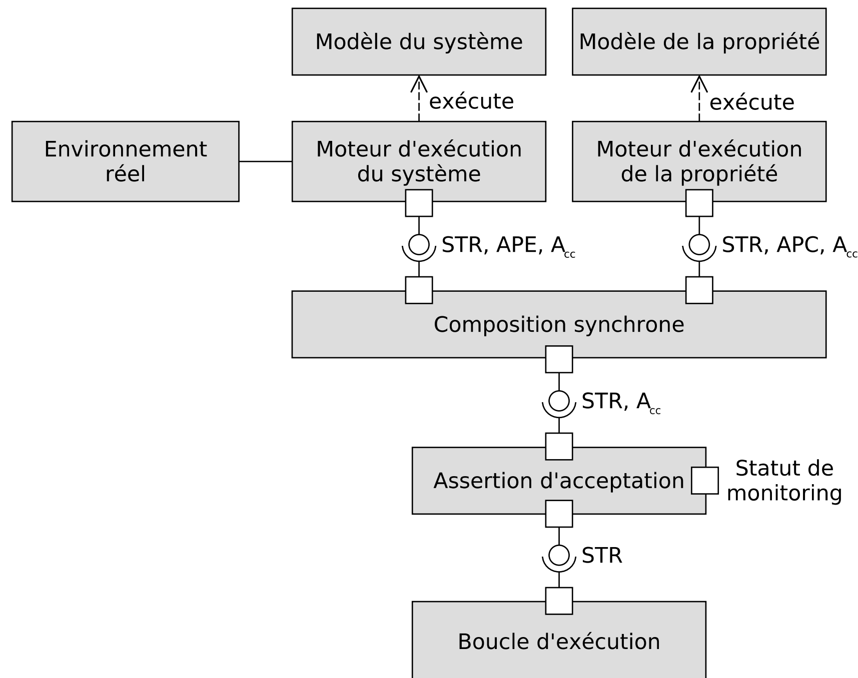



FIGURE 5.4 – Architecture utilisée pour le *monitoring*.

courante est considérée. La composition synchrone prend en compte l'exécution d'un système ordonnancé dans son *Environnement réel*. Le *Statut de monitoring* est déterminé par l'*Assertion d'acceptation* qui utilise l'interface $A_{cc}$ pour savoir si la configuration courante du produit synchrone est un état d'acceptation. En cas de violation de la propriété, le changement d'état du *Statut de monitoring* permet d'avertir l'utilisateur ou de déclencher des mécanismes de sûreté de fonctionnement (p. ex. de recouvrement d'erreurs).

## 5.5 Composition synchrone

La mise en œuvre de notre architecture de vérification formelle repose sur la composition synchrone de l'exécution de la propriété avec l'exécution du système. Cette section présente l'opérateur de composition synchrone permettant de réaliser cette tâche.

### 5.5.1 Description théorique

De manière informelle, le principe de la composition synchrone est assez simple : chaque fois que le système exécute un pas, l'automate de la propriété réalise aussi un pas pour suivre l'exécution du système. En d'autres termes, à chaque pas d'exécution, une transition synchrone

composée d'une transition du système et d'une transition de la propriété est tirée. Ainsi, une défaillance dans l'exécution du système est détectée dès qu'elle se produit.

Pour montrer comment cet opérateur fonctionne plus concrètement, une description formelle de notre opérateur de composition synchrone va maintenant être présentée. Cet opérateur est relié à deux moteurs d'exécution : l'un exécutant le modèle du système et l'autre exécutant le modèle de la propriété. Pour cette définition formelle, nous allons considérer que l'automate du système est représenté par le terme `lhs` et que celui de la propriété est modélisé par le terme `rhs`. L'opérateur de composition synchrone utilise l'interface STR pour piloter chacun des moteurs d'exécution ainsi qu'une fonction d'évaluation pour récupérer les propositions atomiques (interface APC) ou les évaluer (interface APE).

**Définition 5.8.** La composition synchrone de l'exécution du système (`lhs`) avec l'exécution de la propriété (`rhs`) donne une STR définie par :

```
def synchronous_composition (C₁ C₂ A₁ A₂ L₁ : Type)
   (lhs : STR C₁ A₁)
   (ape : APE C₁ A₁ L₁)
   (rhs : STR C₂ A₂)
   (apc : APC C₂ A₂ L₁)
: STR (C₁ × C₂) (A₁ × A₂) := {
initial := { c | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial), c = (c₁, c₂) },
actions := λ c, { a | match c with
   | (c₁, c₂) := ∀ (a₁ ∈ lhs.actions c₁) (a₂ ∈ rhs.actions c₂)
      (t₁ ∈ lhs.execute c₁ a₁) (t₂ ∈ rhs.execute c₂ a₂),
      match t₁, t₂ : ∀ t₁ t₂, Prop with
         | t₁, t₂ := ∀ l ∈ apc.eval c₂ a₂,
            ape.eval l c₁ a₁ t₁ = tt → a = (a₁, a₂)
      end
   end },
execute := λ c a, { t | match c, a with
   | (c₁, c₂), (a₁, a₂) :=  ∀ (t₁ ∈ lhs.execute c₁ a₁)
      (t₂ ∈ rhs.execute c₂ a₂), t = (t₁, t₂)
   end }}
```

Listing 5.9 – Opérateur de composition synchrone.

La configuration initiale du produit synchrone (`initial`) est la concaténation de la configuration initiale du système et de la configuration initiale de la propriété. Pour construire des transitions synchrones (`actions`), le moteur d'exécution de la propriété exécute sa fonction

`apc.eval` (interface APC) sur toutes les actions disponibles et retourne la liste des propositions atomiques nécessaires pour leurs évaluations. Ces atomes sont des prédicats qui sont évalués avec `ape.eval` (interface APE) sur chaque pas d'exécution du système (c.-à-d. sur le tuple configuration source $c_1$, action $a_1$ et configuration cible $t_1$). Le pas d'exécution $a_1$ doit donc être exécuté afin de récupérer la configuration cible du système. En utilisant le résultat de l'évaluation des atomes, les actions de l'automate de la propriété peuvent désormais être évaluées pour savoir si elles sont exécutables. Si une action est exécutable, cela signifie que cette action de l'automate de la propriété ($a_2$) peut être synchronisée avec celle de l'automate du système ($a_1$). Les actions étant ici des transitions, on obtient donc une transition synchrone correspondant au tuple ($a_1$, $a_2$). La dernière partie concerne l'exécution de ces transitions synchrones (**`execute`**). À partir d'une configuration donnée du produit synchrone, une transition synchrone est tirée afin de récupérer sa configuration cible. Cela signifie que la transition du système ($a_1$) est tirée sur l'automate du système et que la transition de la propriété ($a_2$) est tirée sur l'automate de la propriété. La concaténation des deux configurations cibles ($c_1$, $c_2$) donne la configuration cible de la transition synchrone.

De plus, l'opérateur de composition synchrone fournit également une implémentation de l'interface $\text{A}_{\text{cc}}$ permettant de savoir si un état du produit synchrone est un état d'acceptation. Pour cela, l'opérateur de composition synchrone appelle la fonction `accepting` sur chacune des deux entrées et réalise un "et" logique entre les deux valeurs booléennes retournées. En pratique, chaque état de l'automate du système est un état d'acceptation. Un état du produit synchrone est donc un état d'acceptation s'il contient un état d'acceptation de l'automate de la propriété.

En résumé, cet opérateur de composition synchrone permet de fournir au contrôleur d'exécution un point de vue sur le système analysé, et ce, sans avoir besoin de modifier l'implémentation de la sémantique du langage de modélisation. En effet, ce point de vue (c.-à-d. l'interface STR de sortie) est directement construit à partir du point de vue sur l'exécution du système (c.-à-d. l'interface STR `lhs`) et du point de vue sur l'exécution de l'automate de la propriété (c.-à-d. l'interface STR `rhs`).

### 5.5.2 Optimisation pour le *monitoring*

Pour le *monitoring*, l'opération de composition synchrone peut être optimisée puisque seul un chemin d'exécution est parcouru. Pour gagner en efficacité, l'ordonnanceur de la plateforme d'exécution est sollicité avant la construction des transitions synchrones pour choisir la transition du système qui sera tirée au prochain pas d'exécution. Ainsi, la composition synchrone n'a besoin d'être appliquée que sur la transition sélectionnée par l'ordonnanceur et non sur toutes les transitions tirables du système. Ces transitions tirables sont toujours calculées dans la configuration courante de l'exécution du système et la prochaine transition à tirer est toujours

exécutée à partir de cette configuration. La configuration cible de la transition tirée sera alors considérée comme configuration courante au prochain pas d'exécution. Lorsque la transition sélectionnée par l'ordonnanceur a été tirée, il est possible de déterminer quelle transition de la propriété peut être synchronisée avec celle du système afin de tirer ladite transition sur l'automate de la propriété. Non seulement ce mécanisme fait davantage sens pour le *monitoring* car la transition du système est seulement tirée une fois mais cela contribue aussi à améliorer les performances d'exécution.

## 5.6 *Model-checking* avec des propriétés LTL

Après avoir proposé une architecture logicielle pour la vérification formelle, nous allons voir comment celle-ci peut être appliquée au *model-checking* de propriétés LTL. Ce langage basé sur la logique temporelle est l'un des plus utilisés dans la communauté formelle [Hol00 ; Kan+15 ; DP04 ; Bar+17 ; Cim+02 ; LB08 ; Cla+03].

### 5.6.1 Spécification de propriétés LTL

Une première façon d'exprimer des propriétés formelles est d'utiliser les logiques prévues à cet effet comme la Logique Temporelle Linéaire (LTL). Les propriétés sont alors souvent exprimées sous forme textuelle à l'aide de propositions atomiques et d'opérateurs. Les propositions atomiques sont des prédicats portant sur l'exécution du système qui sont exprimés dans l'approche EMI à l'aide du langage d'observation. Ces propositions atomiques sont ensuite reliées entre elles avec des opérateurs de la logique du premier ordre (p. ex. et (&&), ou (or), non (!), implication (->)) ou des opérateurs de logique temporelle (p. ex. *globally* ([]), *eventually* (<>), *until* (U), *weak until* (W)).

L'utilisation de ces opérateurs temporels nécessite des connaissances formelles notamment pour en comprendre le sens et pour pouvoir combiner correctement les propositions atomiques entre elles. À l'instar des patrons de conception, les patrons de Dwyer[1] [DAC98] fournissent des patrons pour les propriétés LTL usuelles et leurs significations en langage naturel. Seules les propositions atomiques doivent être remplacées pour pouvoir spécifier une propriété à l'aide d'un patron de Dwyer. Un langage d'observation basé directement sur les concepts du langage de modélisation apporterait donc une brique complémentaire pour rendre le *model-checking* davantage accessible aux ingénieurs sans expertise formelle.

Pour pouvoir utiliser des propriétés LTL en *model-checking*, le formalisme des automates de Büchi est utilisé. La négation de chaque propriété LTL est transformée automatiquement en automate de Büchi afin d'obtenir un automate décrivant l'ensemble des comportements

1. Patrons de Dwyer pour LTL : https://matthewbdwyer.github.io/psp/patterns/ltl.html.

indésirables ne devant pas se retrouver dans le système. Deux librairies sont particulièrement efficaces pour obtenir les automates de Büchi correspondants à des propriétés LTL. Il s'agit de SPOT [DP04] et ltl3ba [Bab+12]. En termes d'expressivité, les propriétés LTL peuvent exprimer à la fois des propriétés de sûreté et de vivacité mais elles restent toutefois moins expressives que les automates de Büchi. En revanche, ces propriétés sont plutôt compactes et la génération des automates de Büchi associés peut facilement être automatisée. Ces deux avantages ont contribué à la renommée de ce langage de spécification de propriétés dans la communauté du *model-checking*.

En utilisant le même principe, cette approche est également applicable à d'autres types de logiques permettant l'expression de propriétés formelles comme CTL. Les mêmes propositions atomiques peuvent être employées, seuls les opérateurs logiques permettant de les combiner changent en fonction de la logique utilisée.

### 5.6.2 Architecture logicielle pour la vérification LTL

L'architecture logicielle utilisée pour coupler un *model-checker* LTL à un interpréteur EMI est présentée sur la Figure 5.5. Elle correspond à une spécialisation de l'architecture conceptuelle de la Figure 5.3 pour la vérification de propriétés LTL non temporisées avec un interpréteur EMI comme moteur d'exécution.

L'*Interpréteur EMI* interprète le *Modèle du système* et le *Modèle d'environnement abstrait* connecté sur son port *I/O*. Cet interpréteur encode la sémantique du langage de modélisation et est piloté par le *model-checker* connecté sur son port de contrôle d'exécution (*CE*). Sur le *model-checker*, un composant d'exécution dédié, appelé *Moteur d'exécution proxy*, implémente la couche applicative du protocole de communication permettant de piloter l'exécution de l'interpréteur et d'évaluer des propositions atomiques. Ce composant est connecté à l'interpréteur via un *Serveur de langages* qui fournit des services pour l'analyse de modèles (p. ex. une projection de la configuration et des pas d'exécution).

La propriété LTL à vérifier est également prise en entrée du *model-checker*. En pratique, la négation de cette propriété est convertie en un automate de Büchi avec la librairie ltl3ba [Bab+12]. L'automate de Büchi résultant est ensuite interprété par le *Moteur d'exécution Büchi* qui encode la sémantique des automates de Büchi. L'opérateur de *Composition synchrone* permet ensuite de composer l'exécution de l'automate de Büchi de la propriété avec l'exécution du système.

Cet opérateur de *Composition synchrone* est ensuite piloté par un *Algorithme de model-checking* permettant de détecter des boucles d'acceptation dans l'automate de Büchi résultant de la composition synchrone. Dans le cadre de cette thèse, nous utilisons l'algorithme proposé par Gaiser et Schwoon dans [GS09] basé sur deux "DFS imbriqués" : l'un permettant d'explorer l'espace d'état et l'autre de détecter des boucles d'acceptation à la volée. L'exploration de

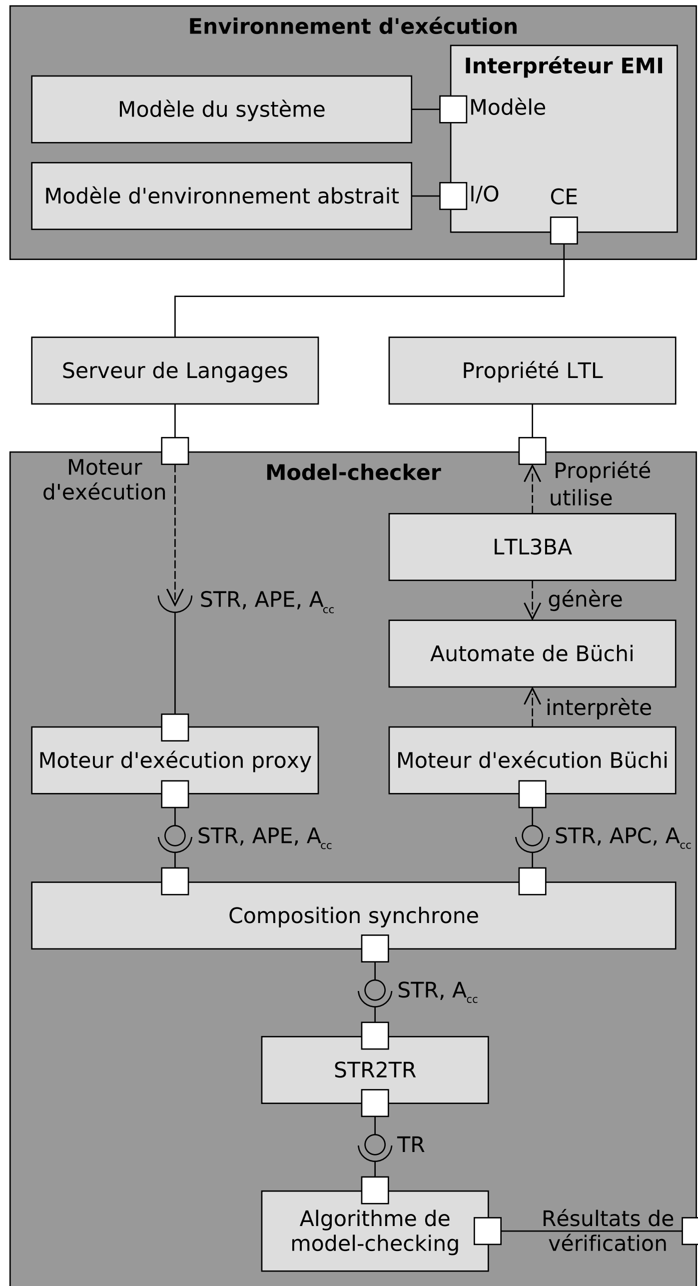



FIGURE 5.5 – Architecture utilisée pour le *model-checking* avec des propriétés LTL.

l'espace d'état continue jusqu'à ce qu'une boucle d'acceptation soit détectée (propriété violée) ou qu'un point fixe sur l'ensemble des configurations explorées soit atteint (propriété vérifiée).

## 5.7 Synthèse

Diverses activités d'analyse peuvent être effectuées directement sur le modèle de conception du système en utilisant un interpréteur EMI comme moteur d'exécution. Pour y parvenir, les outils d'analyse peuvent piloter l'exécution du modèle mais aussi observer son exécution en déléguant l'évaluation de propositions atomiques à l'interpréteur EMI. Grâce à ces mécanismes, l'approche EMI permet de découpler les algorithmes d'analyse des aspects propres au langage de modélisation et à l'exécution des modèles qui s'y conforment. Cette approche permet également de faciliter la compréhension des résultats d'analyse car ceux-ci sont directement exprimés avec les concepts du langage de modélisation.

Pour la vérification formelle, une architecture logicielle permet de coordonner l'exécution du modèle du système avec un automate représentant la propriété formelle à vérifier. Cette technique peut notamment être utilisée pour faire du *model-checking* ou du *monitoring*. Elle repose sur l'utilisation d'un opérateur de composition synchrone dont la définition a été formalisée. La définition d'opérateurs est une perspective intéressante pour définir les différents aspects d'un langage. Cette piste est également celle qui a été suivie dans le chapitre suivant pour composer l'ordonnanceur avec l'exécution du système.

CHAPITRE 6

# L'ORDONNANCEMENT POUR LA VÉRIFICATION ET L'EXÉCUTION EMBARQUÉE DE MODÈLES

---

**Sommaire**


**6.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **100**
**6.2 Non-déterminisme** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **101**
**6.3 Formalisation des opérateurs** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **102**
6.3.1 Opérateur de filtrage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
6.3.2 Opérateur d'ordonnancement . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.3.3 Opérateur de composition asynchrone . . . . . . . . . . . . . . . . . . . 105
**6.4 Vérification formelle et exécution embarquée avec ordonnancement** . . . **106**
6.4.1 Exécution réelle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
6.4.2 *Model-checking* avec filtrage . . . . . . . . . . . . . . . . . . . . . . . . 109
6.4.3 *Model-checking* avec l'ordonnanceur dans la boucle de vérification . . 110
6.4.4 *Model-checking* avec un environnement découplé . . . . . . . . . . . 112
**6.5 Synthèse** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **114**


## 6.1 Introduction

Lors de l'exécution d'un modèle, les différents processus de ce modèle (p. ex. *threads*, acteurs, objets actifs), que nous appellerons unités d'exécution, doivent être synchronisés et ordonnancés. Dans la littérature, différents mécanismes ont été définis pour traiter le problème de concurrence (p. ex. dans SCADE [Ber07], UML [OMG17d], Erlang [Arm07]). L'ordonnancement et les aspects liés à la concurrence d'un langage sont *(i)* souvent rattachés aux mécanismes de la plateforme d'exécution à travers des transformations complexes (p. ex. de la génération de code) ou *(ii)* implicitement encodés dans la sémantique du langage. Il est donc difficile d'adapter l'ordonnancement aux besoins du langage de modélisation.

De plus, l'ordonnanceur et les mécanismes de concurrence ont un impact direct sur l'exécution des modèles et peuvent être responsables de bogues comme des *deadlocks*. Ils doivent donc être pris en compte au cours de la phase de vérification. Pour y parvenir, les approches de la littérature [Hol97 ; Bar+17] reposent sur des transformations de modèles ou une abstraction de la plateforme d'exécution. Ces techniques étant sensibles aux fossés sémantiques ou au problème d'équivalence, la prise en compte de l'ordonnanceur dans la boucle de vérification reste complexe à mettre en œuvre.

Pour traiter ces problèmes, l'approche EMI présente une architecture logicielle permettant de composer de façon modulaire le système, l'ordonnanceur et l'environnement (réel ou abstrait). Dans ce chapitre, la section 6.2 commence par présenter les différents types de non-déterminisme puis la section 6.3 propose une formalisation des différents opérateurs permettant de réaliser la composition modulaire. Ces opérateurs sont ensuite combinés en section 6.4 pour construire l'architecture logicielle permettant d'ordonnancer l'exécution sur une cible embarquée ou de prendre en compte l'ordonnanceur au cours des activités de vérification formelle.

## 6.2 Non-déterminisme

Pour mieux comprendre le rôle de l'ordonnancement, nous allons présenter plus en détail les différents types de non-déterminisme. D'une manière générale, l'exécution possède du non-déterminisme si au moins deux actions sont exécutables à un même instant. Les quatre types de non-déterminisme rencontrés dans l'approche EMI sont :

1. Le non-déterminisme du langage ;
2. Le non-déterminisme général ;
3. Le non-déterminisme de l'environnement ;
4. Le non-déterminisme d'exécution.

Le *non-déterminisme du langage* (ou *"non-determinism or"* [Hoa78]) est directement lié à certains concepts du langage de modélisation qui permettent d'offrir un choix entre plusieurs actions pour une même unité d'exécution. Ce mécanisme peut être offert explicitement par un concept dédié du langage comme l'opérateur `select` du langage Fiacre [Ber+08]. Ce type de non-déterminisme peut aussi apparaître si le langage offre un certain laxisme. C'est par exemple le cas d'UML puisque certaines transitions d'un même objet actif peuvent être tirables en même temps. En effet, rien n'oblige les gardes des transitions sortantes d'un même état à être exclusives. Le *non-déterminisme du langage* est notamment utilisé dans les langages de spécification pour décrire différentes alternatives.

Le *non-déterminisme général* (ou *"general choice"* [Hoa78]) apparaît lorsque plusieurs unités d'exécution sont exécutées de manière concurrente. L'ensemble des pas exécutables correspond à la somme des pas exécutables de chaque unité d'exécution. Ce non-déterminisme est typiquement résolu en utilisant un ordonnanceur.

Le *non-déterminisme de l'environnement* est lié au fait que l'environnement peut solliciter le système de différentes façons à un instant donné. Le modèle abstrait de l'environnement capture (tout ou partie) de ces choix ce qui crée du non-déterminisme au moment de l'exécution. L'environnement peut donc directement impacter le comportement du système en offrant seulement un sous-ensemble des actions possibles. Ce type de non-déterminisme n'est pas traité par l'ordonnanceur mais transférer à l'outil d'analyse afin d'explorer tous les chemins possibles.

Le *non-déterminisme d'exécution* est dû au fait que l'exécution de l'interpréteur en lui-même est non-déterministe. Si deux exécutions avec le même interpréteur, le même modèle du système, le même modèle d'environnement (mêmes sollicitations arrivant aux mêmes instants) produisent des traces différentes, on dit alors qu'il y a du non-déterminisme d'exécution.

En résumé, l'ordonnanceur doit résoudre le non-déterminisme général lié à l'exécution concurrente de plusieurs unités d'exécution ainsi que le non-déterminisme du langage si cela est nécessaire.

## 6.3 Formalisation des opérateurs

Notre architecture logicielle repose sur trois opérateurs : un opérateur de filtrage, une spécialisation de cet opérateur pour l'ordonnancement et un opérateur de composition asynchrone. Ces opérateurs sont combinés avec le moteur d'exécution implémentant la sémantique du langage pour ordonnancer l'exécution de systèmes concurrents. Cette section fournit une description formelle de ces opérateurs en utilisant le langage Lean.

### 6.3.1 Opérateur de filtrage

Cet opérateur prend en entrée une STR et applique un filtrage sur un sous-ensemble de ses actions. La sélection des actions à filtrer est réalisée en accord avec une politique de filtrage (`FilteringPolicy`). En général, une politique de filtrage possède un état d'exécution qui est désigné avec le type `S` dans notre description formelle.

**Définition 6.1.** La politique de filtrage est définie de la façon suivante :

```
structure FilteringPolicy (C A S : Type) :=
   (initial : S)
   (selector : set A)
```

```
  (apply : S → C → set A → set (S × A))
  (subset¹ : ∀ s c 𝔸 (sa ∈ (apply s c 𝔸)), prod.snd sa ∈ 𝔸)

def StatelessFilteringPolicy (C A : Type) := FilteringPolicy C A unit
```

Listing 6.1 – Définition de la politique de filtrage.

La fonction `initial` retourne l'état d'exécution initial (`S`) de la politique de filtrage. La fonction `selector` permet de sélectionner les actions sur lesquelles le filtre est appliqué. Étant donné cet ensemble d'actions (`set A`), une configuration (`C`), et l'état d'exécution du filtre (`S`), la fonction `apply` définit les actions qui doivent être propagées sur la sortie et celles qui doivent être supprimées. Pour chaque action propagée, cette fonction retourne également le nouvel état d'exécution de la politique de filtrage. Comme indiqué par la contrainte `subset`, l'ensemble des actions retournées sur la sortie est un sous-ensemble de l'ensemble des actions (`𝔸`) données en entrée. En utilisant cette définition de la politique de filtrage, il est également possible de définir une version sans état d'exécution appelée `StatelessFilteringPolicy`.

Grâce à cette définition de la politique de filtrage, nous pouvons maintenant donner une définition formelle de l'opérateur de filtrage appelé `filter`.

**Définition 6.2.** Un filtre prend en entrée une STR et une politique de filtrage et retourne une nouvelle STR pour laquelle les actions ont été filtrées.

```
def filter
  (m : STR C A)
  (s : FilteringPolicy C A S)
: STR (C × S) (S × A) := {
initial := {cs | ∀ (c ∈ m.initial), cs = (c, s.initial)},
actions := λ cs,
  let toFilter := {a ∈ m.actions (prod.fst cs) | s.selector a},
      toForward := {fa | ∀ a ∈ m.actions (prod.fst cs),
        ¬ s.selector a → fa = (prod.snd cs, a)}
  in (s.apply (prod.snd cs) (prod.fst cs) toFilter) ∪ toForward,
execute := λ cs sa,
  let r := m.execute (prod.fst cs) (prod.snd sa)
  in {x | ∀ c ∈ r, x = (c, (prod.fst sa))}}
```

Listing 6.2 – Opérateur de filtrage.

1. En Lean, `prod.fst` et `prod.snd` retournent respectivement le premier terme et le second terme d'un produit. Par conséquent, `prod.snd sa` permet d'obtenir le second terme du couple (état du filtre x action), c.-à-d. l'action.

Pour chaque configuration initiale obtenue via l'interface STR, la fonction `initial` retourne un couple contenant la configuration initiale de la STR et l'état d'exécution initial de la politique de filtrage. La fonction `actions` commence par séparer les actions sur lesquelles la politique de filtrage doit être appliquée (`toFilter`) et les actions qui doivent seulement être propagées (`toForward`). La politique de filtrage est ensuite appliquée sur l'ensemble `toFilter` et les actions non filtrées sont regroupées avec `toForward`. La fonction `execute` délègue l'exécution d'une action donnée à la STR et récupère l'ensemble des configurations cibles. Chacune d'entre elles est composée d'une des configurations cibles obtenues via l'interface STR et de l'état d'exécution de la politique de filtrage.

### 6.3.2 Opérateur d'ordonnancement

Parmi un ensemble d'actions, l'ordonnanceur choisit l'une d'entre elles pour être le prochain pas d'exécution du modèle. Le choix de cette action dépend d'une politique d'ordonnancement (`SchedulingPolicy`) pouvant être vue comme une spécialisation de la politique de filtrage.

**Définition 6.3.** La politique d'ordonnancement est définie de la façon suivante :

```
structure SchedulingPolicy (C A S : Type)
      extends (FilteringPolicy C A S) :=
   (unique : ∀ s c 𝔸 (a ∈ (apply s c 𝔸)) (b ∈ (apply s c 𝔸)), a = b)

def StatelessSchedulingPolicy (C A : Type) := SchedulingPolicy C A unit
```

Listing 6.3 – Définition de la politique d'ordonnancement.

La politique d'ordonnancement étend la politique de filtrage avec la contrainte `unique` de sorte que la fonction `apply` retourne une unique action plutôt qu'un ensemble. De plus, il est possible de définir une version de la politique d'ordonnancement sans état d'exécution appelée ici `StatelessSchedulingPolicy`.

En utilisant cette définition de la politique d'ordonnancement, nous pouvons maintenant définir formellement notre opérateur d'ordonnancement appelé `scheduler`. En se basant sur l'interface STR, il offre un point de vue sur l'exécution du système ordonnancé sans modifier la sémantique du langage de modélisation. Dans notre approche, l'ordonnancement peut être vu comme un filtre sur la liste des pas d'exécution disponibles à un instant donné au cours de l'exécution du système. Pour un ensemble non vide de pas d'exécution, l'ordonnanceur doit choisir au plus $P$ pas d'exécution où $P$ est le nombre de processeurs sur la plateforme d'exécution. Dans cette thèse, nous nous focalisons sur des cibles avec un seul processeur ($P = 1$) ce qui implique que l'ordonnanceur doit sélectionner au plus un pas d'exécution. Il est également important de noter que la sémantique du langage peut fournir un ensemble vide de

pas d'exécution disponibles (p. ex. à cause de *deadlocks* ou parce que le modèle a terminé son exécution). Dans ce cas, l'ordonnanceur ne retourne aucun pas d'exécution.

**Définition 6.4.** L'ordonnanceur prend en entrée une STR et une politique d'ordonnancement et retourne une nouvelle STR qui a au plus une action disponible à chaque instant, l'action sélectionnée par la politique d'ordonnancement.

```
def scheduler
   (m : STR C A)
   (s : SchedulingPolicy C A S)
: STR (C × S) (S × A) := filter C A S m ↑s
```

Listing 6.4 – Opérateur d'ordonnancement.

Cette définition montre que l'ordonnanceur peut facilement être défini comme un filtre. Il suffit de caster la politique d'ordonnancement en une politique de filtrage avec une flèche montante (↑).

### 6.3.3 Opérateur de composition asynchrone

La dernière opération nécessaire pour définir notre architecture d'ordonnancement est la composition asynchrone. Cette tâche est réalisée par un opérateur dédié qui permet de regrouper les actions venant de deux STRs différentes. La composition est réalisée de façon asynchrone de telle façon qu'un pas d'exécution fait seulement progresser l'exécution d'une des deux STRs. L'autre ne fait rien, on dit qu'elle "bégaie". On appelle alors pas de bégaiement (en anglais *stuttering step*) le pas d'exécution implicite de la configuration courante de cette STR sur elle-même.

**Définition 6.5.** L'opérateur de composition asynchrone prend en entrée deux STRs et retourne la composition asynchrone de ces deux entrées sous la forme d'une nouvelle STR.

```
def asynchronous_composition { C₁ A₁ C₂ A₂ : Type }
   (lhs : STR C₁ A₁)
   (rhs : STR C₂ A₂)
: STR (C₁ × C₂) (A₁ ⊕ A₂) := {
initial  := {c | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial), c = (c₁, c₂)},
actions  := λ c, {a | match c : ∀ C, Prop with
   | (c₁, c₂) := ∀ (a₁ ∈ lhs.actions c₁) (a₂ ∈ rhs.actions c₂),
      a = sum.inl a₁ ∨ a = sum.inr a₂
   end},
execute  := λ c a, {a' | match c : ∀ C, Prop with
```

```
  | (c₁, c₂) := match a with
    | (sum.inl a₁) := ∀ c₁' ∈ lhs.execute c₁ a₁, a' = (c₁', c₂ )
    | (sum.inr a₂) := ∀ c₂' ∈ rhs.execute c₂ a₂, a' = (c₁ , c₂')
    end
  end}}

notation lhs '⊗ₐ' rhs := asynchronous_composition lhs rhs
```

Listing 6.5 – Opérateur de composition asynchrone.

La fonction `initial` retourne le produit cartésien des configurations initiales des deux STRs. La fonction `actions` regroupe toutes les actions des deux STRs sous forme d'un type somme ($A_1 \oplus A_2$) qui représente une union disjointe des deux types d'actions ($A_1$ et $A_2$). Pour exécuter une action, la fonction `execute` cherche à quelle STR cette action appartient puis l'exécute sur cette STR tandis que l'autre bégaie. Chaque configuration cible retournée est un couple contenant une configuration de la STR sur laquelle l'action a été exécutée et la configuration courante de l'autre STR. Notre formalisation fournit également la notation $\otimes_a$ pour utiliser cet opérateur, notation communément utilisée en *model-checking* pour la composition asynchrone.

## 6.4 Vérification formelle et exécution embarquée avec ordonnancement

En utilisant ces trois opérateurs (c.-à-d. filtrage, ordonnancement, composition asynchrone), nous avons défini une composition entre le système, l'ordonnancement et l'environnement. Cette composition modulaire peut être utilisée à la fois pour l'exécution sur une plateforme embarquée et pour la vérification formelle. Ces opérateurs permettent d'ordonnancer l'exécution de systèmes concurrents lorsqu'ils sont déployés sur une cible embarquée. Durant la phase de vérification, les mêmes opérateurs peuvent également être utilisés pour prendre en compte l'ordonnanceur dans la boucle de vérification et ainsi réduire le fossé sémantique avec l'exécution réelle.

### 6.4.1 Exécution réelle

Pour exécuter un modèle concurrent, nous introduisons en Figure 6.1 l'architecture logicielle d'un moteur d'exécution pour un langage de haut niveau (p. ex. un *Domain Specific Modeling Language* (DSML)). Les composants $SEU_1$ à $SEU_N$ sont les différentes unités d'exécution du système (*System Execution Units* (SEUs)). Leur exécution est réalisée en se basant sur

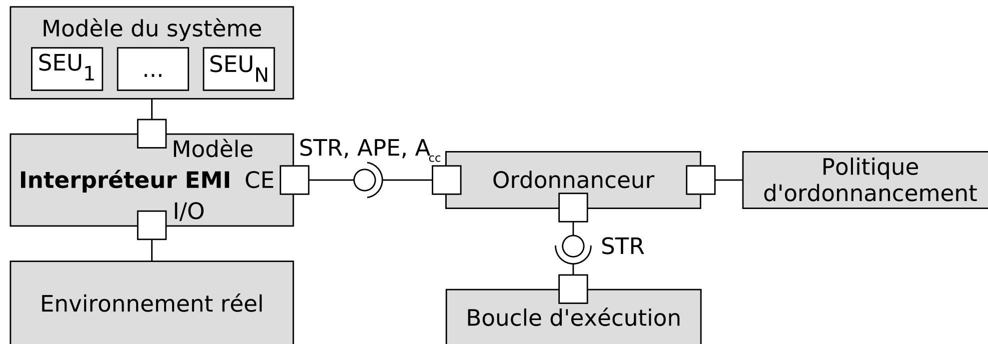



FIGURE 6.1 – Architecture pour l'ordonnancement de l'exécution réelle.

la sémantique du langage de modélisation. Dans cette thèse, nous suggérons d'encoder la sémantique du langage sous forme opérationnelle dans un interpréteur (*Interpréteur EMI*) qui implémente l'interface STR. Cet interpréteur peut interagir avec l'*Environnement réel* au travers d'une API qui permet d'échanger des données avec les périphériques (*I/O*) (p. ex. des capteurs ou des actionneurs) de la plateforme matérielle (p. ex. une carte embarquée).

Cette architecture utilise notre opérateur d'ordonnancement (*Ordonnanceur*) pour ordonnancer l'exécution concurrente d'un modèle en accord avec la *Politique d'ordonnancement*. L'*Ordonnanceur* sélectionne le prochain pas d'exécution parmi l'ensemble des actions disponibles. L'exécution de ce pas est ensuite déclenchée par la *Boucle d'exécution* principale de la plateforme d'exécution. Ce composant exécute seulement deux instructions en boucle comme le montre le Listing 6.6. *(1)* Il demande à l'ordonnanceur de récupérer l'ensemble des pas d'exécution disponibles et de choisir l'un d'entre eux (`actions` de l'interface STR). *(2)* Il demande l'exécution de ce pas d'exécution à l'ordonnanceur qui délègue son exécution à l'interpréteur et met à jour l'état d'exécution de la politique d'ordonnancement (`execute` de l'interface STR).

```
// Récupération de la configuration initiale
Configuration current = str.initial();

while(1) {
    // Récupération du prochain pas d'exécution
    List<Actions> actions = str.actions(current);
    // Exécution du pas d'exécution
    current = str.execute(current, actions.get(0));
}
```

Listing 6.6 – Pseudo-code de la boucle d'exécution principale.

La STR contrôlée par la boucle d'exécution est donnée par la définition formelle suivante de notre architecture logicielle.

**Définition 6.6.** L'architecture logicielle pour l'exécution réelle est formellement définie de la façon suivante :

```
def runtime_execution
    (system : STR C A)
    (scheduling_policy : SchedulingPolicy C A S)
: STR (C × S) (S × A) := scheduler C A S system scheduling_policy
```

Listing 6.7 – Architecture pour l'exécution réelle.

L'ordonnanceur peut être configuré avec différentes politiques d'ordonnancement qui sont regroupées en deux catégories. Les politiques d'ordonnancement *sans état d'exécution* (ou en anglais *stateless scheduling policies*) peuvent choisir un pas d'exécution en se basant uniquement sur l'ensemble des pas d'exécution disponibles et le contenu de la configuration courante. Par exemple, l'ordonnancement à priorité fixe est une politique d'ordonnancement sans état d'exécution. Elle sélectionne le pas d'exécution appartenant à l'unité d'exécution avec la priorité statique la plus élevée. En revanche, les politiques d'ordonnancement *avec état d'exécution* nécessitent également une zone mémoire pour stocker des données persistantes. À titre d'exemple, la politique d'ordonnancement *round-robin* sélectionne à tour de rôle un pas d'exécution de chaque unité d'exécution. Dans ce cas, il est nécessaire de stocker la variable indiquant quelle unité d'exécution aura la plus forte priorité à la prochaine itération.

L'utilisation d'un opérateur d'ordonnancement à plusieurs avantages. Premièrement, l'ordonnanceur est indépendant du modèle de concurrence utilisé mais est lié à la sémantique du langage au travers de l'interface STR. L'ordonnanceur apparaît explicitement dans notre architecture et est configuré par une politique d'ordonnancement permettant de s'adapter aux besoins d'une application ou d'un domaine. Deuxièmement, cet ordonnanceur respecte la sémantique du langage de telle sorte qu'il ne puisse pas briser l'atomicité du langage. Par exemple pour UML, il doit respecter la sémantique d'exécution du *Run-To-Completion step* qui assume que le traitement d'un nouvel évènement ne peut débuter que si le traitement de l'évènement courant est terminé et qu'un état d'exécution stable a été atteint. L'ordonnanceur doit aussi faire confiance dans le fait que la sémantique du langage va relâcher le contrôle et ne pas s'exécuter infiniment. Cette confiance est donnée au concepteur du langage (c.-à-d. l'ingénieur en charge d'implémenter la sémantique du langage) plutôt qu'au développeur de l'application logicielle ou au développeur de la plateforme d'exécution sous-jacente (comme c'est habituellement le cas). Le concepteur du langage est en effet la personne la plus apte pour réaliser cette tâche grâce à son expertise de la sémantique du langage. Troisièmement, contrairement aux ordonnanceurs des OS temps réel pour lesquels l'ordonnanceur et la boucle d'exécution principale ne

font qu'une entité, nous avons découplé ces deux composants afin de rendre l'ordonnanceur réutilisable et pilotable pour la phase de vérification.

### 6.4.2 *Model-checking* avec filtrage

Pour le *model-checking*, le système doit être fermé par une abstraction de l'environnement réel pour la vérification formelle (cf. section 5.2). Cette abstraction est modélisée comme différentes unités d'exécution de l'environnement (ou en anglais *Environment Execution Units* (EEUs)) qui sont exécutées par le même interpréteur EMI. Le *model-checker* est directement connecté à ce composant (cf. section 5.4.4) afin d'explorer l'espace d'état complet du modèle. Cependant, sur des systèmes industriels, cette technique peut mener à une explosion de l'espace d'état notamment à cause de tous les entrelacements d'évènements venant de l'environnement. Pour éviter cette explosion combinatoire, il est possible d'utiliser du *model-checking* compositionnel [CLM89] ou des techniques de génération automatique d'environnements [Tka08] mais il peut aussi être nécessaire de considérer des hypothèses de haut niveau.

En effet, pour réduire ces entrelacements d'évènements, une stratégie est de considérer l'*hypothèse de réactivité*. Cette hypothèse a été considérée pour la première fois pour un langage synchrone en [DSH94]. L'hypothèse de réactivité suppose que l'exécution du système est infiniment plus rapide que les réactions de l'environnement. Cela signifie que si le système a des pas d'exécution disponibles, l'un d'entre eux doit être exécuté sinon un pas provenant de l'environnement doit être exécuté pour créer une nouvelle stimulation. En conséquence, cette hypothèse réduit le nombre d'entrelacements causés par les évènements de l'environnement.

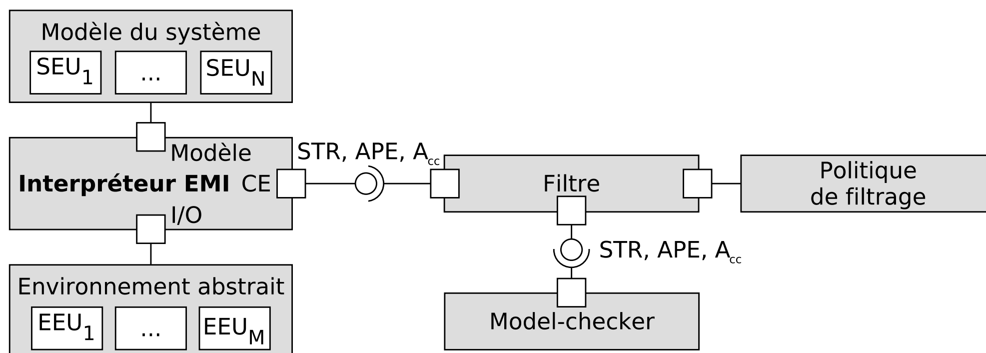



FIGURE 6.2 – Architecture de *model-checking* avec filtrage.

D'un point de vue technique, l'hypothèse de réactivité agit comme un filtre sur l'ensemble des actions disponibles. L'architecture logicielle présentée sur la Figure 6.2 peut être utilisée pour considérer l'hypothèse de réactivité durant le *model-checking*. Cette architecture n'est cependant pas spécifique à l'implémentation de l'hypothèse de réactivité mais peut être utilisée pour appliquer n'importe quel politique de filtrage. La même architecture peut par exemple être

utilisée pour supprimer les actions qui causent des débordements sur des files de messages à taille fixe. Pour cela, seule la politique de filtrage doit être remplacée. Dans la Figure 6.2, notre opérateur de filtrage (*Filtre*) est introduit entre l'*Interpréteur EMI* et le *model-checker*. Ce filtre est paramétré par une *Politique de filtrage* qui spécifie quels pas d'exécution doivent être filtrés. Par exemple, pour appliquer l'hypothèse de réactivité, la politique de filtrage doit sélectionner en priorité un pas d'exécution du système ou un pas d'exécution de l'environnement si aucun pas du système n'est disponible.

**Définition 6.7.** L'architecture de *model-checking* avec filtrage est formellement définie de la façon suivante :

```
def model_checking_with_filtering
   (sys_and_env : STR C A)
   (filtering_policy : FilteringPolicy C A S)
: STR (C × S) (S × A) := filter C A S sys_and_env filtering_policy
```

Listing 6.8 – Architecture de *model-checking* avec filtrage.

Dans cette définition formelle et toutes celles qui vont suivre jusqu'à la fin de ce chapitre, la STR résultante est celle qui est contrôlée par le *model-checker*. Tous nos opérateurs permettent également de propager les requêtes des autres interfaces (APE et $A_{cc}$ notamment).

### 6.4.3 *Model-checking* avec l'ordonnanceur dans la boucle de vérification

Cette section décrit comment prendre en compte l'ordonnancement dans la boucle de vérification. Le fait de considérer un aspect spécifique à la plateforme d'exécution durant le *model-checking* permet de supprimer, dans l'espace d'état, certains chemins d'exécution qui ne sont jamais parcourus lors de l'exécution réelle à cause des choix d'ordonnancement.

Comme mentionné dans la section 5.2, une abstraction de l'environnement réel du système est nécessaire pour appliquer la vérification formelle. Pour inclure l'ordonnanceur dans la boucle de vérification, un premier réflexe serait d'étendre manuellement la politique d'ordonnancement pour que l'ordonnanceur puisse choisir arbitrairement parmi les pas d'exécution du système et ceux de l'environnement celui qui devra être exécuté. Cependant, cette approche simpliste induit une interférence de l'environnement sur l'ordonnancement puisque l'ordre des pas d'exécution du système est ici potentiellement différent de celui considéré lors de l'exécution réelle.

Une meilleure solution est simplement d'exclure les pas d'exécution de l'environnement des choix offerts à l'ordonnanceur. Nous ne voulons pas ordonnancer l'environnement en particulier à cause de son non-déterminisme intrinsèque (c.-à-d. le non-déterminisme de l'environnement en section 6.2) et parce que nous ne savons pas comment ordonnancer les pas

de l'environnement (c.-à-d. avec quelle politique d'ordonnancement ?). C'est la raison pour laquelle l'ordonnanceur doit simplement propager tous les pas d'exécution de l'environnement et choisir seulement un pas d'exécution parmi ceux du système. Cette solution est illustrée sur la Figure 6.3. Cette architecture logicielle présente comment composer de façon modulaire le système, l'ordonnanceur et l'abstraction de l'environnement pour le *model-checking*.

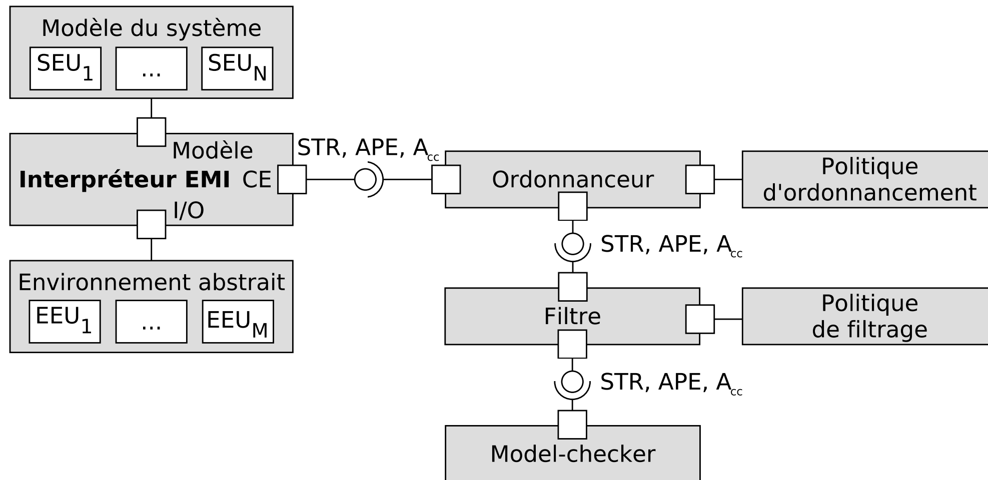



FIGURE 6.3 – Architecture de *model-checking* avec ordonnancement.

Cette architecture logicielle repose sur notre opérateur d'ordonnancement. Sa fonction `selector` permet d'appliquer l'ordonnancement seulement sur les pas venant des SEUs. Tous les pas d'exécution possibles du système sont ainsi pris en compte par l'ordonnanceur qui sélectionne l'un d'entre eux en accord avec la politique d'ordonnancement choisie. Le pas sélectionné est ensuite regroupé avec tous les pas d'exécution possibles de l'environnement avant d'être envoyé au *model-checker* soit directement soit via un composant de filtrage (p. ex. pour appliquer l'hypothèse de réactivité).

**Définition 6.8.** L'architecture de *model-checking* avec l'ordonnanceur dans la boucle de vérification est formellement définie de la façon suivante :

```
def model_checking_with_scheduling (S₁ S₂ : Type)
   (sys_and_env : STR C A)
   (scheduling_policy : SchedulingPolicy C A S₁)
   (filtering_policy : FilteringPolicy (C×S₁) (S₁×A) S₂)
: STR ((C×S₁)×S₂) (S₂×(S₁×A)) :=
   filter (C×S₁) (S₁×A) S₂
      (scheduler C A S₁ sys_and_env scheduling_policy)
      filtering_policy
```

Listing 6.9 – Architecture de *model-checking* avec ordonnancement.

Pour avoir des exécutions reproductibles en *model-checking*, la politique d'ordonnancement doit être *déterministe*. Étant donné un ensemble de pas d'exécution, une configuration donnée et un état d'ordonnancement donné, la politique d'ordonnancement est déterministe (au sens déterminisme d'exécution) si l'ordonnanceur choisit toujours le même pas d'exécution. Ceci n'est pas en contradiction avec le fait que l'exécution de modèle peut quant à elle posséder du non-déterminisme général. En *model-checking*, un ordonnanceur déterministe est nécessaire pour résoudre le non-déterminisme général du système mais le non-déterminisme de l'environnement est quant à lui propagé jusqu'au *model-checker*. Même une politique d'ordonnancement aléatoire est déterministe si l'état du générateur de nombres aléatoires est sauvegardé dans l'état d'exécution de la politique d'ordonnancement. À chaque pas, cet état est restauré de façon à ce que le générateur de nombres aléatoires retourne un nombre déterministe (voir la fonction *rand_r* du langage C).

Notre approche pour la vérification de modèles offre une façon modulaire de composer le système, l'ordonnanceur et l'environnement à l'aide de deux opérateurs : un ordonnanceur et un filtre (p. ex. pour appliquer l'hypothèse de réactivité). Cette architecture considère l'ordonnanceur dans la phase de vérification et permet ainsi de réduire le fossé entre l'exécution réelle et la vérification de modèles. Cette approche permet également d'améliorer les performances de *model-checking* en évitant d'explorer certains chemins d'exécution qui sont impossibles compte tenu des choix de la politique d'ordonnancement. Cependant, cette prise en considération de l'ordonnanceur en *model-checking* a aussi une contrepartie. Les résultats de vérification obtenus ne sont plus valides avec d'autres politiques d'ordonnancement. Le moteur d'exécution doit en effet être déployé avec la même politique d'ordonnancement que celle utilisée en phase de vérification formelle. Une autre limitation est que l'environnement d'exécution peut interférer avec l'exécution du système car les unités d'exécution du système et de l'environnement sont toutes exécutées en utilisant la même instance de l'interpréteur EMI. En conséquence, l'exécution de l'environnement peut altérer la configuration du système (p. ex. en cas de pointeurs errants (ou en anglais *dangling pointers*)).

### 6.4.4 *Model-checking* avec un environnement découplé

Pour résoudre ces problèmes d'interférences, nous proposons d'introduire une seconde instance de l'interpréteur de modèles (*Interpréteur EMI (environnement)*) qui interprète seulement l'abstraction de l'environnement. Un tel découplage permet de connecter facilement différentes abstractions de l'environnement à l'exécution du système en remplaçant seulement l'interpréteur de modèles correspondant. Cela peut également être intéressant pour isoler chaque exécution dans des zones de mémoire séparées afin de préserver l'intégrité de leurs états d'exécution. Cette architecture logicielle est présentée sur la Figure 6.4. L'abstraction de l'environnement doit être composée de façon asynchrone avec le système ordonnancé produit par

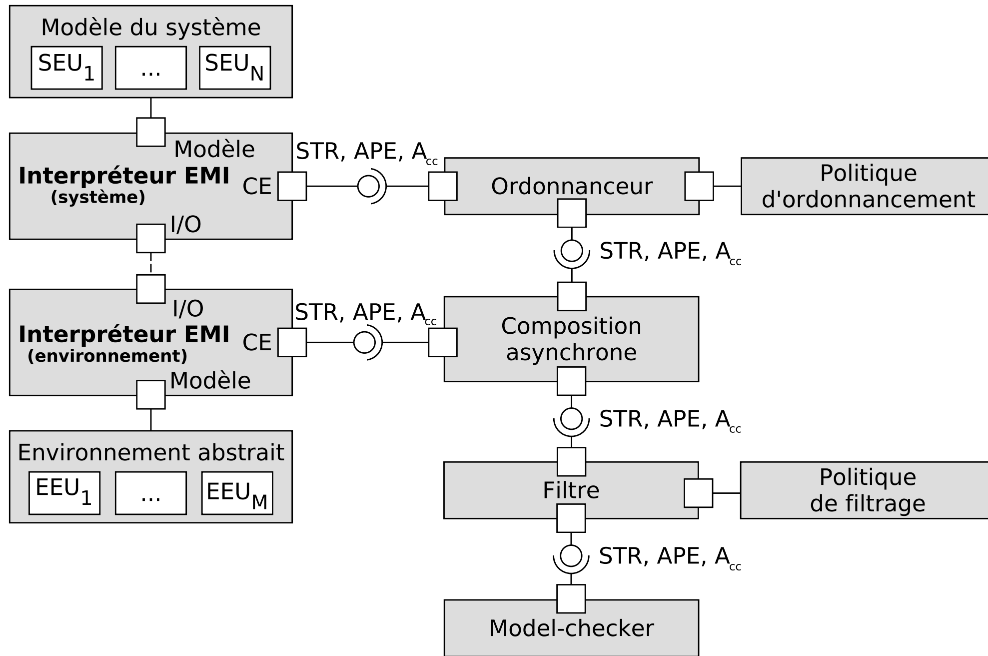



FIGURE 6.4 – Architecture de *model-checking* avec l'ordonnanceur dans la boucle de vérification et un environnement découplé.

l'ordonnanceur afin d'obtenir le modèle fermé nécessaire aux algorithmes de *model-checking*. Cette tâche est réalisée par l'opérateur de *Composition asynchrone* et non plus directement par la sémantique du langage de modélisation comme auparavant. Un *Filtre* peut ensuite être utilisé pour filtrer certaines actions et appliquer certaines hypothèses (p. ex. l'hypothèse de réactivité) sur la STR fournie par la composition asynchrone. Cette architecture est finalement contrôlée par un *model-checker* qui applique des algorithmes de vérification formelle exhaustive sur l'espace d'état du modèle.

**Définition 6.9.** L'architecture de *model-checking* avec le modèle d'environnement découplé du modèle du système est formellement définie de la façon suivante :

```
def model_checking_with_decoupled_env (C₁ C₂ A₁ A₂ S₁ S₂ : Type)
   (system : STR C₁ A₁)
   (environment : STR C₂ A₂)
   (scheduling_policy : SchedulingPolicy C₁ A₁ S₁)
   (filtering_policy : FilteringPolicy ((C₁×S₁)×C₂) ((S₁×A₁) ⊕ A₂) S₂)
: STR (((C₁×S₁)×C₂)×S₂) (S₂×((S₁×A₁) ⊕ A₂))  :=
   filter ((C₁ × S₁) × C₂) ((S₁×A₁) ⊕ A₂) S₂
   ((scheduler C₁ A₁ S₁ system scheduling_policy) ⊗ₐ environment)
   filtering_policy
```

Listing 6.10 – Architecture de *model-checking* avec découplage de l'environnement.

De plus, les unités d'exécution de l'environnement ont besoin d'échanger des données avec les unités d'exécution du système. Pour cela, les ports *I/O* des interpréteurs EMI sont connectés ensemble pour relier les sorties de l'environnement aux entrées du système et les sorties du système aux entrées de l'environnement. Dans notre architecture logicielle, la communication est réalisée via des variables partagées de façon à relier ensemble les données d'exécution des deux composants. Différents mécanismes de communication interprocessus (ou en anglais *Inter-Process Communication* (IPC)) doivent également pouvoir être utilisés en définissant d'autres opérateurs de composition en plus de la composition asynchrone.

Si les unités d'exécution du système et de l'environnement sont spécifiées dans le même langage de modélisation, l'*Interpréteur EMI (système)* et l'*Interpréteur EMI (environnement)* peuvent être deux instances du même interpréteur de modèles. Cependant, il est également possible de spécifier les unités d'exécution de l'environnement dans un autre langage produisant ainsi une exécution hétérogène de modèles. Cette possibilité est un défi scientifique constituant une perspective de ces travaux de thèse. En pratique, l'exécution hétérogène de modèles n'est pas triviale à réaliser car elle nécessite de définir une conversion sémantique et structurelle des données échangées afin d'exprimer les données sources en termes de la sémantique cible et avec la forme appropriée. Ces conversions sont non seulement complexes mais elles doivent également être prouvées afin que nous puissions les utiliser en vérification formelle. L'exécution hétérogène induit également d'autres problèmes comme la synchronisation et la coordination des exécutions du système et de l'environnement. Plusieurs travaux [Buc+01 ; Var+15] dans la communauté de l'IDM se sont focalisés sur ces problématiques et ont proposé des solutions comme le langage de coordination BCOol [Var+15].

## 6.5 Synthèse

Les préoccupations liées à la concurrence font partie intégrante de la définition d'un langage concurrent et doivent, pour cette raison, être considérées à la fois en phase d'exécution et de vérification. Ce chapitre a présenté une approche permettant de définir une composition modulaire du système, de l'ordonnanceur et de l'environnement. Ainsi, l'ordonnanceur utilisé lors de l'exécution réelle peut désormais être intégré dans la boucle de vérification formelle afin de vérifier le comportement du système ordonnancé.

Cette composition modulaire est réalisée en utilisant différents opérateurs. Le *filtre* permet de prendre en compte certaines hypothèses (p. ex. l'hypothèse de réactivité) en *model-checking*. L'*ordonnanceur* devient un composant explicite et pilotable pouvant être configuré par une politique d'ordonnancement. Ces opérateurs ont été formellement définis et peuvent être combinés pour définir l'architecture logicielle répondant aux besoins de l'activité réalisée. Ils permettent à la fois de redescendre certaines hypothèses de haut niveau (p. ex. l'hypothèse

de réactivité) dans les outils d'analyse et de remonter certains concepts propres à la plateforme d'exécution (p. ex. l'ordonnancement) à un niveau d'abstraction plus élevé.

Cette technique permet ainsi de réduire le problème d'équivalence entre ce qui est vérifié et ce qui est exécuté. Elle contribue également à améliorer les performances de *model-checking* en supprimant de l'espace d'état les chemins d'exécution devenus impossibles par rapport aux choix d'ordonnancement réalisés.

CHAPITRE 7

# PILOTAGE ET OBSERVATION DE L'EXÉCUTION DE MODÈLES

## Sommaire


**7.1 Introduction** . . . . . . . . . . **116**
**7.2 Pilotage de l'interpréteur** . . . . . . . . . . **117**
**7.3 Le langage d'observation** . . . . . . . . . . **119**
7.3.1 Présentation du langage d'observation . . . . . . . . . . 119
7.3.2 Les opérateurs du langage d'observation . . . . . . . . . . 122
7.3.3 Évaluation des prédicats . . . . . . . . . . 123
**7.4 Architecture logicielle d'analyse avec un interpréteur EMI** . . . . . . . . . . **123**
7.4.1 Description de l'architecture logicielle . . . . . . . . . . 123
7.4.2 Le serveur de langages . . . . . . . . . . 124
7.4.3 Canonisation de la configuration . . . . . . . . . . 125
**7.5 Synthèse** . . . . . . . . . . **125**


## 7.1 Introduction

Pour analyser le comportement d'un système logiciel, les outils d'analyse doivent (*i*) piloter et (*ii*) observer l'exécution de ce système. Pour le pilotage, la section 7.2 présente l'interface utilisée pour piloter un interpréteur EMI. Cette interface permet d'implémenter l'interface STR utilisée par nos opérateurs et par les outils d'analyse pour contrôler l'exécution du modèle. Pour l'observation, les outils d'analyse ont besoin de pouvoir évaluer des propositions atomiques portant sur l'exécution du système. Ces propositions atomiques sont souvent exprimées dans des formalismes spécifiques (différents du langage de modélisation) qui sont difficiles à maîtriser par les ingénieurs. Pour faciliter cette tâche, la section 7.3 présente le langage d'observation utilisé dans l'approche EMI pour exposer les objets du système et exprimer les propositions atomiques directement en termes des concepts du langage de modélisation. Le langage d'observation est donc propre à chaque langage de modélisation. Il reste néanmoins important de définir explicitement les caractéristiques générales de ce langage. Une

fois exprimées, les propositions atomiques sont évaluées directement sur l'interpréteur EMI en utilisant l'architecture d'analyse présentée en section 7.4. Celle-ci repose notamment sur un serveur de langages permettant d'offrir des services aux outils d'analyse sans compromettre les performances d'exécution de l'interpréteur EMI.

## 7.2 Pilotage de l'interpréteur

Cette section présente la couche applicative du protocole de communication utilisé pour contrôler l'exécution de modèles sur un interpréteur EMI. Il est tout à fait possible d'utiliser directement l'interface STR. Néanmoins, la plupart des langages de modélisation sont des langages déterministes c.-à-d. avec un seul état initial et une seule configuration cible pour chaque pas d'exécution. L'interface EMI présentée ici est certes plus spécifique que l'interface STR mais elle offre un support plus adapté pour les langages déterministes. Cette nouvelle interface, plus naturelle à implémenter pour les ingénieurs, définit cinq requêtes pour piloter l'interprétation de modèles :

**Get configuration** récupère l'état d'exécution courant de l'interpréteur c.-à-d. toutes les données d'exécution modifiables et persistantes pour l'exécution de modèles ;

**Set configuration** charge une configuration comme état d'exécution courant de l'interpréteur afin de reprendre l'exécution de modèles dans cet état ;

**Get executable steps** récupère l'ensemble des pas exécutables à partir de la configuration courante de l'interpréteur ;

**Execute step** exécute un pas à partir de la configuration courante de l'interpréteur ;

**Reset** réinitialise l'interpréteur et repositionne l'exécution dans son état initial.

Ces cinq requêtes sont suffisantes pour contrôler l'exécution pas-à-pas de n'importe quel modèle. Pour les ingénieurs, le principal défi concerne l'implémentation des requêtes *Set configuration* et *Get executable steps*. En effet, la requête *Set configuration* offre la possibilité de retourner en arrière (c.-à-d. d'implémenter la fonctionnalité de *back-in-time* [Bou+17 ; PTP07]) alors que l'exécution de modèles suit généralement une seule trace d'exécution (la trace d'exécution courante). La requête *Get executable steps* nécessite quant à elle une représentation symbolique des pas d'exécution alors même que la notion de pas d'exécution est souvent implicitement définie.

Nous allons maintenant voir comment implémenter l'interface STR à partir de cette nouvelle interface de communication. À cette effet, il est important de souligner que l'interpréteur garde en mémoire le modèle (*model*) étant exécuté et la configuration courante stockée sous forme d'un tableau d'octets dans une zone de mémoire dynamique (*dynamic*). Comme une *state*

```
structure EMI : Type :=
  (model : EMIModel)
  (dynamic : EMIDynamicMemory)

def EMIState := state EMI
```

Listing 7.1 – Définition du type `EMI` et de la monade `EMIState`.

*monad*, toutes les fonctions de l'interface EMI peuvent accéder ou modifier cet état par effet de bord. Afin de définir l'implémentation de l'interface STR à partir de l'interface EMI, nous utilisons le langage Lean.

Pour commencer, une définition simple du type `EMI` et de la monade `EMIState` est donnée par le Listing 7.1. Le type `EMI` est en réalité plus complexe mais la définition présentée ici offre une abstraction appropriée pour notre formalisation. En particulier, les types `EMIModel` et `EMIDynamicMemory` (ainsi que `EMITransition`) sont des structures de données complexes qui ne sont pas détaillées ici mais seulement considérées comme des types de données abstraits.

Il est maintenant possible de définir les fonctions de l'interface EMI. Ces fonctions reposent sur `EMIState`, l'état d'exécution interne de l'interpréteur, et sur `EMITransition`, la représentation interne des pas d'exécution d'un modèle. L'implémentation de ces fonctions ne peut être présentée ici car elle dépend fortement de la sémantique du langage de modélisation. Néanmoins, les prototypes de ces fonctions peuvent être définis de la façon suivante :

```
def get_configuration : EMIState EMIDynamicMemory
def set_configuration (c : EMIDynamicMemory) : EMIState unit
def get_executable_steps : EMIState (set EMITransition)
def execute_step (t : EMITransition) : EMIState unit
def reset : EMIState unit
```

Listing 7.2 – Fonctions de l'API de EMI.

Étant donné ces définitions, l'interface STR peut maintenant être exprimée à partir de l'interface EMI. Dans ce but, trois fonctions basées sur la monade `EMIState` ont été définies afin de réaliser la conversion d'interface EMI vers STR.

```
def emi_initial
: EMIState EMIDynamicMemory :=
do reset, c ← get_configuration, return c

def emi_actions (c : EMIDynamicMemory)
: EMIState (set EMITransition) :=
```

```
do set_configuration c, a ← get_executable_steps, return a

def emi_execute (c : EMIDynamicMemory) (t : EMITransition)
: EMIState EMIDynamicMemory :=
do set_configuration c, execute_step t, x ← get_configuration, return x

def EMI2STR (emi : EMI)
: @STR EMIDynamicMemory EMITransition := {
initial := { prod.fst (emi_initial.run emi) },
actions := λ c, prod.fst ((emi_actions c).run emi),
execute := λ c a, { prod.fst ((emi_execute c a).run emi) }}
```

Listing 7.3 – Conversion de l'interface EMI en STR.

La conversion `EMI2STR` donne ainsi la possibilité de piloter l'exécution d'un modèle s'exécutant sur un interpréteur EMI avec l'interface STR. Dans le cas général, les fonctions `initial` et `execute` peuvent retourner plusieurs configurations (p. ex. avec certains langages synchrones). Cependant, dans cette approche, `EMI2STR` met en évidence qu'une seule configuration est retournée à chaque fois (la configuration initiale ou celle obtenue après avoir exécuté une action). L'interface EMI considère donc que l'exécution de modèles est déterministe ce qui est le cas pour la plupart des langages. Pour que l'interface EMI soit aussi expressive que l'interface STR, il faudrait que la requête `get_configuration` retourne un ensemble de configurations plutôt qu'une seule afin de prendre en compte le non-déterminisme de l'exécution d'un langage.

## 7.3 Le langage d'observation

Pour mener certaines activités d'analyse (p. ex. du *model-checking*, du débogage multivers), piloter l'exécution du modèle n'est pas suffisant, il faut aussi être capable d'observer son exécution. À cet effet, cette section présente le langage d'observation utilisé dans l'approche EMI pour exprimer des prédicats sur l'exécution du système dans les termes du langage de modélisation.

### 7.3.1 Présentation du langage d'observation

Le langage d'observation fournit différents opérateurs permettant d'introspecter et d'observer l'exécution du modèle. Ce langage se base directement sur les concepts du langage de modélisation et permet d'exposer les données d'exécution. Ce langage facilite donc l'expression des propriétés formelles par les ingénieurs puisque les propositions atomiques sont

spécifiées avec les mêmes concepts que la modélisation du système. Dans le cadre de cette thèse, ce langage est utilisé lors des activités d'analyse pour exprimer des prédicats pour :
— Les propositions atomiques des propriétés LTL ;
— Les conditions booléennes des points d'arrêt conditionnels.

Par souci d'uniformisation de la terminologie utilisée dans cette thèse, tous ces prédicats seront appelés propositions atomiques, ou atomes, dans le reste de ce manuscrit.

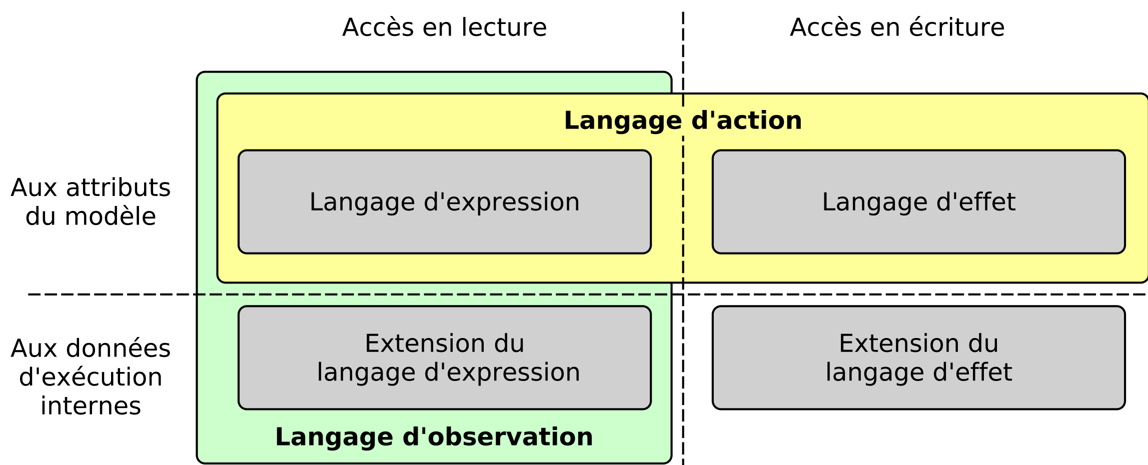



FIGURE 7.1 – Positionnement du langage d'observation par rapport au langage d'action d'un interpréteur EMI.

À l'instar du langage d'action qui permet de spécifier finement le comportement du système, le langage d'observation permet de définir finement les prédicats des propriétés formelles devant être vérifiées sur le système. La Figure 7.1 montre le positionnement du langage d'observation par rapport au langage d'action. Ces deux langages ont accès en lecture et/ou en écriture aux données d'exécution du modèle. On distingue deux types de données d'exécution : *(i)* les attributs dynamiques issus du modèle et *(ii)* les données internes au moteur d'exécution utilisées pour implémenter la sémantique du langage. En utilisant cette classification, les opérateurs d'un langage peuvent être divisés en quatre groupes : le langage d'expression, le langage d'effet, l'extension du langage d'expression et l'extension du langage d'effet.

Le langage d'action utilisé pour la modélisation du système est composé du langage d'expression et du langage d'effet employés respectivement pour lire et affecter des valeurs aux attributs du modèle. En pratique, le langage d'action ne doit pas donner accès aux données d'exécution internes pour deux raisons principales : *(i)* pour ne pas interférer avec l'exécution interne du moteur d'exécution et *(ii)* pour garder le modèle indépendant de tout moteur d'exécution.

Le langage d'observation se différencie du langage d'action sur plusieurs aspects. Il se compose du langage d'expression et de son extension lui permettant ainsi d'avoir un accès en lecture seule aux attributs dynamiques du modèle et aux données d'exécution internes. Contrairement au langage d'action, le langage d'observation est un langage sans effet de bord de sorte que son utilisation n'interfère pas avec l'exécution du modèle. Étant donné que les propriétés formelles sont exprimées sur des chemins d'exécution dynamiques, elles sont vérifiées sur un modèle donné s'exécutant sur un moteur d'exécution spécifique. Pour exprimer des propriétés formelles complexes à propos de l'exécution du modèle, il est naturel que les propriétés puissent aussi porter sur les données d'exécution internes au moteur d'exécution.

De la même façon que l'extension du langage d'expression, il est possible de définir une extension au langage d'effet pour changer les valeurs des données d'exécution internes. Cette possibilité n'a pas été exploitée dans cette thèse mais elle pourrait s'avérer utile pour certaines activités d'analyse comme le débogage ou pour déclencher des mécanismes de sûreté de fonctionnement. Il est en revanche interdit de l'utiliser pour définir les propositions atomiques des propriétés utilisées en *model-checking*. En effet, tout effet de bord risque de compromettre la vérification et/ou de modifier l'espace d'état. C'est la raison pour laquelle cette extension du langage d'effet ne fait pas partie du langage d'observation.

Pour exprimer les propositions atomiques des propriétés formelles, le langage d'observation fournit un ensemble d'opérateurs permettant d'accéder aux données d'exécution et de les exposer aux ingénieurs. Contrairement aux langages de programmation conventionnels, ce langage possède la particularité de manipuler des données qui sont déclarées et définies à l'extérieur du langage (c.-à-d. les éléments des modèles). Pour obtenir des références sur ces éléments externes, la phase d'analyse syntaxique (ou en anglais *parsing*) est complètement séparée de la phase de résolution des symboles. Chaque identifiant de l'analyse syntaxique est associé au symbole approprié en utilisant une table des symboles. Suivant le contexte, chaque symbole peut être recherché dans la table des symboles propre à un objet (p. ex. pour un attribut ou une méthode) ou dans la table des symboles globaux au modèle. De façon générale, ces opérations, permettant d'associer un identifiant à une référence, peuvent être réalisées dynamiquement lors de l'exécution du modèle ou statiquement lors de son chargement dans l'interpréteur.

**Application à UML.** Pour un interpréteur UML, les opérateurs du langage d'observation peuvent être définis directement dans le langage hôte (c.-à-d. le langage utilisé pour implémenter l'interpréteur de modèles). Étant donné que dans ces travaux de thèse l'interpréteur UML doit servir à exécuter des modèles de systèmes embarqués, l'association des identifiants aux références du modèle sera réalisée statiquement.

### 7.3.2 Les opérateurs du langage d'observation

Le langage d'observation fournit différents opérateurs appartenant soit au langage d'expression soit à son extension.

Les opérateurs du **langage d'expression** remplissent trois objectifs principaux : *(i)* lire les valeurs des attributs dynamiques issus du modèle, *(ii)* accéder au contexte de l'objet auquel appartient l'expression spécifiée (habituellement via le mot-clé `this` ou `self`) et *(iii)* naviguer le modèle. Tous ces opérateurs sont spécifiques au langage de modélisation et ne peuvent donc pas être explicités de manière générale.

Les opérateurs de l'**extension du langage d'expression** permettant quant à eux *(i)* de lire les valeurs des données d'exécution internes (p. ex. des compteurs de programme, des piles d'exécution) utilisées pour l'implémentation de la sémantique. D'autres opérateurs permettent *(ii)* d'accéder aux références des éléments racines des données d'exécution. Lors de l'écriture d'une proposition atomique pour une propriété LTL, cette proposition n'a pas de contexte d'exécution spécifique puisqu'elle n'est pas rattachée à un objet. Elle se trouve dans le contexte d'exécution global et il devient donc nécessaire de naviguer le modèle depuis ses racines pour accéder aux objets du modèle. Ce langage fournit également des opérateurs pour *(iii)* faciliter la navigation dans le modèle (p. ex. pour naviguer le modèle selon des modalités plus souples que celles normalement permises par le langage) et pour *(iv)* faciliter les activités de vérification formelle. Par rapport à ce dernier point, l'opérateur qui permet de détecter que l'exécution du modèle est bloquée dans l'état d'exécution courant est particulièrement utile pour la vérification de *deadlocks*. Tous ces opérateurs de l'extension du langage d'expression sont spécifiques aux données d'exécution utilisées et ne peuvent donc pas être employés sur toutes les plateformes d'exécution implémentant la sémantique d'un même langage de modélisation.

**Application à UML.** Le langage d'observation utilisé pour les modèles UML dans le cadre de ce projet possède différents opérateurs dont `GET` pour lire la valeur d'un attribut ou naviguer le modèle, `IS_IN_STATE` pour vérifier si l'état courant d'un objet actif est un état donné de sa machine à états, et `EP_IS_EMPTY` pour vérifier si l'*event pool* (c.-à-d. la file de stockage des évènements) associé à un objet actif donné est vide. `ROOT_instMain` donne accès à l'instance de la structure composite *Main* du modèle à partir duquel tous les autres objets peuvent être atteints. `GET_ACTIVE_PEER` permet de faciliter la navigation du modèle en offrant un accès à l'objet actif (p. ex. un objet de l'environnement) connecté à l'autre bout d'un lien de communication contenant des ports. En ce qui concerne les opérateurs facilitant les activités de vérification, `DEADLOCK` permet de vérifier si l'état d'exécution courant possède un *deadlock* et `OBSERVER_FAIL` permet de vérifier si un automate observateur a détecté une défaillance. La liste complète des opérateurs du langage d'observation pour notre interpréteur UML est donnée en annexe C.

### 7.3.3 Évaluation des prédicats

Une fois les propositions atomiques spécifiées à l’aide du langage d’observation, il faut pouvoir les évaluer et retourner le résultat de leurs évaluations aux outils d’analyse.

La connaissance de la configuration courante n’est généralement pas suffisante pour pouvoir évaluer ces propositions atomiques car elles dépendent également de la sémantique opérationnelle capturée dans l’interpréteur de modèles. Par exemple, l’évaluation d’un atome peut nécessiter d’exécuter des instructions (p. ex. des opérations (sans effets de bord) comme des appels de fonction) dont le comportement est défini par la sémantique du langage.

En conséquence, les outils d’analyse doivent faire intervenir l’interpréteur dans le processus d’analyse pour l’évaluation de ces prédicats. Dans la littérature, certains *model-checkers* comme LTSmin [Kan+15] effectuent l’évaluation des atomes directement sur la configuration en dehors de l’environnement d’exécution. Ces outils nécessitent de réimplémenter une partie de la sémantique du langage de modélisation ou limitent l’expressivité des propositions atomiques. Contrairement à ces approches, l’architecture proposée ici offre un meilleur découplage entre les outils d’analyse et le moteur d’exécution qui est le seul à avoir connaissance de la syntaxe abstraite et de la sémantique du langage de modélisation.

Pour prendre en compte ce besoin, l’interface EMI a été étendue avec une requête supplémentaire permettant de soumettre à l’interpréteur des propositions atomiques à évaluer. Cette requête permet d’implémenter la fonction d’évaluation de l’interface APE.

**Evaluate atomic propositions** envoie un ensemble de propositions atomiques à évaluer et retourne les résultats de leurs évaluations. L’interpréteur évalue ces prédicats directement dans l’environnement d’exécution du modèle.

## 7.4 Architecture logicielle d’analyse avec un interpréteur EMI

Cette section décrit plus en détail l’architecture logicielle utilisée pour piloter et observer l’exécution d’un modèle sur un interpréteur EMI. Cette architecture correspond à la solution technique mise en place dans ces travaux de thèse. Il ne s’agit donc que d’une proposition permettant l’analyse de modèles avec un interpréteur EMI.

### 7.4.1 Description de l’architecture logicielle

Le diagramme de la Figure 7.2 présente l’architecture logicielle permettant d’effectuer des activités d’analyse avec un interpréteur EMI comme moteur d’exécution. L’*Outil d’analyse* est connecté à l’*Interpréteur EMI* via un *Serveur de langages* qui offre divers services pour l’analyse de modèles. Au niveau de l’interpréteur, le *Pilote* coordonne l’exécution du modèle avec

les requêtes envoyées par l'*Outil d'analyse* sur le flux de communication. Le *Pilote* utilise également les services fournis par l'*Évaluateur* pour évaluer des propositions atomiques arbitraires sur l'exécution du modèle. En utilisant cette architecture, diverses activités d'analyse peuvent être réalisées à la fois sur des PCs de développement et des cibles embarquées. Par exemple, pour réaliser une simulation *hardware-in-the-loop* (c.-à-d. directement avec le matériel réel mais avec un environnement simulé), les outils d'analyse peuvent être connectés à une cible embarquée alors que pour faire du *model-checking* l'utilisation d'un PC de développement sera plus adapté.

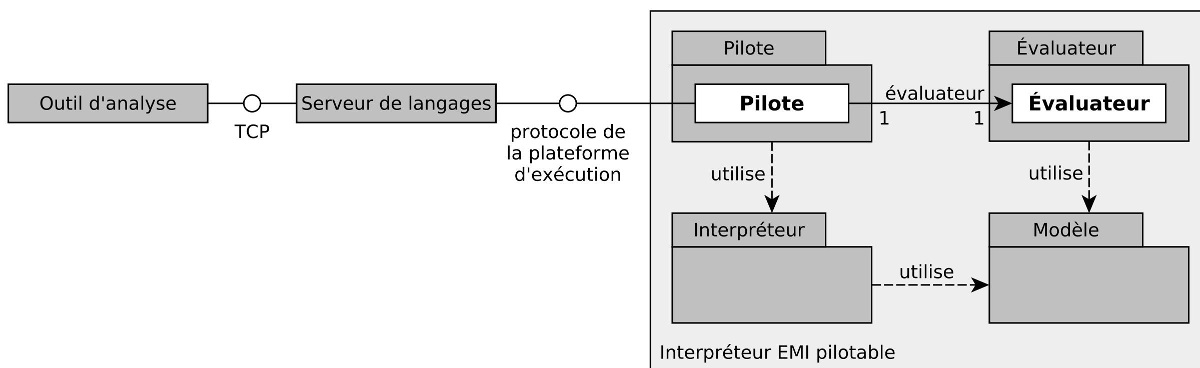



FIGURE 7.2 – Architecture logicielle pour l'analyse de modèles avec un interpréteur EMI.

### 7.4.2 Le serveur de langages

Pour faciliter la connexion des outils d'analyse à de multiples cibles embarquées et PCs de développement, notre approche repose sur l'utilisation d'un serveur de langage. Ce composant offre différents services aux outils d'analyse pour piloter ou observer l'exécution de modèles. Les rôles du serveur de langages sont :

— D'uniformiser la connexion des outils d'analyse à l'interpréteur de modèles via une connexion TCP en traduisant les trames TCP vers le protocole adéquat utilisé par la plateforme d'exécution (p. ex. un protocole série RS232 pour une cible embarquée, TCP pour un PC de développement) ;

— De construire une projection (c.-à-d. une vue) de la configuration et des pas d'exécution à partir des données binaires renvoyées par l'interpréteur, du modèle statique de conception, et de l'organisation des données d'exécution dans la mémoire de la plateforme d'exécution ;

— De convertir les propositions atomiques en code exécutable. Les propositions atomiques ne sont pas toujours connues lors de la création de l'exécutable de l'interpréteur. Par conséquent, elles doivent être compilées et chargées a posteriori sur l'interpréteur afin

d'être évaluées par l'*Évaluateur*.

Ainsi, pour se connecter à l'interpréteur de modèles, les outils d'analyse n'ont besoin d'implémenter qu'un client TCP et la couche applicative du protocole de communication utilisé pour échanger des messages avec le *Pilote* (c.-à-d. les six requêtes présentées dans ce chapitre).

### 7.4.3 Canonisation de la configuration

L'architecture d'analyse possède également un rôle important en *model-checking*. Chaque fois que la configuration courante de l'interpréteur est demandée par un outil d'analyse, le *Pilote* doit trouver la forme canonique de cette configuration. Lors de l'exécution, les données d'exécution évoluent de sorte que plusieurs configurations correspondent à un même état d'exécution. La forme canonique correspond à la configuration auxquelles toutes les autres configurations équivalentes peuvent se rapporter. Pour l'embarqué, il est par exemple courant d'implémenter des files (ou en anglais FIFO) avec des tableaux circulaires afin de ne pas avoir à recopier tous les éléments chaque fois que le premier élément est retiré de la file. Il existe donc plusieurs états mémoire où la file possède le même nombre d'éléments avec des contenus identiques. Pour obtenir la forme canonique d'une file implémentée avec un tableau circulaire, une solution est de déplacer tous les éléments vers le début du tableau et de réinitialiser toutes les cases vides avec la valeur 0. La canonisation de la configuration permet ainsi de réduire l'espace d'état à explorer par le *model-checker*.

## 7.5 Synthèse

Cette proposition d'architecture permet aux outils d'analyse de piloter et d'observer l'exécution d'un modèle sur un interpréteur EMI à travers une interface de communication commune. L'utilisation d'un langage d'observation permet de faciliter l'expression des propositions atomiques des propriétés formelles. Ces propositions atomiques portent directement sur l'exécution du système et sont évaluées par l'interpréteur EMI sur le modèle exécutable du système. L'évaluation de ces prédicats repose donc sur l'unique implémentation de la sémantique du langage de modélisation capturée dans l'interpréteur. Cette architecture offre ainsi un meilleur découplage entre le moteur d'exécution et les outils d'analyse. Ce découplage est également facilité par l'utilisation d'un serveur de langages permettant de déporter l'implémentation de certains services propres aux outils d'analyse hors de l'interpréteur. Ce serveur de langages implémente également la conversion de protocole de communication permettant aux outils d'analyse de se connecter à l'interpréteur lorsqu'il s'exécute sur une cible embarquée et ainsi d'assurer le continuum entre conception et exécution.

CHAPITRE 8

# CONCEPTION D'UN INTERPRÉTEUR DE MODÈLES EMBARQUÉ ET PILOTABLE

**Sommaire**

**8.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **126**
**8.2 Méthodologie de conception** . . . . . . . . . . . . . . . . . . . . . . . . . . **127**
**8.3 Métamodélisation** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **129**
8.3.1 Métamodélisation du langage de conception . . . . . . . . . . . . . . . 129
8.3.2 Métamodélisation des données d'exécution . . . . . . . . . . . . . . . 129
8.3.3 Le modèle objet et l'instanciation . . . . . . . . . . . . . . . . . . . . . 130
8.3.4 La notion de configuration . . . . . . . . . . . . . . . . . . . . . . . . . 133
8.3.5 Sérialisation des métamodèles . . . . . . . . . . . . . . . . . . . . . . . 135
**8.4 Implémentation de la sémantique opérationnelle** . . . . . . . . . . . . . . . **135**
8.4.1 Implémentation d'une sémantique pilotable . . . . . . . . . . . . . . . 135
8.4.2 Une sémantique utilisable en vérification formelle . . . . . . . . . . . . 136
**8.5 Chargement du modèle** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **137**
**8.6 Déploiement sur la plateforme d'exécution** . . . . . . . . . . . . . . . . . . **138**
**8.7 Synthèse** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **139**

## 8.1 Introduction

De nombreux interpréteurs (cf. section 1.3.3) permettent d'exécuter des modèles conformes à des DSMLs exécutables. Un interpréteur EMI se différencie de ces outils sur au moins trois points principaux. *(i)* Un interpréteur EMI possède l'implémentation de référence de la sémantique du langage de modélisation. *(ii)* L'exécution du modèle peut être pilotée par un contrôleur d'exécution (p. ex. des outils d'analyse). *(iii)* Un interpréteur EMI peut être déployé sur une plateforme embarquée et pas seulement sur un PC de développement. Ces trois spécificités impactent la conception d'un interpréteur EMI et les choix technologiques liés à son implémentation.

Pour concevoir un interpréteur EMI, la section 8.2 donne un aperçu de la méthodologie de conception employée dans cette thèse. Cette méthodologie permet de définir le métamodèle d'un langage de modélisation en section 8.3 et d'implémenter la sémantique de ce langage en section 8.4. Une fois que la syntaxe et la sémantique du langage de modélisation ont été définies, des modèles conformes à ce langage peuvent alors être conçus. La section 8.5 décrit les mécanismes nécessaires à l'instanciation de ces modèles et à leurs chargements dans l'interpréteur EMI. Enfin, la section 8.6 décrit comment ces modèles sont déployés et exécutés sur une cible embarquée.

## 8.2 Méthodologie de conception d'un interpréteur embarqué et pilotable

La conception et l'utilisation d'un interpréteur de modèles, ou plus généralement d'un moteur d'exécution pour un DSML, sont communément réalisées dans deux environnements de développement distincts (cf. GEMOC Studio [Bou+16]) :

— L'atelier à langages (ou en anglais *language workbench*) permet de concevoir le langage c.-à-d. de définir sa syntaxe abstraite, ses syntaxes concrètes, et d'implémenter sa sémantique ;
— L'atelier de modélisation (ou en anglais *modeling workbench*) est utilisé pour définir des modèles conformes à ce langage et offre les mécanismes permettant de les exécuter et de les analyser.

En suivant ces lignes directrices, cette section présente la méthodologie de conception retenue dans cette thèse pour concevoir un interpréteur de modèles de la définition du DSML jusqu'à l'obtention du binaire exécutable associé à un modèle donné. Les interpréteurs obtenus via cette méthodologie sont conformes à l'architecture candidate présentée dans la section 4.2. Ils sont basés sur une seule implémentation de la sémantique du langage de modélisation et peuvent à la fois être pilotés par des outils d'analyse externes et déployés sur des microcontrôleurs embarqués. Étant donné ces particularités, ce chapitre vise à mettre en lumière les caractéristiques d'un interpréteur EMI ainsi que les points clés de sa conception.

La Figure 8.1 donne un aperçu de la méthodologie de conception d'un interpréteur EMI exécutable avec (en haut) l'atelier à langages et (en bas) l'atelier de modélisation.

Dans l'atelier à langages, quatre activités doivent être réalisées pour concevoir la syntaxe du langage et définir sa sémantique :

1. La conception du métamodèle du langage (c.-à-d. de sa syntaxe abstraite) et d'au moins une syntaxe concrète ;
2. La conception d'une extension du métamodèle permettant de définir les données d'exé-

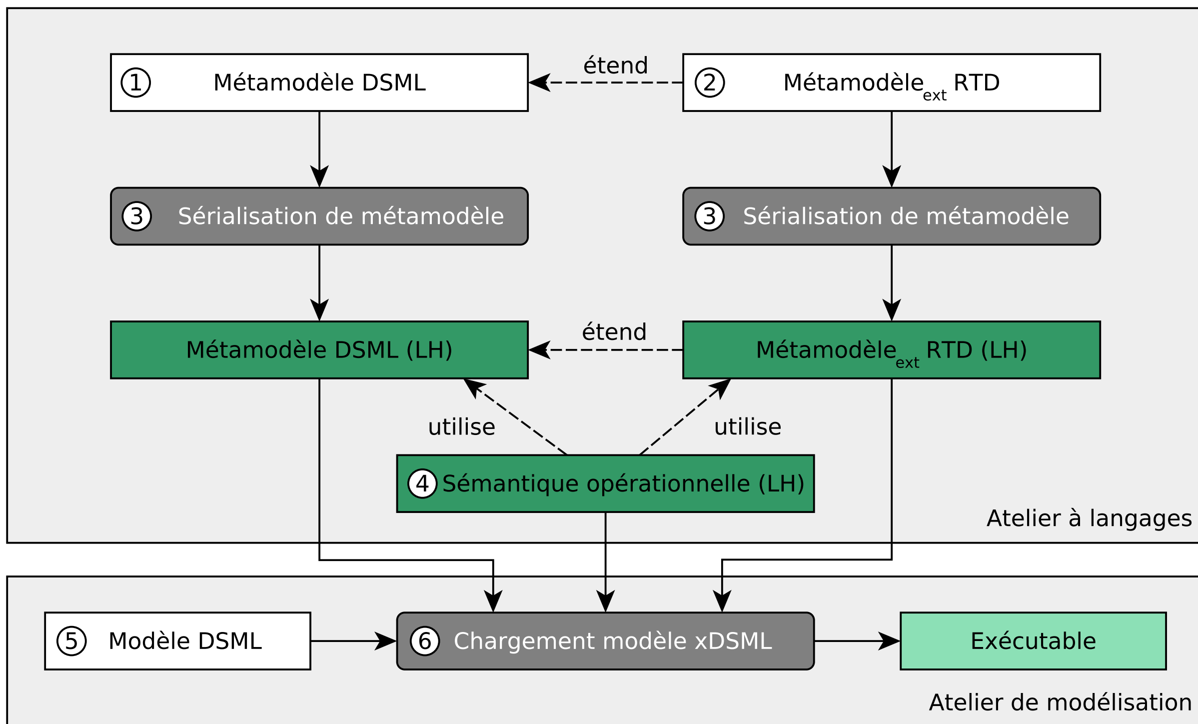



FIGURE 8.1 – Méthodologie de conception d'un interpréteur EMI.

cution (ou en anglais *RunTime Data* (RTD)) du modèle ;

3. La sérialisation du métamodèle et de son extension vers le langage d'implémentation de la sémantique (ou langage hôte (LH)) ;
4. L'implémentation de la sémantique opérationnelle du langage de modélisation dans le langage hôte afin de définir le comportement associé à chaque concept du langage mais aussi d'implémenter l'interface de pilotage STR.

Dans l'atelier de modélisation, deux activités permettent de définir des modèles conformes au langage et de les exécuter sur l'interpréteur conçu :

5. La modélisation d'un système sous forme d'un modèle en utilisant une des syntaxes concrètes (graphiques ou textuelles) du langage de modélisation ;
6. Le chargement du modèle dans l'interpréteur afin d'obtenir un exécutable permettant d'exécuter le modèle sur une cible embarquée ou de piloter son exécution avec un outil d'analyse.

Dans la suite de ce chapitre, chacune des étapes de cette méthodologie va être détaillée et illustrée sur le langage UML.

## 8.3 Métamodélisation

Lors de la conception d'un langage, la métamodélisation est l'activité qui permet de définir le métamodèle du langage de conception ainsi qu'une extension de ce métamodèle pour l'exécution. Cette extension permet de définir le modèle objet et les données dynamiques utilisées pour l'exécution de modèles conformes à ce langage.

### 8.3.1 Métamodélisation du langage de conception

L'élaboration du métamodèle du langage de modélisation est réalisée conformément à l'état de l'art en utilisant par exemple Ecore comme méta-métamodèle et *Eclipse Modeling Framework* (EMF) [Ste+09] comme environnement de développement. Le métamodèle d'un langage est habituellement composé de deux parties : une partie structurelle qui décrit les objets du système et leurs relations, et une partie comportementale qui permet de spécifier le comportement de ces objets. La partie comportementale repose souvent sur un langage d'action qui permet de décrire finement le comportement du système.

Il est utile de remarquer que ce métamodèle ne décrit que la partie statique du langage (et pas les concepts nécessaires à son exécution). Les modèles conçus sont ainsi indépendants de la sémantique permettant de définir leur comportement.

**Application à UML.** Le sous-ensemble retenu dans cette thèse pour la conception d'un interpréteur UML se base sur les concepts représentés par les diagrammes de classes et de structures composites pour la partie structurelle, et par les diagrammes de machines à états pour la partie comportementale.

### 8.3.2 Métamodélisation des données d'exécution

Pour définir les données d'exécution (ou *RunTime Data* (RTD)), l'approche proposée (en Figure 8.2) vise à définir, au niveau M2, une extension du métamodèle du langage de conception. On parle ici "d'extension du métamodèle" (ou de fragment) car la brique *Métamodèle$_{ext}$ RTD* fait des références au métamodèle du langage. Elle ne peut donc être qualifiée de métamodèle en tant que tel car elle n'est pas complète. En revanche, l'association du fragment *Métamodèle$_{ext}$ RTD* avec le *Métamodèle DSML* forme un métamodèle exécutable du langage de conception noté *Métamodèle xDSML*. Au niveau M1, un *Modèle DSML* se conforme au métamodèle statique du langage de conception (*Métamodèle DSML*). Une extension de ce modèle appelée *Modèle$_{ext}$ RTD* permet de définir les données d'exécution associées à ce modèle. Le tout forme alors un modèle exécutable appelé *Modèle xDSML*.

Pour un langage donné, le fragment *Métamodèle$_{ext}$ RTD* n'est pas unique. Des outils avec des paradigmes d'exécution différents peuvent nécessiter des définitions différentes de cette

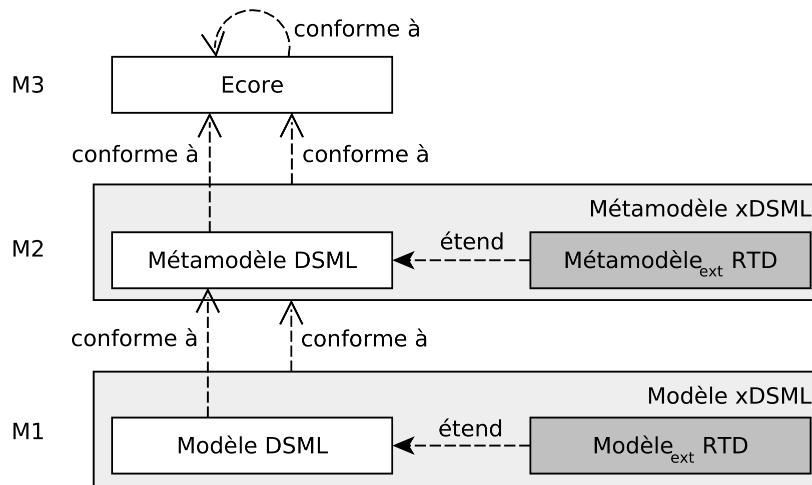



FIGURE 8.2 – Métamodélisation des données d'exécution d'un langage.

extension du métamodèle. Quelle que soit la définition considérée, cette extension du métamodèle doit définir *(i)* le métamodèle objet nécessaire pour l'instanciation des modèles conformes à ce langage et *(ii)* les données d'exécution nécessaires pour l'interprétation de ces modèles. Pour construire le fragment *Métamodèle$_{ext}$ RTD*, il semble important de comprendre comment celui est instancié pour construire le modèle objet servant à l'exécution d'un modèle.

### 8.3.3 Le modèle objet et l'instanciation

Le modèle objet est une représentation physique (ici logicielle) des objets haut niveau du modèle de conception. Il définit des objets physiques (c.-à-d. typiquement des objets de la programmation orientée objet) servant de support pour l'exécution des objets du modèle de conception sur une plateforme d'exécution. Le modèle objet permet également de définir les données d'exécution des modèles. Pour rappel, ces données d'exécution peuvent être divisées en deux catégories :

— Les attributs dynamiques issus du modèle et pour lesquels la valeur courante doit être stockée dans la mémoire de la plateforme d'exécution (p. ex. une variable d'instance) ;
— Les attributs ou les objets internes au moteur d'exécution qui sont nécessaires pour l'implémentation de la sémantique du langage (p. ex. la variable stockant la valeur du compteur de programme (ou en anglais *program counter*) qui indique l'instruction actuellement exécutée).

Étant donné ces éléments, il apparaît évident que les concepts du fragment *Métamodèle$_{ext}$ RTD* définissant le métamodèle du modèle objet ne doivent pas être définis directement dans le *Métamodèle DSML* pour au moins deux raisons. *(i)* Pour définir un modèle, les éléments de

la syntaxe abstraite doivent être associés aux éléments de la syntaxe concrète. Cependant, avoir les données d'exécution (p. ex. le compteur de programme) dans la syntaxe concrète n'a pas de sens car le modèle est défini de façon statique. *(ii)* Différentes plateformes d'exécution peuvent nécessiter différentes relations d'instanciation et chacune de ces relations peut être réutilisée pour d'autres langages.

On distingue deux grands types de relations d'instanciation [AK03 ; BL98] :

— L'instanciation logique (ou instanciation linguistique) qui permet de passer d'un niveau de modélisation au niveau inférieur (p. ex. de M2 à M1) ;
— L'instanciation physique (ou instanciation ontologique) qui permet de passer des concepts d'un langage aux concepts physiques d'un domaine donné.

Cette seconde relation d'instanciation est justement celle qui est employée dans notre approche pour construire le modèle objet qui servira à l'exécution du modèle de conception. Il existe cependant différentes relations d'instanciation physique [BL98] car chaque domaine définit des concepts qui lui sont propres. Dans cette thèse, les objets du langage de modélisation sont instanciés en objets informatiques pouvant être déployés sur une plateforme d'exécution.

La Figure 8.3 présente l'architecture de modélisation utilisée pour construire le modèle objet d'un modèle de Train à Grande Vitesse (TGV). Pour illustrer les concepts de façon générique, on considère ici que le modèle du TGV est conçu dans un DSML quelconque. Le métamodèle de ce langage (*Métamodèle DSML*) est conforme à *Ecore*. Pour faciliter la compréhension, seul le concept de *Class* est représenté dans le métamodèle du DSML considéré. Par la relation d'instanciation logique, *Class* est une instance de *EClass* et le *Modèle DSML* possède une classe *Train* instance de *Class*. Toujours pour simplifier l'illustration de cet exemple, on considère ici que le *Modèle DSML* possède une seule instance par *Class*[1]. Ici, l'instance de la classe *Train* représente un TGV. Néanmoins, le niveau M0 n'étant pas un niveau de modélisation, il n'est pas possible de modéliser cette instance pour définir les données dynamiques nécessaires à son exécution.

Pour y parvenir, on utilise le mécanisme de promotion [JB06] qui permet de remonter le modèle objet au niveau M1 et donc d'en faire un modèle au sens de l'IDM. L'extension du modèle permettant de représenter ce modèle objet est *Modèle$_{ext}$ RTD*. Il définit les instances physiques du modèle qui seront utilisées pour l'exécution. De même, les concepts de modélisation de ce modèle objet sont promus au niveau M2 sous la forme du fragment *Métamodèle$_{ext}$ RTD* qui étend le métamodèle du DSML. Le fragment *Métamodèle$_{ext}$ RTD* définit ici le concept de *PhysicalInstance* pour pouvoir les représenter. *PhysicalInstance* représente l'instance d'une *Class* par la relation d'instanciation physique. À titre d'exemple, une donnée d'exécution appelée *programCounter* a également été ajoutée pour représenter le compteur de programme de chaque instance et ainsi pouvoir implémenter la sémantique du langage. Au niveau M1, il devient ainsi

1. Cette contrainte sera levée par la suite.

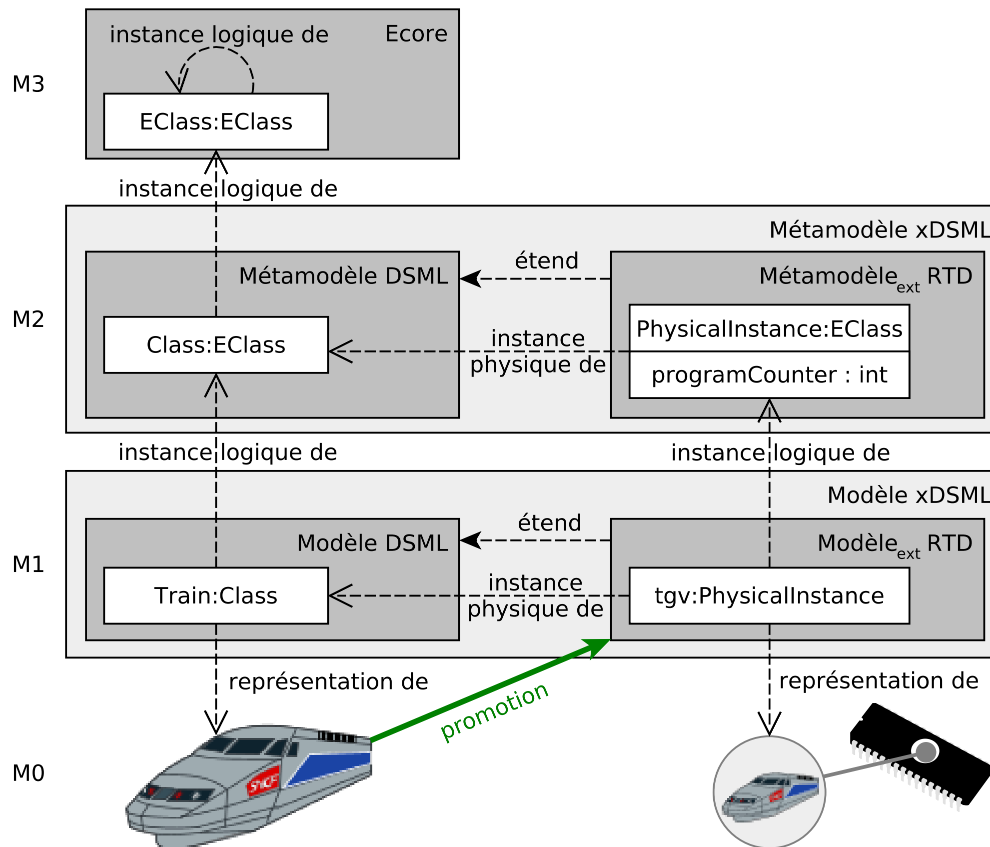



FIGURE 8.3 – Architecture de modélisation utilisée pour construire un modèle objet.

possible de modéliser l'instance de la classe *Train*. Celle-ci est en fait une instance de *PhysicalInstance* construite à partir de la classe *Train* via la relation d'instanciation physique. En pratique, la relation d'instanciation physique permettant de passer du *Modèle DSML* aux objets du *Modèle$_{ext}$ RTD* doit être renseignée par l'ingénieur du langage car elle fait partie intégrante de sa sémantique. Le modèle exécutable (*Modèle xDSML*) possède ainsi le modèle statique, les instances du modèle et les données d'exécution qui leur sont associées.

Concrètement, chaque instance de *PhysicalInstance* possède les attributs d'exécution définis par cette classe ainsi qu'une référence vers le type de l'objet modélisé (ici *Train*). Pour permettre d'instancier plusieurs objets pour chaque *Class* (p. ex. plusieurs trains), il suffit simplement que chaque instance de *PhysicalInstance* possède également une référence pointant sur la propriété du *Modèle DSML* représentant l'objet devant être instancié.

En résumé, cette architecture permet ainsi d'avoir quatre niveaux de modélisation. Ecore définit la relation d'instanciation (logique) permettant de passer de M3 à M2 et de M2 à M1. Pour le passage à M0, c'est la sémantique du langage de conception qui définit comment faire

l'instanciation (et non plus celle d'Ecore). Le mécanisme de promotion permet de remonter le modèle objet dans un espace de modélisation et d'instancier les objets physiques servant à l'exécution via une relation d'instanciation physique.

**Application à UML.** Un exemple de métamodèle$_{ext}$ RTD pour le langage UML est donné sur la Figure 8.4. Un interpréteur (*Interpreter*) contient l'ensemble des objets (*Object*) dynamiques du modèle. Par analogie à une *PhysicalInstance*, chaque *Object* possède une *part* et un *type* indiquant respectivement l'objet instancié par instanciation physique et sa classe. Un *Object* peut être soit passif (*PassiveObject*) soit actif (*ActiveObject*). Un *PassiveObject* possède uniquement des attributs dynamiques modélisés par *InstanceSlot* (c.-à-d. des variables d'instance). Un *ActiveObject* possède également un *currentState* représentant l'état courant de sa machine à états et un *eventPool* permettant de stocker les évènements (*Event*) envoyés par les autres objets du système.

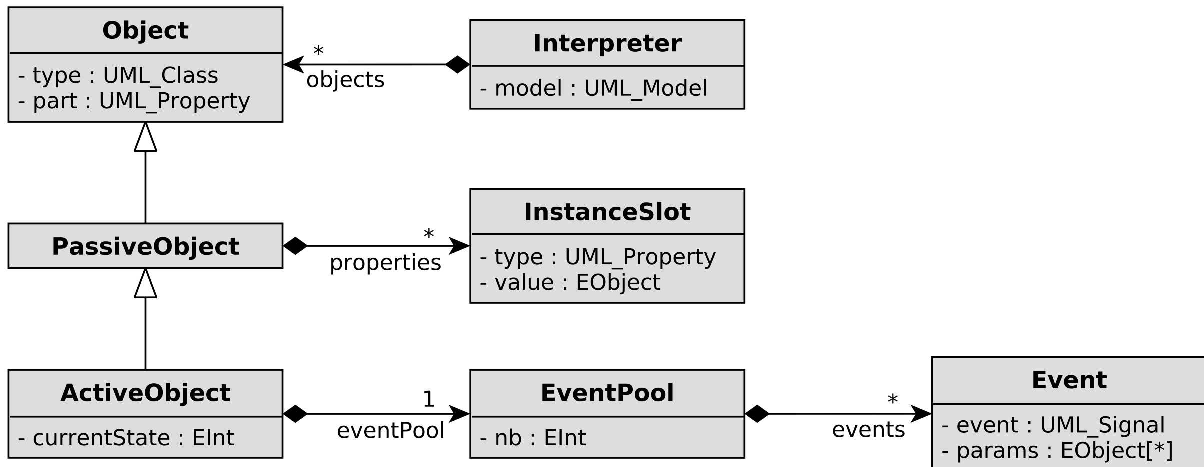



FIGURE 8.4 – Exemple simplifié de métamodèle$_{ext}$ RTD pour le langage UML.

### 8.3.4 La notion de configuration

La métamodélisation des RTD permet de définir un autre élément clé : la configuration. Cette notion est particulièrement importante dans la conception d'un interpréteur pilotable. On appelle configuration l'ensemble des données d'exécution d'un modèle. Dans la littérature, la configuration porte aussi les noms d'état d'exécution [Bou+16] ou de contexte d'exécution. Dans cette thèse, le terme configuration sera privilégié en référence à la terminologie du *model-checking* [BK08].

La configuration regroupe à la fois les attributs dynamiques issus du modèle et ceux utilisés en interne par l'interpréteur pour implémenter la sémantique. Tous les attributs modifiables et

qui ont besoin d'être persistants au cours de l'exécution du modèle sont contenus dans la configuration. En revanche, elle ne contient pas les attributs du modèle en lecture seule qui sont des constantes du point de vue de l'exécution et les informations dérivées c.-à-d. les informations qui peuvent être recalculées à partir des données de la configuration.

Les attributs dynamiques contenus dans la configuration sont tous modélisés dans l'extension du métamodèle définissant les RTD. Il est évident que le contenu exact de la configuration dépend à la fois du langage de modélisation et des besoins de la sémantique. Ces travaux de thèse ont néanmoins permis de caractériser les éléments de la configuration qui sont indispensables pour rendre un interpréteur pilotable. En effet, la configuration doit contenir tous les éléments permettant de mémoriser l'état d'exécution d'un modèle à un instant donné dont :

— Un compteur de programme indiquant quelle est l'instruction courante pour chaque unité d'exécution du modèle (p. ex. pour chaque objet actif en UML) ;
— Des files ou des piles d'exécution permettant de stocker les messages (p. ex. les évènements, les appels de fonctions) devant être traités ;
— Les attributs dynamiques des objets du modèle (p. ex. les valeurs des variables d'instance d'un objet).

En règle générale, la configuration possède une taille dynamique car la création ou la suppression d'objets impacte directement le contenu de la configuration. Elle peut également avoir une taille fixe si le moteur d'exécution ne supporte pas l'allocation dynamique de mémoire ou si celui-ci se contraint à ne pas en utiliser (p. ex. pour respecter la norme MISRA [2]). Dans ce cas, tous les objets du modèle sont connus statiquement et la configuration garde une taille fixe tout au long de l'exécution du modèle.

De plus, le coût des opérations d'accès en lecture ou en écriture à la configuration peut être élevé car *(i)* la configuration contient potentiellement beaucoup de données et *(ii)* ces opérations peuvent être appelées de nombreuses fois, par exemple pour explorer l'espace d'état du modèle. Plusieurs optimisations sont possibles pour réduire le coût des accesseurs et des mutateurs de la configuration. Par défaut, pour récupérer la configuration, il faut visiter un à un chacun des objets possédant des données d'exécution dynamiques (p. ex. avec un visiteur [EHV95]). Pour une configuration à taille fixe, une première optimisation est de stocker tous les éléments de la configuration de façon contiguë en mémoire. La configuration peut alors facilement être manipulée sous forme d'un tableau d'octets. Une seconde optimisation est de récupérer les changements effectués sur la configuration sous la forme d'une liste de différences par rapport à une version plus ancienne.

**Application à UML.** Pour un modèle UML, la configuration contient pour chaque objet la valeur de l'ensemble de ses attributs dynamiques (cf. Figure 8.4). Pour chaque objet passif, elle contient la valeur courante (*value*) de chaque *properties*. Pour chaque objet actif, elle contient

2. MISRA : `https://www.misra.org.uk/`.

également l'état courant de sa machine à états (*currentState*), le nombre d'évènements reçus dans son *event pool* (*nb*) et son contenu (*events*). En revanche, elle ne contient ni la *part* et le *type* de chaque *Object* ni le *type* de chaque *InstanceSlot* qui sont des attributs en lecture seule et donc qui ne changent pas au cours de l'exécution.

### 8.3.5 Sérialisation des métamodèles

Une fois que le métamodèle du DSML et que le métamodèle$_{ext}$ RTD ont été modélisés, il est possible de les sérialiser dans le langage hôte, c.-à-d. le langage d'implémentation de la sémantique. Cette étape va permettre de fournir tous les types de ces deux métamodèles dans l'environnement de développement natif de l'interpréteur. En plus des types qui sont mis à disposition, cela offre de nombreuses facilités de développement comme l'autocomplétion ou la coloration syntaxique.

## 8.4 Implémentation de la sémantique opérationnelle

La sémantique opérationnelle du langage de conception définit l'ensemble des règles permettant d'associer un comportement à chaque concept du langage. Son implémentation est réalisée directement à l'aide des concepts du langage hôte (p. ex. avec des fonctions). Dans le cadre de cette thèse, la sémantique doit non seulement être pilotable par un ordonnanceur embarqué mais elle doit aussi être utilisable pour mener des activités de vérification formelle.

### 8.4.1 Implémentation d'une sémantique pilotable

L'implémentation de la sémantique opérationnelle d'un interpréteur EMI est d'autant plus importante que cette implémentation sera la seule utilisée par toutes les activités d'analyse et d'exécution de modèles. La sémantique doit être définie de façon à pouvoir être pilotée à travers l'interface EMI (ou STR). Pour y parvenir, les ingénieurs peuvent s'appuyer sur le modèle objet afin de faire le lien entre le modèle statique et les données d'exécution (RTD). Pour faciliter l'implémentation de la sémantique d'un langage, différents patrons de conception ont été définis dans la littérature (p. ex. Interpréteur [EHV95], Visiteur [EHV95], Revisiteur [Led+17], Object Algebras [OC12], Kermeta [Jéz+15], ALE [3], Trivially [WO16]). En théorie, l'approche proposée dans cette thèse reste compatible avec les patrons existants à condition que *(i)* le langage hôte le permette (c.-à-d. que certains concepts de programmation soient disponibles dans le langage hôte) et que *(ii)* le patron puisse être modifié pour définir une sémantique pilotable.

3. Action Language for EMF : `http://gemoc.org/ale-lang/`.

Pour répondre à ce second point, ces travaux de thèse se sont intéressés à caractériser les éléments nécessaires à la définition d'une sémantique pilotable. Un des premiers éléments à déterminer (comme dans toute sémantique) est la granularité des pas d'exécutions observables c.-à-d. les concepts qui seront utilisés comme pas d'exécution dans la sémantique du langage. L'interpréteur de modèles doit permettre d'évaluer les pas d'exécution observables pour savoir lesquels sont exécutables à un instant donné mais aussi d'exécuter ces pas. Pour que la sémantique du langage soit pilotable, ces opérations ne doivent reposer ni sur le contexte d'exécution ni sur le flot de contrôle du langage hôte. En effet, entre chaque pas d'exécution, la configuration courante de l'interpréteur doit être mise à jour de façon à pouvoir être capturée puis réinjectée a posteriori dans l'interpréteur afin de reprendre l'exécution du modèle dans ce même état. De plus, en réponse à une sollicitation de l'environnement, une réaction en chaîne peut déclencher l'exécution de plusieurs pas d'exécution. Ces pas d'exécution doivent être vus comme une séquence de pas d'exécution observables et non pas comme un seul grand pas d'exécution. En partant ce ces observations, on peut conclure qu'une sémantique pilotable nécessite :

— D'utiliser la configuration du moteur d'exécution comme contexte d'exécution ;
— De casser le flot de contrôle pour isoler chaque pas d'exécution observable.

**Application à UML.** Pour le sous-ensemble d'UML choisi, la sémantique est décrite dans les standards UML [OMG17d], *Precise Semantics of UML Composite Structures* (PSCS™) [OMG19] et *Precise Semantics of UML State Machines* (PSSM) [OMG17b]. Pour l'exécution des machines à états UML, les pas observables sont habituellement les pas de *Run-To-Completion* [OMG17d]. Un même pas d'exécution observable peut donc contenir l'exécution de plusieurs transitions UML notamment lorsque des pseudo-états (p. ex. *choice*) sont utilisés.

À partir de ces considérations, il est possible d'implémenter la sémantique d'UML en respectant les contraintes permettant de la rendre pilotable. Il n'est cependant pas possible de proposer une architecture logicielle générique pour l'implémentation de la sémantique d'UML car cela dépend fortement du patron d'implémentation utilisé et des contraintes des applications visées (p. ex. allocation statique ou dynamique, performances d'exécution). À titre d'exemple, notre contribution en [Bes+18c] présente une architecture permettant l'implémentation d'un interpréteur de modèles UML.

### 8.4.2 Une sémantique utilisable en vérification formelle

Dans cette thèse, un autre point important est que la sémantique du langage de conception doit également pouvoir être utilisée pour faire de la vérification formelle. Cependant, certains langages sont semi-formels car leurs spécifications possèdent des points de variation sémantique ou des comportements indéterminés. À première vue, il semble donc difficile de pouvoir appliquer des techniques formelles comme le *model-checking* sur des modèles conformes à

ces langages. Néanmoins, avec l'approche proposée, la même implémentation de la sémantique opérationnelle est utilisée pour l'exécution et les activités d'analyse. En cas de points de variation sémantique ou de comportements indéterminés, c'est l'implémentation de la sémantique qui impose le choix et fait office de référence. Ainsi même un langage semi-formel peut désormais être utilisé pour faire de la vérification formelle car les outils se basent non pas sur la spécification semi-formelle de ce langage mais sur une implémentation (déterministe) de sa sémantique faisant office de référence pour l'analyse et l'exécution.

**Application à UML.** UML est un langage semi-formel car il possède des points de variation sémantique. Par exemple, l'ordre de traitement des évènements, ou encore l'interprétation de la réception d'un évènement qui ne correspond à aucun *trigger* valide sont des points non-définis par le standard UML. L'implémentation de la sémantique UML doit faire un choix sur la technique d'implémentation de l'*event pool* afin de rendre déterministe le traitement des évènements. Étant donné que plusieurs solutions peuvent être mises en place, il est possible de rendre l'implémentation de la sémantique d'un langage paramétrable afin de proposer différents choix d'implémentation pour un même point de variation sémantique [Bes+18c]. Ce mécanisme permet ainsi de s'adapter au mieux aux préférences des utilisateurs. Néanmoins, ces choix doivent être appliqués à la fois pour l'analyse et l'exécution embarquée du modèle afin de garantir l'unicité de la sémantique du langage.

## 8.5 Chargement du modèle

Une fois que la syntaxe et la sémantique du langage de modélisation ont été définies, il devient possible de concevoir des modèles conformes à ce langage. Ces modèles peuvent être conçus avec une syntaxe textuelle ou graphique et sont généralement sauvegardés sous le format standard XMI. Pour pouvoir les exécuter avec un interpréteur de modèles embarqué, une étape importante est de charger ces modèles dans la mémoire de la plateforme d'exécution. Étant donné que cette approche vise des applications embarquées (et potentiellement sans OS), l'usage d'un chargeur XMI classique est trop coûteux. Un mécanisme davantage adapté à nos besoins et permettant de réduire l'empreinte mémoire est ici nécessaire pour charger en mémoire une représentation binaire du modèle. Ce chargement comprend trois étapes : *(i)* la sérialisation du modèle statique, *(ii)* l'instanciation physique permettant de créer le modèle objet (c.-à-d. le modèle$_{ext}$ RTD) et *(iii)* la compilation en code binaire exécutable.

**Sérialisation.** Pour la sérialisation du modèle statique, les éléments du modèle au format XMI sont sérialisés vers les concepts du langage hôte (p. ex. vers des initialisateurs de structure). Contrairement à la génération de code qui génère à la fois les données et le programme (c.-à-d. le code exécutable capturant la sémantique), la sérialisation n'effectue une conversion que

des données représentant la partie statique du modèle. Comme expliqué dans la section 8.4, le code source correspondant à la sémantique du langage est directement exprimé dans le langage hôte. L'utilisation de cette technique de sérialisation peut néanmoins nécessiter de réimplémenter certains concepts de programmation non supportés par le langage hôte (p. ex. l'héritage multiple, le *downcasting*) afin de pouvoir prendre en compte tous les aspects du langage de modélisation.

**Instanciation.** Concernant l'instanciation du modèle, deux possibilités peuvent être envisagées pour créer le modèle objet des RTD et les données d'exécution. L'instanciation *dynamique* permet de créer le modèle objet via allocation dynamique lors de l'initialisation de l'interpréteur sur la plateforme d'exécution. Le modèle est alors exploré dynamiquement afin de créer les instances physiques des objets du modèle. L'instanciation *statique* permet quant à elle de créer le modèle objet des RTD avant son déploiement sur la cible embarquée. L'exploration du modèle est alors basée sur une analyse statique. Dans ce cas, l'instanciation peut être réalisée avant la sérialisation de la partie statique du modèle de façon à ce que la sérialisation génère aussi les données correspondant au modèle$_{ext}$ RTD. Cette méthode est moins coûteuse que l'instanciation dynamique et tend à respecter la norme MISRA qui interdit les allocations dynamiques dans la conception des systèmes embarqués. La méthode d'instanciation statique est celle qui a été utilisée dans ces travaux de thèse.

**Compilation.** À partir du métamodèle xDSML, de l'implémentation de la sémantique, et du modèle DSML (et éventuellement du modèle$_{ext}$ RTD si instanciation statique), un outil de compilation peut alors être utilisé pour générer le code binaire exécutable (ou le *bytecode*) contenant à la fois la sémantique du langage et le modèle exécutable.

Grâce à cette technique de chargement, le modèle est instancié et chargé une seule fois dans la mémoire de la plateforme d'exécution au lieu d'une fois par outil (e.g., interpréteur, débogueur, *model-checker*). Le modèle chargé en mémoire est donc celui qui fait foi pour l'exécution embarquée et les activités d'analyse.

## 8.6 Déploiement sur la plateforme d'exécution

Une fois le code exécutable de l'interpréteur EMI produit, celui-ci peut être déployé soit sur une cible embarquée soit sur un PC de développement. Le déploiement peut être fait de deux façons :

— En *bare-metal* (c.-à-d. sans OS) : l'interpréteur interagit directement avec le hardware de la cible embarquée (valable uniquement pour le déploiement sur une cible embarquée) ;

— Sur un OS (p. ex. Linux) : l'interpréteur utilise les services de l'OS et la couche d'abstraction matérielle qu'il fournit (valable pour le déploiement sur une cible embarquée ou un PC de développement).

Pour interagir avec l'environnement réel, une API de haut niveau permet de communiquer avec les périphériques d'entrées/sorties de la plateforme d'exécution. Cette API permet de récupérer les informations fournies par les capteurs mais aussi de piloter les actionneurs. Elle est indépendante des spécificités du matériel ce qui permet à un modèle donné d'être facilement déployable sur de multiples cibles embarquées. Pour chaque type de carte, une implémentation de cette API doit alors être fournie pour s'adapter au firmware de chaque plateforme d'exécution.

**Application à UML.** Dans le contexte d'UML, nous avons mis en place une solution pour le déploiement de modèles dans notre contribution en [Bes+20]. La modélisation du système à l'étude est réalisée dans plusieurs fichiers représentant le système, les interfaces et signaux échangés, et l'environnement. Un dernier fichier décrit la structure composite principale, appelée *Main*, permettant de connecter ensemble le système et son environnement (réel ou abstrait). Chaque fichier peut faire des références externes vers d'autres fichiers grâce au concept d'*ElementImport* d'UML et à des identifiants (IDs) XMI “stables”. Le nom qualifié des objets est utilisé comme XMI IDs à la place des XMI IDs usuels qui ne sont pas lisibles par l'homme. Ces XMI IDs sont dits “stables” car ils peuvent être partagés par plusieurs fichiers.

Cette technique permet ainsi de concevoir des modèles UML modulaires dans lesquels le modèle d'environnement est interchangeable. En effet, une abstraction de l'environnement réel, utilisée en vérification formelle, peut facilement être remplacée par un modèle d'environnement décrivant les liens avec les périphériques de la plateforme d'exécution. Ainsi, exactement la même modélisation du système est utilisée pour toutes les activités d'analyse et pour l'exécution sur une cible embarquée.

## 8.7 Synthèse

Dans ce chapitre, une méthodologie de conception permettant de construire un interpréteur EMI pour un langage de modélisation donné a été proposée. Avec cette méthodologie, un interpréteur EMI satisfait aux exigences identifiées dans les chapitres précédents. L'interpréteur de modèles peut être déployé sur une cible embarquée et la sémantique opérationnelle qu'il implémente est pilotable et déterministe ce qui permet de l'utiliser à la fois pour l'exécution réelle et les activités d'analyse.

Avec l'architecture d'analyse et l'implémentation de l'interface STR (cf. chapitre 7), l'exécutable produit peut être utilisé pour mettre en œuvre toutes les techniques d'analyse du chapitre 5 (c.-à-d. simulation interactive, débogage multivers, détection de *deadlocks*, *model-*

*checking* et *monitoring*) et les opérateurs du chapitre 6 pour ordonnancer l'exécution du modèle. Pour illustrer ces propos, des détails supplémentaires ont été donnés pour appliquer cette méthodologie au langage UML. Ce même langage va maintenant servir d'exemple pour montrer comment les techniques de vérification formelle peuvent être mises en œuvre de façon plus spécifique sur UML à l'aide d'un interpréteur EMI.

CHAPITRE 9

# UNIFICATION DU MODEL-CHECKING ET DU MONITORING DE SPÉCIFICATIONS UML EXÉCUTABLES

## Sommaire


**9.1 Introduction** . . . . . . . . . . **141**
**9.2 Expression de propriétés dans le langage de modélisation** . . . . . . . . . . **142**
9.2.1 Modélisation de propriétés en UML . . . . . . . . . . 142
9.2.2 Modélisation des automates de Büchi en UML . . . . . . . . . . 143
9.2.3 Modélisation des automates observateurs en UML . . . . . . . . . . 145
9.2.4 Conversion des propriétés LTL en PUSMs . . . . . . . . . . 146
**9.3 Composition synchrone** . . . . . . . . . . **148**
9.3.1 Mise en œuvre de la composition synchrone . . . . . . . . . . 148
9.3.2 Ajout de transitions implicites . . . . . . . . . . 148
9.3.3 Composition synchrone avec un automate de Büchi . . . . . . . . . . 150
9.3.4 Composition synchrone avec un automate observateur . . . . . . . . . . 150
**9.4 *Model-checking* et *monitoring* avec des machines à états UML** . . . . . . . . . . **150**
9.4.1 Architecture de *model-checking* avec des PUSMs . . . . . . . . . . 151
9.4.2 Architecture de *monitoring* avec des PUSMs . . . . . . . . . . 152
**9.5 Synthèse** . . . . . . . . . . **154**


*Ce chapitre est partiellement adapté de notre contribution en [Bes+19d].*

## 9.1 Introduction

Pour vérifier et monitorer l'exécution de modèles, les ingénieurs se heurtent à deux problèmes principaux. (*i*) D'une part, les langages utilisés pour spécifier des propriétés formelles sont généralement différents des langages utilisés pour modéliser des systèmes. L'expression de propriétés dans un langage formel est donc une tâche complexe pour les ingénieurs sans

expertise dans ce domaine. (*ii*) D'autre part, les automates des propriétés utilisés en *model-checking* ne peuvent pas être réutilisés tels quels pour monitorer l'exécution du système. Des techniques de transformation de modèles et d'instrumentation de code doivent généralement être utilisées pour déployer ces automates sur la plateforme d'exécution réelle.

Pour apporter une solution à ces problèmes dans le cadre d'UML, la section 9.2 présente le concept de *Property UML State Machine*s (PUSMs) pour exprimer des propriétés formelles directement en UML. Chaque PUSM est interprété avec la même sémantique d'UML que le modèle du système. La section 9.3 décrit les spécificités de la composition synchrone d'un PUSM avec l'exécution du système. Enfin, la section 9.4 présente l'architecture logicielle permettant d'utiliser les PUSMs en phase de *model-checking* ou de déployer ces mêmes PUSMs sur la cible embarquée pour continuer la vérification à l'exécution (via *monitoring*).

## 9.2 Expression de propriétés dans le langage de modélisation

Une alternative explorée dans cette thèse pour exprimer des propriétés formelles est d'utiliser directement le langage de modélisation du système. Ce mécanisme est applicable à n'importe quel langage permettant d'encoder une propriété sous forme d'un modèle (ou d'un programme) pilotable via une interface STR. Les propriétés peuvent donc prendre des formes diverses et variées comme des machines à états à base de *switch/case*, des machines à états utilisant le patron de conception état [EHV95], ou des tables de transitions d'états. Pour illustrer ce mécanisme, l'application de cette technique au langage UML va maintenant être présentée.

### 9.2.1 Modélisation de propriétés en UML

Dans le contexte d'UML, les propriétés formelles sont directement exprimées en UML en utilisant le langage d'observation et le même sous-ensemble d'UML que celui utilisé pour la modélisation du système. Dans la phase de conception, chaque propriété est encodée sous forme d'un PUSM. Plus concrètement un PUSM est une instance d'une classe active UML dont le comportement est décrit avec une machine à états UML. Tous les PUSMs sont instanciés comme des *parts* d'une classe composite appelée *Prop*. Cette classe est utilisée comme structure composite racine pour l'instanciation des PUSMs. L'instance de la classe *Prop* est appelée *instProp* et est représentée dans le modèle à l'aide du concept UML d'*InstanceSpecification*.

Dans ce travail, les PUSMs sont intentionnellement conçus pour représenter des automates de Büchi ou des automates observateurs. Étant donné la définition 5.3, il est possible de définir des correspondances entre ces automates et les PUSMs. Pour les PUSMs, les étiquettes de l'alphabet sont les propositions atomiques exprimées avec le langage d'observation. Ces atomes sont utilisés pour définir les contraintes sur les gardes des transitions des PUSMs. Ces

gardes sont utilisées pour spécifier comment la machine à états associée passe d'un état à un autre afin d'atteindre potentiellement des états d'acceptation. Chaque état (respectivement transition) de l'automate de Büchi ou de l'automate observateur est modélisé par un état UML (respectivement une transition UML) dans le PUSM. L'état initial est préservé et peut facilement être identifié en UML par une transition entrante venant du pseudo-état initial de la machine à états.

Pour la vérification formelle, ces automates doivent modéliser la négation de la propriété à vérifier. À l'aide d'un opérateur de composition synchrone, ces automates sont synchronisés avec l'exécution du système afin de détecter des défaillances dans le comportement du système. Dans ce but, les machines à états UML utilisées pour les PUSMs ont besoin de définir des états d'acceptation qui sont spécifiés par des invariants d'état dans notre approche. L'utilisation d'invariants d'état généralise des approches plus spécifiques comme l'utilisation de profils UML avec un stéréotype pour les états d'acceptation. Les PUSMs utilisés pour modéliser les automates de Büchi ou les automates observateurs doivent également respecter une contrainte supplémentaire. Ils ne doivent pas interagir avec les objets du modèle du système mais seulement observer des changements dans l'état d'exécution du modèle.

Pour illustrer la modélisation de propriétés sous forme de PUSMs, nous allons considérer un système d'exclusion mutuelle, appelé Mutex, dans lequel deux processus concurrents, nommés `one` et `two`, souhaitent accéder à une ressource partagée. À chaque instant, seul l'un des processus peut avoir accès à la ressource. Pour pouvoir la prendre, chaque processus doit faire une requête (ou en anglais *request*) afin de signaler à l'autre processus son intention d'accéder à la section critique (ou en anglais *Critical Section* (CS)).

### 9.2.2 Modélisation des automates de Büchi en UML

Un premier formalisme d'automates qui peut être encodé sous forme de PUSMs est celui des automates de Büchi. En termes d'expressivité, ces automates peuvent encoder n'importe quel $\omega$-langage régulier c.-à-d. n'importe quel langage régulier avec des traces d'exécution infinies. Ces automates sont strictement plus expressifs que LTL. Avec cette approche, il est possible de définir manuellement des automates de Büchi comme des machines à états UML pour exploiter pleinement tout le potentiel d'expressivité des automates de Büchi. Ainsi, certaines propriétés qui ne pourraient pas être exprimées en LTL peuvent toujours être vérifiées par *model-checking*.

En général, les automates de Büchi possèdent du non-déterminisme (lié au langage) qui leur permet d'explorer différents chemins d'exécution simultanément. De plus, ils ne sont pas obligés d'être complets c.-à-d. que certains chemins d'exécution peuvent être intentionnellement coupés de façon à ne pas être explorés si à l'évidence aucune défaillance ne peut se produire sur ces chemins.

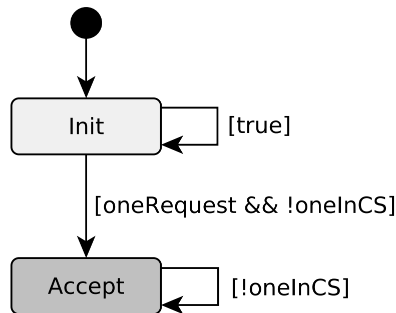



FIGURE 9.1 – Exemple d'une machine à états de PUSM représentant un automate de Büchi.

À titre d'exemple, la Figure 9.1 présente la machine à états d'un PUSM représentant un automate de Büchi pour le modèle Mutex. Il encode le fait que "Toute requête du processus `one` lui permet d'aboutir dans la section critique". Sur cette figure, l'état foncé *Accept* représente un état d'acceptation et les prédicats "`oneRequest`" et "`oneInCS`" sont des propositions atomiques exprimées à l'aide du langage d'observation (cf. annexe C). La première indique que le processus `one` a fait une requête et la seconde qu'il est dans la section critique. L'automate commence son exécution dans l'état *Init* et peut y rester indéfiniment en utilisant la transition avec la garde "`true`". Ceci permet d'explorer l'espace d'état du modèle. Dans le même temps, si le processus `one` fait une requête pour accéder à la section critique (sans y être déjà), la garde "`oneRequest && !oneInCS`" exprimée dans le langage d'observation d'EMI-UML devient vraie. Dans ce cas, l'état d'acceptation *Accept* est atteint. Si la garde "`!oneInCS`" est toujours vraie, la transition sur cet état peut être tirée et une boucle d'acceptation est détectée. La propriété est donc violée car un contre-exemple a été trouvé.

De plus, seulement un PUSM peut être synchronisé avec le système à un instant donné. En théorie, une seule propriété peut donc être vérifiée à la fois. Cependant, une astuce existe pour réaliser la vérification de $N$ propriétés, nommées $P_1$ à $P_N$, simultanément. Toutes ces propriétés peuvent être combinées en une seule propriété ($P_{all}$) en utilisant l'opérateur "et" logique (`&&` en LTL) : $P_{all} = P_1 \,\&\&\, P_2 \,\&\&\, \ldots \,\&\&\, P_N$. Un automate de Büchi pour la propriété $P_{all}$ peut ensuite être produit et utilisé pour la phase de vérification formelle. Cette technique possède néanmoins un inconvénient. En règle générale, si un contre-exemple est trouvé, l'utilisateur ne peut pas savoir quelle propriété a été violée. Pour le déterminer, il faudrait que le *model-checker* retourne également l'état d'acceptation qui a été visité infiniment souvent et que l'utilisateur comprenne l'automate de Büchi de $P_{all}$ pour savoir à quelle propriété cet état d'acceptation est associé.

### 9.2.3 Modélisation des automates observateurs en UML

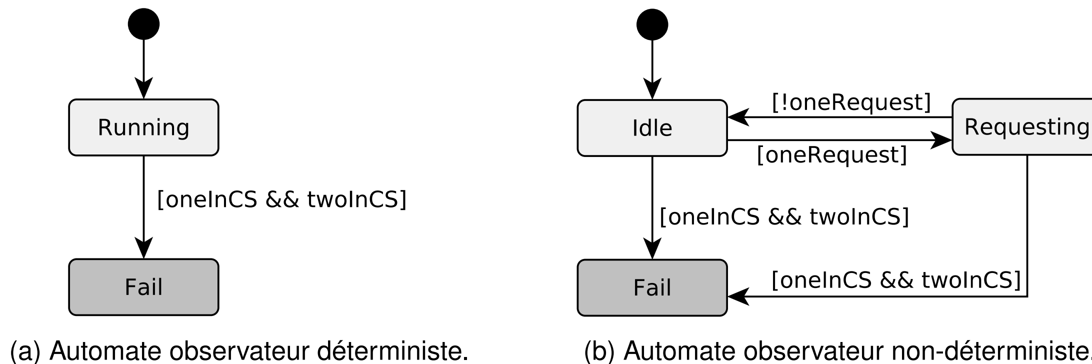
Le second formalisme étudié dans cette thèse est celui des automates observateurs, communément utilisés pour surveiller l'exécution du système. Dans ce cas, la vérification se restreint à l'analyse de la trace d'exécution courante du système par opposition au *model-checking* qui analyse toutes les traces possibles. Cette observation partielle du système limite l'expressivité des propriétés à vérifier à la classe des *propriétés monitorables* [BLS11] qui inclut notamment toutes les propriétés de sûreté. Néanmoins, cette contrainte peut aussi être vue comme un bénéfice pour la vérification hors-ligne puisque l'utilisation d'automates observateurs réduit le problème de *model-checking* à un problème d'atteignabilité. Le *model-checker* doit ainsi seulement vérifier si l'un des états d'acceptation, aussi appelés états d'échec pour un automate observateur, est atteint dans au moins une des configurations de l'espace d'état.

Pour surveiller l'exécution du système, les PUSMs utilisés pour modéliser les automates observateurs doivent satisfaire deux contraintes supplémentaires. La *contrainte de déterminisme* assure que les automates observateurs n'introduisent pas de non-déterminisme (lié au langage), mais suivent simplement l'exécution du système. Pour garantir le déterminisme, les gardes des transitions sortantes des états d'un automate observateur doivent être exclusives. La *contrainte d'exhaustivité* veille à ce que les automates observateurs ne bloquent pas l'exécution du système lorsqu'ils sont composés de manière synchrone avec celui-ci. Cette contrainte est automatiquement inférée par l'opérateur de composition synchrone, qui complète les automates observateurs avec des pas de bégaiement. En conséquence, seuls les automates déterministes et complets (c.-à-d. avec exactement une seule transition tirable à chaque instant) doivent être déployés sur la cible réelle pour surveiller l'exécution du système. Grâce à la contrainte de déterminisme, plusieurs automates observateurs peuvent être composés de façon synchrone avec le système. Ainsi, autant de moniteurs que nécessaire peuvent être utilisés simultanément sur la cible d'exécution réelle pour vérifier le comportement du système. Même dans ce cas (en opposition aux automates de Büchi), il reste assez aisé de déterminer si une propriété a été violée ou non en analysant l'état courant de l'automate observateur correspondant.

À titre d'exemple, la Figure 9.2 présente les machines à états de deux PUSMs représentant des automates observateurs pour le modèle Mutex. Ils encodent de façon différente la contrainte d'exclusion mutuelle pouvant être formulée par : "Les deux processus ne sont pas en même temps dans la section critique". Chacun d'entre eux définit un état d'échec (*Fail*), en foncé sur la figure, qui doit être atteint en cas de défaillance. Ces deux automates observateurs ne sont pas complets mais cette contrainte va être inférée pendant la composition synchrone. La contrainte de déterminisme n'est cependant vérifiée que sur l'automate en Figure 9.2a ce qui indique que seul celui-ci peut être utilisé en *monitoring*. Cet automate commence dans l'état *Running* et reste dans celui-ci jusqu'à ce que la garde "`oneInCS && twoInCS`" exprimée dans

le langage d'observation de EMI-UML devienne vraie. Dans ce cas, l'état d'acceptation *Fail* est atteint et la propriété est violée.

L'automate en Figure 9.2b [1] est non-déterministe notamment car les gardes "`!oneRequest`" et "`oneInCS && twoInCS`" ne sont pas exclusives. Ainsi, les deux transitions sortantes de l'état *Requesting* sont potentiellement tirables en même temps, ce qui est révélateur de non-déterminisme.




(a) Automate observateur déterministe.

(b) Automate observateur non-déterministe.

FIGURE 9.2 – Exemple des machines à états de PUSMs représentant des automates observateurs.

### 9.2.4 Conversion des propriétés LTL en PUSMs

Cette thèse défend fermement le fait que la vérification formelle d'un modèle doit être basée sur la sémantique exécutable du langage de modélisation capturée via un interpréteur, et non sur des transformations ou l'utilisation de langages intermédiaires. Cependant, du point de vue de la spécification des propriétés, nous ne voulons pas contraindre l'utilisateur à utiliser un langage particulier. Cette thèse fournit donc l'outil ltl4uml comme alternative à la modélisation des PUSMs.

Si notre approche offre la possibilité d'écrire manuellement des automates de Büchi en UML, la génération de ces automates en UML (ou dans n'importe quel autre formalisme) peut facilement être automatisée lorsqu'une description formelle de la propriété est donnée. Dans cet objectif, cette thèse présente un outil, appelé ltl4uml, qui peut automatiquement convertir des propriétés LTL en tUML [JD14 ; Jou+14], un formalisme textuel pour UML. Cet outil ajoute une surcouche logicielle à la librairie ltl3ba [Bab+12] qui effectue des conversions efficaces de propriétés LTL en automates de Büchi. La librairie ltl3ba a initialement été conçue par Paul Gastin et Denis Oddoux sous le nom ltl2ba dans [GO01] avant d'être améliorée par Tomáš Babiak et al. dans [Bab+12]. Notre outil ltl4uml prend en entrée la propriété LTL à vérifier.

1. Cet automate non-déterministe a été construit dans un but illustratif tel que l'ajout de l'état *Requesting* n'apporte pas de plus-value conceptuelle si ce n'est la création de non-déterminisme.

Il capture toutes les propositions atomiques exprimées dans notre langage d'observation et ajoute une négation à la propriété. La propriété résultante est envoyée à ltl3ba qui produit alors l'automate de Büchi encodant tous les comportements indésirables. Un générateur de code, réalisé avec Xtend[2], est ensuite utilisé pour générer le code tUML correspondant au PUSM représentant l'automate de Büchi correspondant.

Pour illustrer l'utilisation de ltl4uml, nous allons considérer la propriété LTL suivante :

```
"[] (|oneRequest| -> (<> |oneInCS|))"
```

où `|oneRequest|` et `|oneInCS|` sont des propositions atomiques. À partir de cette propriété LTL, ltl4uml génère un PUSM dont le code tUML est présenté en Listing 9.1. La machine à états de ce PUSM correspond à celle de la Figure 9.1.

```
class |PropertyBüchi| behavesAs SM {
  stateMachine SM { region R {
    Initial -> Init;
    Init -> Init :
      [constraint "true" is opaqueExpression =
        ‘true’
        in C;] /;
    Init -> Accept :
      [constraint "!oneInCS && oneRequest" is opaqueExpression =
        ‘(!(IS_IN_STATE(GET(ROOT_instMain, one), STATE_One_CS))
        && (GET(GET(ROOT_instMain, one), request)))’
        in C;] /;
    Accept -> Accept :
      [constraint "!oneInCS" is opaqueExpression =
        ‘(!(IS_IN_STATE(GET(ROOT_instMain, one), STATE_One_CS)))’
        in C;] /;
    initial pseudoState Initial;
  }}
}
```

Listing 9.1 – Exemple de code tUML généré à partir d'une propriété LTL.

2. Xtend : https://www.eclipse.org/xtend/.

## 9.3 Composition synchrone

La mise en œuvre de l'architecture de vérification formelle à base de PUSMs repose sur l'opérateur de composition synchrone introduit dans la section 5.5. La présente section décrit les particularités de son application sur des PUSMs représentant des automates de Büchi ou des automates observateurs.

### 9.3.1 Mise en œuvre de la composition synchrone

Notre opérateur de composition synchrone est mis en application au sein d'un composant d'*Exécution synchrone* de modèles, illustré sur la Figure 9.3, pouvant être utilisé pour la simulation, le *model-checking* et le *monitoring*. Ce composant prend en entrée le modèle UML du *Système* et celui de la *Propriété* qui contient un PUSM. Le modèle du système est conçu à partir de ses spécifications alors que le modèle de la propriété est défini à partir des exigences logicielles. Chaque modèle est interprété par une instance d'un *Interpréteur EMI*. Ainsi, l'*Interpréteur EMI* est instancié deux fois de telle sorte que la même implémentation de la sémantique du langage est utilisée pour interpréter les deux modèles. Ensuite, notre opérateur de *Composition synchrone* élabore le produit synchrone de l'exécution du modèle de la propriété avec l'exécution du modèle du système. L'*Ordonnanceur* peut également être pris en compte grâce à notre opérateur d'ordonnancement afin d'appliquer la composition synchrone sur l'exécution ordonnancée du système. Enfin, un *Contrôleur d'exécution* est utilisé pour contrôler l'exécution du produit synchrone résultant.

### 9.3.2 Ajout de transitions implicites

En utilisant cet opérateur générique de composition synchrone, différents formalismes d'automates de propriétés peuvent être supportés. Chacun a cependant ses propres spécificités de sorte que la composition synchrone doit être légèrement adaptée pour chaque formalisme. Un ajustement usuel est d'ajouter des transitions implicites (c.-à-d. des pas de bégaiement) soit sur l'automate du système soit sur le PUSM de façon à ce que l'automate résultant ne puisse plus avoir de *deadlocks*. Ce mécanisme est utilisé pour compléter un automate afin que son exécution ne puisse jamais se bloquer. Dans ce but, l'opérateur `add_implicit_transitions` peut être utilisé avant l'application de la composition synchrone pour compléter un automate. Le principe de cet opérateur, décrit formellement dans le Listing 9.2, est assez simple. Si certaines actions sont disponibles, ces actions sont utilisées pour l'exécution du modèle. En revanche, si un *deadlock* est détecté (c.-à-d. qu'aucune action n'est disponible ou qu'aucune des actions disponibles ne permet d'atteindre une configuration cible), une action implicite est ajoutée pour qu'un pas de bégaiement puisse être pris à partir de la configuration courante.

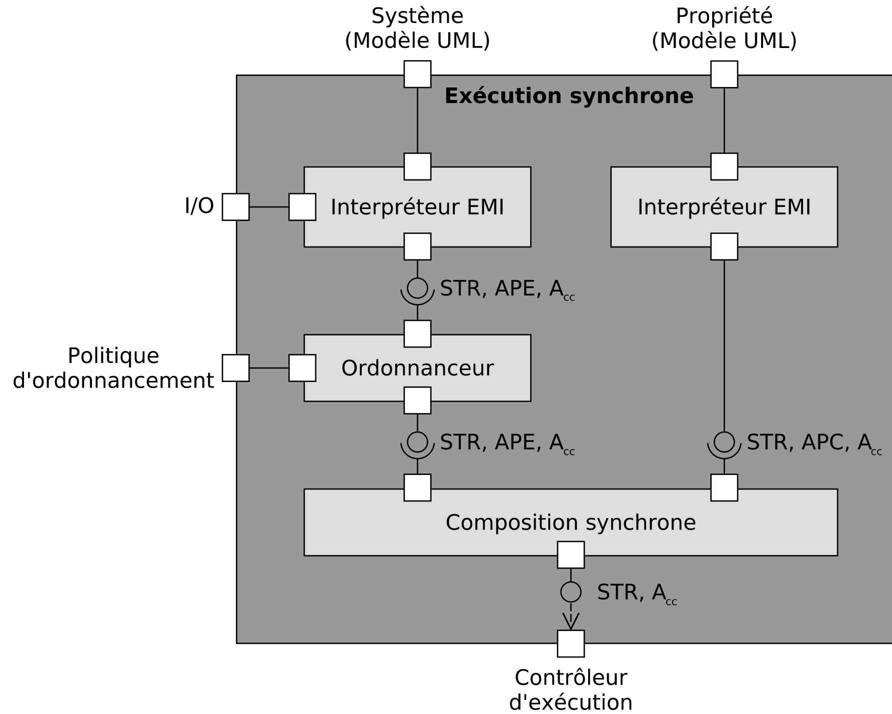



FIGURE 9.3 – Composant d'exécution de modèles avec composition synchrone.

**Définition 9.1.** La fonction `add_implicit_transitions` prend en entrée une STR et retourne une STR complétée.

```
def add_implicit_transitions (str : STR C A)
   [∀ c, decidable (str.actions c = ∅)]
   [∀ c, decidable (∀ (a : A), a ∈ str.actions c → str.execute c a = ∅)]
: STR C (completed A) := {
initial := str.initial,
actions := λ c,
   if (str.actions c = ∅) ∨ (∀ a ∈ str.actions c, str.execute c a = ∅) then
      singleton completed.deadlock
   else
      { oa | ∀ a ∈ str.actions c, oa = completed.some a },
execute := λ c oa, match oa with
   | completed.deadlock  := singleton c
   | completed.some a := { oc | ∀ t ∈ str.execute c a, oc = t }
   end }
```

Listing 9.2 – Opérateur ajoutant des transitions implicites.

Étant donné ce fondement commun, nous pouvons maintenant décrire les spécificités des deux formalismes supportés dans cette thèse : les automates de Büchi et les automates observateurs.

### 9.3.3 Composition synchrone avec un automate de Büchi

Dans le cas où un PUSM représente un automate de Büchi, plusieurs cas spécifiques doivent être gérés pour appliquer la composition synchrone.

Si aucune transition du PUSM ne peut être synchronisée avec une transition du système, la trace d'exécution correspondant à la transition du système est ignorée. Cela signifie que cette trace d'exécution n'est d'aucun intérêt pour la vérification de la propriété considérée (c.-à-d. que le *model-checker* est sûr qu'aucune violation ne peut survenir sur cette trace).

Une autre spécificité est liée au formalisme des automates de Büchi en lui-même, qui se focalise seulement sur des traces d'exécution infinies. Si l'exécution du système aboutit à un *deadlock*, la trace d'exécution est finie et aucune transition synchrone ne peut alors être construite. L'exécution du PUSM est bloquée et il risque de passer à côté de certaines défaillances identifiées par des propriétés de vivacité. Pour surmonter cette limitation, une transition implicite est ajoutée sur l'automate du système en utilisant `add_implicit_transitions`. Ainsi, une transition synchrone peut être calculée et l'exécution du modèle de la propriété peut toujours se poursuivre.

### 9.3.4 Composition synchrone avec un automate observateur

Une particularité de la composition synchrone avec un PUSM représentant un automate observateur concerne le calcul des transitions synchrones. Un automate observateur tire une transition explicite si une des transitions sortantes de son état courant est tirable. Sinon, l'automate observateur tire une transition implicite, créée en accord avec `add_implicit_transitions`, pour satisfaire la contrainte d'exhaustivité. Ainsi, les automates observateurs ne vont jamais bloquer l'exécution du système. En utilisant ce mécanisme, les transitions implicites manquantes pour rendre un automate observateur complet sont inférées automatiquement. Le concepteur de ces automates n'a donc pas besoin de s'en préoccuper durant la phase de modélisation.

## 9.4 *Model-checking* et *monitoring* avec des machines à états UML

En utilisant des PUSMs, cette section décrit le processus de vérification utilisé pour *(i)* vérifier des propriétés formelles avec un *model-checker* ou *(ii)* monitorer l'exécution du système sur une cible embarquée.

### 9.4.1 Architecture de *model-checking* avec des PUSMs

En se basant sur le composant d'*Exécution synchrone* de modèles, il devient possible de connecter un *model-checker* à un interpréteur EMI pour vérifier des propriétés encodées sous forme de PUSMs.

L'architecture logicielle utilisée pour le *model-checking* à base de PUSMs est illustrée sur la Figure 9.4 et correspond à une application de l'architecture de la section 5.4.4. Pour la vé-

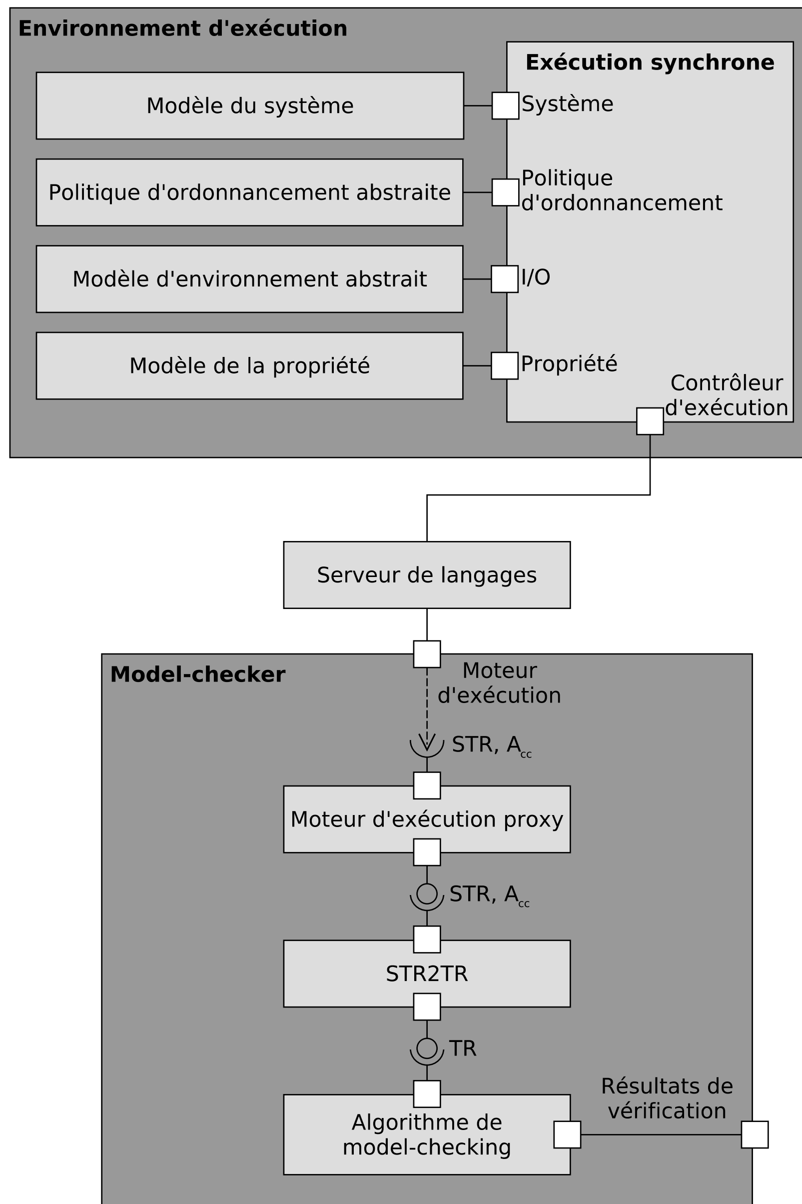



FIGURE 9.4 – Architecture utilisée pour le *model-checking* avec des PUSMs.

rification hors-ligne, le composant d'*Exécution synchrone* de modèles est connecté au *Modèle du système*, au *Modèle d'environnement abstrait*, et au *Modèle de la propriété* qui ont tous été conçus en UML. Ce composant est aussi connecté à une *Politique d'ordonnancement abstraite* pour considérer un sur-ensemble de toutes les traces d'exécution possibles. Une abstraction très générale consiste à retourner toutes les transitions tirables du système pour explorer l'espace d'état complet du modèle. Dans ce cas, l'ordonnanceur n'effectue aucun choix et la composition synchrone est appliquée à toutes les transitions tirables du système. La vérification est alors indépendante de l'algorithme d'ordonnancement utilisé. Il est également possible d'utiliser la politique d'ordonnancement réelle durant la phase de vérification (cf. section 6.4.3) si celle-ci est déterministe et connue à ce moment du cycle de développement logiciel.

Par ailleurs, le *model-checker* est connecté au composant d'*Exécution synchrone* de modèles via le *Serveur de langages*. Un *Algorithme de* model-checking permet de contrôler le processus de vérification grâce au composant *STR2TR* qui effectue la conversion d'interface appropriée et au *Moteur d'exécution proxy* qui permet de communiquer avec l'*Environnement d'exécution* à travers les interfaces exposées par le composant d'*Exécution synchrone*.

Durant l'exploration de l'espace d'état, le *model-checker* vérifie si la condition d'acceptation (cf. définition 5.3) est remplie en utilisant l'interface $A_{cc}$ pour savoir si l'exécution du PUSM est dans un état d'acceptation. Dès que la condition d'acceptation est satisfaite, le *model-checker* stoppe le processus de vérification et retourne la trace du contre-exemple trouvé. En revanche, l'espace d'état doit être complètement exploré pour assurer qu'une propriété encodée par un PUSM est vérifiée.

Étant donné cette architecture générique de vérification, différents formalismes peuvent être supportés. La seule différence est l'*Algorithme de* model-checking utilisé. Pour les automates de Büchi, nous utilisons l'algorithme de détection de boucle d'acceptation (basé sur deux DFS imbriqués) défini par Gaiser et Schwoon dans [GS09]. Pour les automates observateurs, le problème de vérification se réduit à un problème d'atteignabilité (cf. section 9.2.3). Par conséquent, un algorithme d'atteignabilité est suffisant pour contrôler l'exécution du processus de vérification.

### 9.4.2 Architecture de *monitoring* avec des PUSMs

Une fois que la vérification hors-ligne du modèle a été effectuée, le modèle UML peut être déployé sur une cible embarquée. Pour continuer la vérification des propriétés monitorables à l'exécution, il est possible d'embarquer tous les PUSMs représentant des automates observateurs déterministes. Contrairement au *model-checking* qui vérifie le modèle hors-ligne avec un environnement abstrait, le *monitoring* permet la vérification du système en ligne dans un environnement réel.

L'architecture logicielle utilisée pour le *monitoring* est illustrée sur la Figure 9.5 et corres-

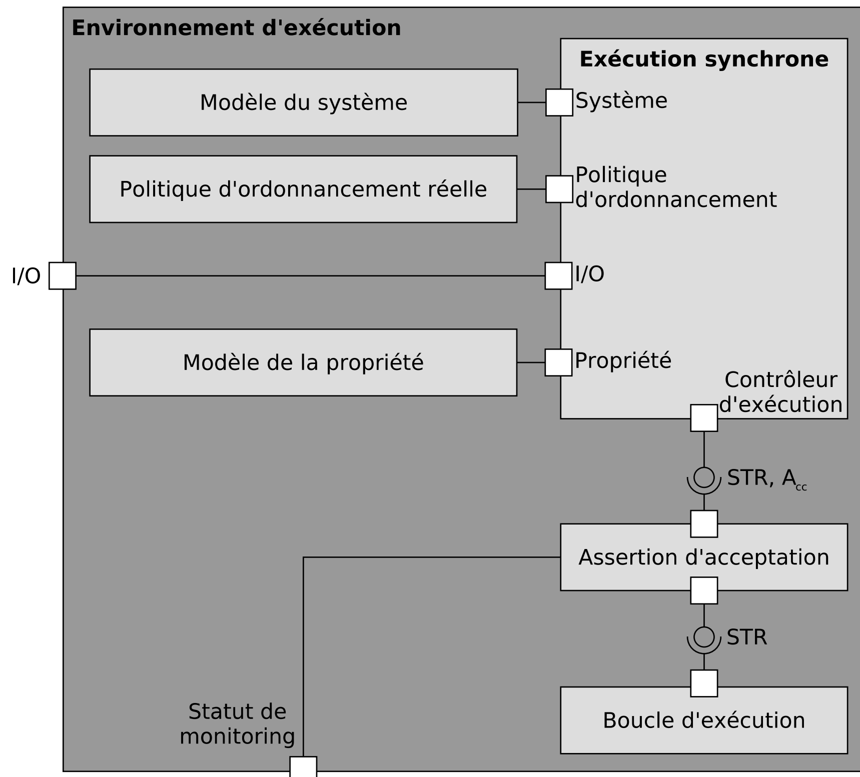



FIGURE 9.5 – Architecture utilisée pour le *monitoring* avec des PUSMs.

pond à une application de l'architecture en section 5.4.5. Le composant d'*Exécution synchrone* de modèles utilisé en *model-checking* est réutilisé pour le *monitoring* avec les mêmes modèles pour le système (*Modèle du système*) et la propriété (*Modèle de la propriété*). Néanmoins, cette fois le composant d'*Exécution synchrone* est relié aux entrées/sorties réelles (*I/O*) de la cible embarquée et la *Politique d'ordonnancement réelle* du système est utilisée plutôt qu'une abstraction. À chaque pas d'exécution, l'ordonnanceur sélectionne une et une seule transition à tirer parmi la liste des transitions tirables du système. Pour le *monitoring*, le choix de la prochaine transition à tirer est fait avant d'appliquer la composition synchrone (cf. section 5.5.2) afin d'éliminer les risques d'interférence sur le *monitoring* du système et garder de bonnes performances d'exécution. Pour piloter le composant d'*Exécution synchrone* de modèles, la *Boucle d'exécution* principale de la plateforme d'exécution est utilisée comme contrôleur d'exécution. À chaque pas d'exécution, le contrôleur d'exécution délègue la vérification des propriétés formelles au composant d'*Assertion d'acceptation*. Ce dernier vérifie si les automates observateurs ont atteint leurs états d'acceptation via l'interface $A_{cc}$ et met à jour le *Statut de monitoring* indiquant si une défaillance est survenue.

L'un des principaux avantages de cette approche est que les mêmes automates obser-

vateurs déterministes que ceux utilisés durant la phase de vérification formelle peuvent être déployés sur une cible embarquée. Ces automates peuvent ainsi être réutilisés sans effort (c.-à-d. sans transformation de modèles, sans réécriture et sans génération de code) pour faire du *monitoring* à l'exécution. Malgré les possibilités offertes par la vérification hors-ligne, il reste utile de surveiller l'exécution du système pour plusieurs raisons. Premièrement, si l'abstraction de l'environnement utilisée pour le *model-checking* est incomplète ou mal définie, il est possible que tous les cas d'exécution réels n'aient pas été couverts et que la vérification soit passée à côté d'une erreur de conception. Deuxièmement, à cause des problèmes d'explosion de l'espace d'état, il n'est pas toujours possible de vérifier des propriétés monitorables par *model-checking*. Avec notre approche, la véracité de ces propriétés peut toujours être surveillée à l'exécution par *monitoring* sans avoir besoin de transformations de modèles. Un autre bénéfice est que le *monitoring* peut détecter des violations de propriétés causées par des composants hardware défectueux, ce qui n'est pas possible avec le *model-checking*. Lorsqu'une défaillance est détectée, les PUSMs peuvent simplement notifier le problème à l'utilisateur (p. ex. en affichant un message d'erreur) ou activer des mécanismes de sûreté de fonctionnement (p. ex. de recouvrement d'erreurs). Enfin, les traces des automates observateurs peuvent être utilisées en analyse post-mortem pour comprendre pourquoi une défaillance est apparue dans le système.

En ce qui concerne les limitations de notre approche, l'utilisation d'automates observateurs en *monitoring* induit un surcoût en termes d'empreinte mémoire et de performance d'exécution comme la plupart des activités de *monitoring*. Un compromis entre l'efficacité de la vérification et les performances d'exécution doit être trouvé pour chaque contexte. Un autre inconvénient est que le *monitoring* peut seulement détecter la présence d'erreurs mais ne peut pas prouver leur absence. Le *monitoring*, contrairement à la vérification formelle exhaustive, observe les pas d'exécution du système en interaction avec l'environnement réel. Par conséquent, son efficacité dépend de la couverture des défaillances fournie par les moniteurs embarqués avec le système.

## 9.5 Synthèse

La spécification de propriétés sous forme de PUSMs permet de faciliter l'expression de ces propriétés par les ingénieurs. En effet, le langage de modélisation du système est également utilisé pour modéliser les propriétés formelles tout en réutilisant le langage d'observation pour l'expression des propositions atomiques (ici les gardes des transitions des PUSMs). L'exécution des PUSMs est pilotée à travers l'interface de contrôle d'exécution et est composée de façon synchrone avec l'exécution du système. Pour la vérification hors-ligne, l'espace d'état du produit synchrone est exploré avec un *model-checker* pour vérifier que les propriétés encodées

par les PUSMs ne sont pas violées. Pour le *monitoring*, tous les automates observateurs déterministes utilisés lors de la vérification hors-ligne peuvent être déployés sur une cible embarquée sans utiliser de transformations de modèles et d'instrumentation de code. Cette technique permet ainsi d'unifier le *model-checking* et le *monitoring* de modèles UML en assurant que les propriétés utilisées pour le *monitoring* sont exactement celles utilisées pour le *model-checking*.

Troisième partie

# Évaluation de l'approche

CHAPITRE 10

# EXPÉRIMENTATIONS

## Sommaire


**10.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159**

**10.2 Présentation de l'outil EMI-UML . . . . . . . . . . . . . . . . . . . . . . . . . . 159**

**10.3 Présentation des cas d'études . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161**

10.3.1 Contrôleur de passage à niveau . . . . . . . . . . . . . . . . . . . . . . . 161

10.3.2 Interface d'un régulateur de vitesse . . . . . . . . . . . . . . . . . . . . . 162

**10.4 Mise en œuvre des activités d'analyse de modèles . . . . . . . . . . . . . 163**

10.4.1 Simulation interactive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164

10.4.2 Débogage multivers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165

10.4.3 *Model-checking* LTL et détection de *deadlocks* . . . . . . . . . . . . . 166

10.4.4 *Model-checking* avec l'ordonnanceur . . . . . . . . . . . . . . . . . . . . 169

10.4.5 *Model-checking* avec des PUSMs . . . . . . . . . . . . . . . . . . . . . . 172

10.4.6 *Model-checking* des mécanismes de sûreté de fonctionnement . . . . 176

10.4.7 Exécution embarquée et *monitoring* . . . . . . . . . . . . . . . . . . . . 177

10.4.8 Évaluation des performances d'exécution . . . . . . . . . . . . . . . . . 178

**10.5 Expérimentations avec d'autres outils . . . . . . . . . . . . . . . . . . . . . 180**

10.5.1 EMI-LOGO : un interpréteur de modèles LOGO . . . . . . . . . . . . . 180

10.5.2 AnimUML : un outil d'interprétation et d'animation de modèles UML . . 181

10.5.3 Intégration d'EMI-UML dans Gemoc Studio . . . . . . . . . . . . . . . 183

**10.6 Synthèse des expérimentations . . . . . . . . . . . . . . . . . . . . . . . . . 184**

**10.7 Contributions scientifiques . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185**

10.7.1 C#1 Contribution à l'unification de l'analyse et de l'exécution embarquée de modèles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185

10.7.2 C#2 Contribution à l'adoption des techniques de vérification formelle par les ingénieurs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187

**10.8 Synthèse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188**

## 10.1 Introduction

Après avoir exploré les possibilités offertes par l'architecture candidate, il est temps de la mettre en pratique pour évaluer son applicabilité. À ce titre, trois questions de recherche ont été formulées pour évaluer et valider l'approche EMI à travers différentes expérimentations.

**V#1** Quel est l'effort nécessaire pour concevoir un interpréteur pilotable et y connecter différents outils d'analyse ?

**V#2** L'interface de contrôle d'exécution est-elle suffisamment générique pour répondre aux besoins des différentes activités d'analyse ?

**V#3** L'approche EMI facilite-t-elle l'analyse de l'exécution des modèles ?

Pour répondre à ces questions, la section 10.2 présente l'interpréteur de modèles UML conçu pendant cette thèse comme preuve de concept principale de l'approche EMI. Cet interpréteur a notamment été mis en œuvre sur deux cas d'études de systèmes embarqués présentés en section 10.3 : un contrôleur de passage à niveau et l'interface utilisateur d'un régulateur de vitesse. Dans la section 10.4, diverses activités d'analyse sont menées sur ces deux cas d'études en utilisant notamment le *model-checker* OBP2 pour simuler, déboguer et vérifier formellement l'exécution de ces modèles.

Pour montrer la versatilité de l'approche EMI, la section 10.5 présente d'autres expérimentations avec différents interpréteurs de modèles conformes à l'approche EMI et différents outils d'analyse permettant d'éprouver l'architecture candidate et son interface de contrôle d'exécution. Une synthèse de ces expérimentations est réalisée en section 10.6 pour répondre à V#1, V#2 et V#3. Enfin, la section 10.7 met en lumière les contributions revendiquées par cette thèse en s'appuyant sur les résultats de ces expérimentations.

## 10.2 Présentation de l'outil EMI-UML

L'interpréteur EMI-UML est une implémentation de l'approche EMI pour un sous-ensemble du langage UML. Ce prototype met en œuvre la méthodologie de conception d'un interpréteur embarqué et pilotable décrite dans le chapitre 8.

L'implémentation d'EMI-UML a été réalisée en utilisant le langage C comme langage hôte car ce langage est particulièrement adapté pour l'implémentation de systèmes embarqués. Il offre notamment de bonnes performances d'exécution, une faible empreinte mémoire et consomme relativement peu d'énergie [Per+17]. De plus, il permet de s'interfacer facilement avec les périphériques des microcontrôleurs embarqués. Notre implémentation tend également à se conformer aux règles de codage MISRA qui interdisent par exemple l'allocation dynamique de mémoire pour garantir davantage de sûreté de fonctionnement.

Cet outil supporte un sous-ensemble d'Eclipse UML pouvant être représenté par les dia-

grammes de classes et de structures composites pour la partie structurelle et par des machines à états pour la partie comportementale. Il tend à se conformer aux standards PSCS [OMG19] et PSSM [OMG17b] qui précise la sémantique d'UML concernant les structures composites et les machines à états. Cet interpréteur utilise également un langage d'action pour décrire les gardes et les effets sur les transitions des machines à états UML. Le choix de ce langage dépend notamment de la finesse des pas d'exécution observables (c.-à-d. de la limite à partir de laquelle la modélisation s'arrête au profit du code exécutable). Le standard fUML [OMG17c] possède déjà un langage d'action standardisé appelé *Action Language for Foundational UML* (ALF) [OMG17a] qui est fortement lié aux activités UML. Ce langage n'a cependant pas été utilisé car les pas d'exécution observables d'EMI-UML correspondent aux transitions des machines à états UML. Les gardes et les effets des transitions ne sont pas observables et sont exécutés comme des instructions atomiques par l'interpréteur. Du code exécutable a donc été utilisé plutôt que des activités afin de réduire l'empreinte mémoire. De plus, l'implémentation de référence de ALF est en Java ce qui rend sa réutilisation plus complexe pour notre outil basé sur le langage C. C'est la raison pour laquelle le langage d'action et le langage d'observation de EMI-UML reposent sur les concepts du langage C. Des macros C permettent d'exposer les données d'exécution du modèle et d'implémenter les opérateurs nécessaires pour ces deux langages (cf.annexe C). Les concepts de ces deux langages peuvent ainsi être exécutés sans transformation par l'interpréteur de modèles.

Concernant l'implémentation de la sémantique, notre outil a dû faire des choix concernant les points de variation sémantique d'UML. Dans le sous-ensemble supporté, l'un des principaux points de variation sémantique concerne l'ordre de traitement des évènements reçus et l'interprétation de la réception d'un évènement qui ne correspond à aucun *trigger* valide. Pour apporter plus de flexibilité à l'utilisateur, l'interpréteur EMI-UML offre différentes variantes de l'implémentation de l'*event pool* :

— *FifoEventPool* se base sur une file qui supprime les évènements qui ne correspondent à aucun *trigger* valide dans l'état d'exécution courant ;
— *OrderedListDefferedEventPool* se base sur une liste ordonnée qui reporte à plus tard le traitement des évènements qui ne correspondent à aucun *trigger* valide.

Le choix de l'implémentation de l'*event pool* utilisé pour l'exécution de modèles est fait lors de la compilation de l'interpréteur.

L'interpréteur EMI-UML peut-être déployé à la fois sur un PC de développement (avec Linux) ou en *bare-metal* sur une cible embarquée STM32 discovery (CPU à un cœur cadencé à 168 MHz, 192ko RAM). Nous avons fait le choix de déployer l'interpréteur en *bare-metal* sur une cible embarquée pour montrer qu'un interpréteur EMI est portable sur une plateforme d'exécution sans OS et potentiellement fortement contrainte en termes d'espace mémoire et de performances d'exécution. L'utilisation d'un OS offre une abstraction du matériel et des

services supplémentaires (p. ex. pour gérer la mémoire, les fichiers), dont l'interpréteur peut bénéficier si la plateforme d'exécution possède un OS. Grâce au continuum, ces deux types de cibles d'exécution peuvent être utilisées en phase de conception et de vérification.

Pour mener des activités d'analyse, l'interpréteur EMI-UML est piloté à travers l'interface de contrôle d'exécution. Dans ces travaux de thèse, l'outil OBP2 a été utilisé pour analyser et vérifier l'exécution de modèles UML sur l'interpréteur EMI-UML. OBP2 est un *model-checker* permettant de vérifier des propriétés LTL à la volée sur l'espace d'état complet d'un modèle. Il offre également un environnement permettant de simuler et de déboguer l'exécution de modèles à travers une interface graphique.

Dans la suite de ce chapitre, on fait l'hypothèse que l'exécution de l'interpréteur EMI-UML est déterministe c.-à-d. qu'il ne présente pas de non-déterminisme d'exécution. Le langage hôte d'EMI-UML est le langage C avec lequel des bogues de corruption mémoire peuvent facilement être introduits par inadvertance. Ce langage possède également plusieurs comportements indéfinis [Reg10] pouvant créer du *non-déterminisme d'exécution*. Pour limiter cette probabilité, l'implémentation d'EMI-UML s'est inspirée des normes de qualité de l'embarqué et s'est appuyée sur plusieurs outils d'analyse de code (p. ex. les *sanitizers* [1] ou encore Valgrind [2]). On suppose donc que l'exécution de l'outil EMI-UML est déterministe.

## 10.3 Présentation des cas d'études

Pour illustrer l'utilisation de l'interpréteur EMI-UML, cette section présente deux systèmes embarqués : l'un pour contrôler un passage à niveau et l'autre pour gérer l'interface d'un régulateur de vitesse automobile. Ces deux systèmes ainsi qu'une abstraction de leur environnement ont été modélisés en UML pour servir de cas d'études à nos travaux. L'interpréteur EMI-UML a également été mis en œuvre sur de nombreux autres cas d'études dont un système de train d'atterrissage [BW14], un défibrillateur cardiaque [3] [FDD17] ou encore le cas d'étude du *workshop* MDETools 2019 [4] [Bes+19b].

### 10.3.1 Contrôleur de passage à niveau

Le passage à niveau est un système de protection utilisé à l'intersection d'une voie ferrée et d'une route pour avertir et protéger les automobilistes lors du passage d'un train. Le but est ici de modéliser le système logiciel permettant de contrôler le passage à niveau. Pour simplifier le fonctionnement de ce système, nous considérons ici une seule voie ferrée sur laquelle les

1. Sanitizers : `https://clang.llvm.org/docs/MemorySanitizer.html`.
2. Valgrind : `https://valgrind.org/`.
3. Modèle de défibrillateur : `https://github.com/Pyponou/defibrillator`.
4. MDETools'19 : `https://mdetools.github.io/mdetools19/challengeproblem.html`.

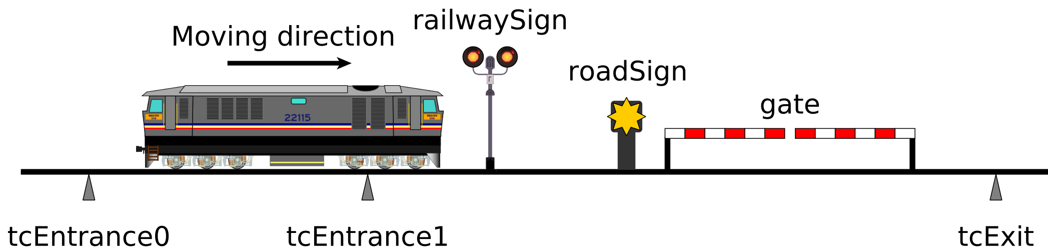



FIGURE 10.1 – Schéma d'un passage à niveau.

trains circulent à sens unique. Une illustration de ce Contrôleur de Passage à Niveau (CPN) est donnée sur la Figure 10.1.

Ce système est composé de trois capteurs, appelés *Track Circuits*, disposés le long de la voie ferrée permettant de détecter le passage d'un train : deux capteurs *tcEntrance* et un capteur *tcExit*. Il possède également un actionneur *gate* permettant d'ouvrir ou de fermer la barrière et un feu de signalisation *roadSign* permettant d'indiquer le passage d'un train aux automobilistes. Un second feu de signalisation *railwaySign* est également utilisé pour signaler au conducteur du train que la section de voie ferrée du passage à niveau peut être empruntée.

Le scénario nominal du fonctionnement de ce système est le suivant. Lorsque le train a été détecté par les deux capteurs *tcEntrance*, le contrôleur déclenche l'allumage du feu *roadSign* et la fermeture de la barrière *gate*. Le feu *railwaySign* donne ensuite l'autorisation au train d'emprunter le passage à niveau. Lorsque le capteur *tcExit* détecte le train, le passage à niveau est de nouveau libre. Cela déclenche l'ouverture de la barrière *gate* et l'extinction du feu de signalisation *roadSign*. Une description plus précise de ce cas d'étude est donnée en section D.1. Pour vérifier la fiabilité de ce système, quatre exigences ont été sélectionnées :

**E1$_{\textbf{CPN}}$** La barrière *gate* est fermée lorsque le train est sur le passage à niveau.

**E2$_{\textbf{CPN}}$** Le feu *roadSign* est allumé lorsque le train est sur le passage à niveau.

**E3$_{\textbf{CPN}}$** La barrière *gate* s'ouvre fatalement après avoir été fermée.

**E4$_{\textbf{CPN}}$** Le feu *roadSign* s'éteint fatalement après avoir été allumé.

### 10.3.2 Interface d'un régulateur de vitesse

Le régulateur de vitesse est un système permettant de maintenir la vitesse d'un véhicule à une vitesse constante, appelée vitesse de croisière, telle que définie par le conducteur. Le système agit sur l'accélération du véhicule pour effectuer la régulation. Pour ce cas d'étude, nous focaliserons les efforts d'analyse sur l'Interface utilisateur du Régulateur de Vitesse (IRV) car ce sous-système contient une grande partie de la logique de commande.

Le conducteur peut interagir avec l'interface du régulateur de vitesse à l'aide de boutons permettant de démarrer ("start") ou d'arrêter ("stop") le système, de mettre en pause ("pause") ou de reprendre ("resume") la régulation. D'autres boutons permettent d'agir sur la vitesse de croisière en lui affectant ("set") la vitesse courante du véhicule, en l'incrémentant ("inc") ou en la décrémentant ("dec") d'une unité à chaque appui. Ce sous-système prend également en compte les interactions avec les pédales d'accélération, de débrayage et de frein du véhicule. Le conducteur peut appuyer ("pedalPressed") ou relâcher ("pedalReleased") chacune de ces trois pédales. Chaque interaction du conducteur avec un bouton ou une pédale envoie un évènement au système.

L'interface du régulateur traite tous ces évènements afin d'engager ou de désengager la régulation. Pour cela, des évènements sont envoyés en sortie du système pour activer ou désactiver la boucle de régulation et lui envoyer de nouvelles consignes lorsque la vitesse de croisière change. La boucle de régulation (en dehors de l'IRV) assure ensuite la régulation en implémentant les lois d'automatique permettant d'asservir le moteur à la vitesse souhaitée. Une description plus précise de ce cas d'étude est donnée en section D.2.

Pour vérifier le fonctionnement de ce système, six exigences ont été sélectionnées :

**$E1_{IRV}$** Lorsqu'un évènement "stop" a été reçu, l'IRV va fatalement se désengager.

**$E2_{IRV}$** Lorsqu'un évènement "set" a été reçu et si le système est désengagé, l'IRV va fatalement se réengager.

**$E3_{IRV}$** Lorsqu'un évènement "pedalReleased" en provenance de la pédale d'accélération a été reçu et si la boucle de régulation peut être réactivée, l'évènement "pedalReleased" n'est pas consommé jusqu'à ce qu'un évènement "resume" soit envoyé au contrôleur du système.

**$E4_{IRV}$** Après la détection d'un évènement qui désactive la boucle de régulation et jusqu'à ce qu'un évènement contraire soit envoyé, l'IRV ne doit pas essayer d'envoyer de nouvelles consignes.

**$E5_{IRV}$** La vitesse de croisière doit être ni en dessous de 40km/h ni au-dessus de 180km/h.

**$E6_{IRV}$** Lorsque la régulation est engagée, la vitesse de croisière doit être définie.

## 10.4 Mise en œuvre des activités d'analyse de modèles

Les modèles UML de ces deux cas d'études ont été utilisés pour mettre en œuvre diverses activités d'analyse avec l'interpréteur EMI-UML comme moteur d'exécution.

### 10.4.1 Simulation interactive

Une des premières activités d'analyse qui peut être menée en connectant l'outil OBP2 à l'interpréteur EMI-UML est la simulation interactive. La Figure 10.2 présente l'interface graphique de simulation de OBP2 capturée lors d'une simulation du modèle de contrôleur de passage à niveau.

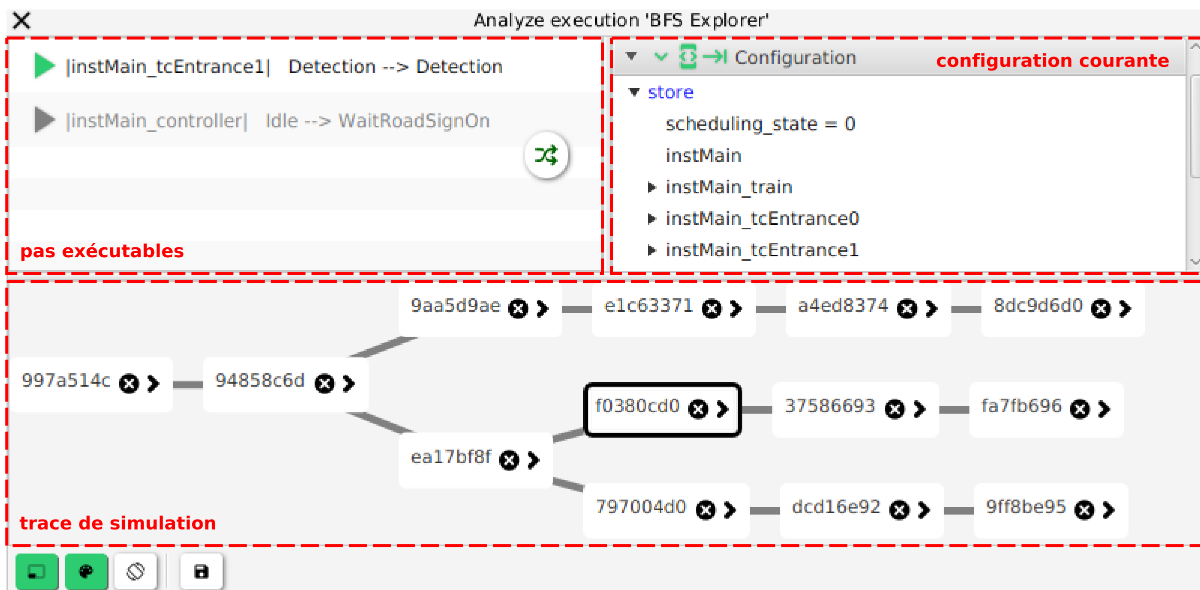



FIGURE 10.2 – Interface graphique de simulation de OBP2.

L'interface se décompose en trois parties. Elle expose à l'utilisateur la liste des pas d'exécution disponibles à partir de la configuration courante (en haut à gauche) et le contenu de la configuration (en haut à droite). Ces deux vues sont obtenues en utilisant les fonctions de projection fournies par le serveur de langages afin d'obtenir une représentation textuelle des données binaires renvoyées par l'interpréteur. L'interface graphique montre également (en bas) les traces d'exécution explorées lors de cette session de simulation. Le fait de cliquer sur un nœud du graphe permet de replacer l'interpréteur dans cette configuration et correspond à la fonctionnalité de retour en arrière (*back-in-time*). La configuration courante est celle encadrée en noir sur le graphe. À partir de cette configuration, deux transitions sont tirables. La transition du *controller* est grisée car elle a déjà été tirée pour explorer une trace d'exécution à partir de cette configuration. La transition de *tcEntrance1* n'a pas encore été exécutée mais elle peut l'être pour débuter l'exploration d'un nouveau chemin d'exécution.

L'interpréteur EMI-UML fournit également des mécanismes permettant de générer des traces d'exécution en utilisant le formalisme PlantUML[5]. La Figure 10.3 présente un *Message Sequence Chart* (MSC) illustrant l'activation du feu de signalisation *roadSign* et la fermeture

5. PlantUML : `https://plantuml.com/`.

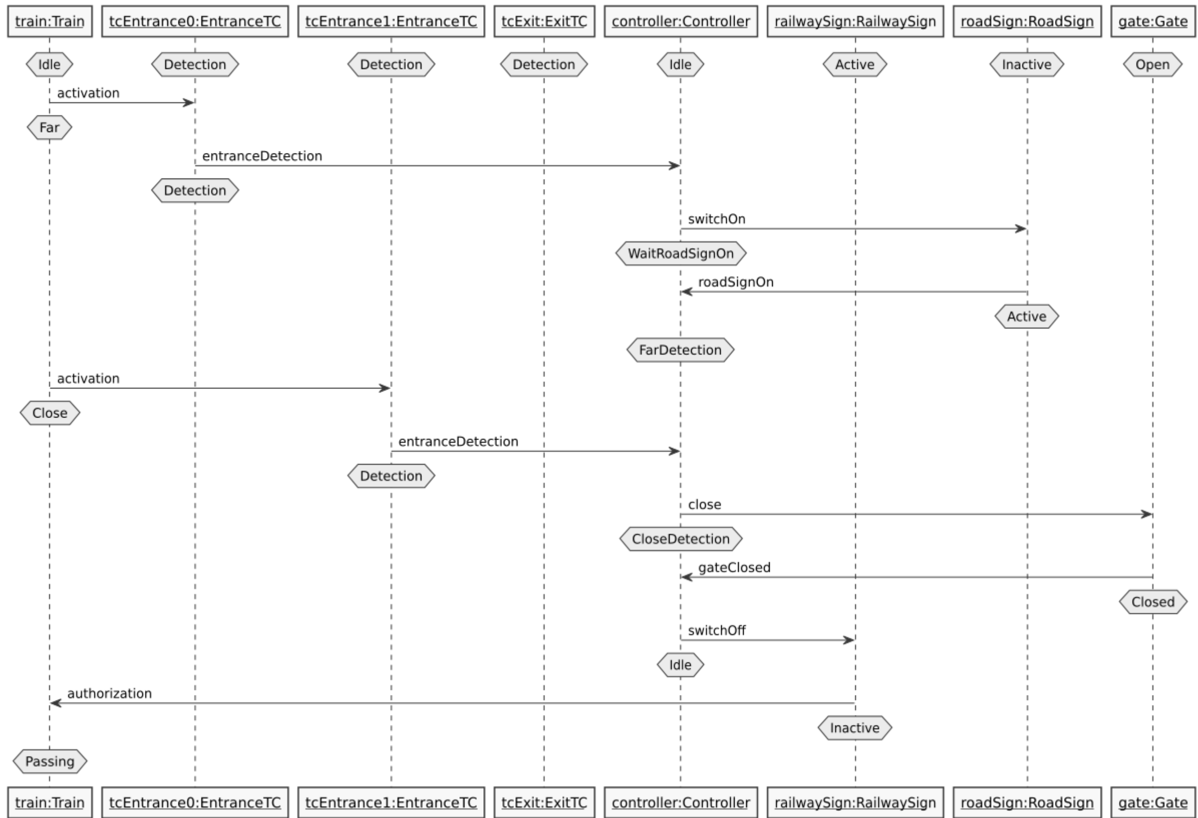



FIGURE 10.3 – Diagramme MSC montrant la fermeture du passage à niveau.

de la barrière *gate* lors de l'arrivée d'un train sur le passage à niveau. Ce mécanisme permet de visualiser des traces d'exécution de différentes façons. Les MSCs permettent par exemple de visualiser graphiquement les évènements échangés entre les différents objets actifs du modèle UML.

### 10.4.2 Débogage multivers

Pour analyser le comportement du système exécuté sur l'interpréteur EMI-UML, OBP2 permet de réaliser du débogage multivers. La Figure 10.4 montre qu'il est possible de définir des points d'arrêt conditionnels dont les expressions sont spécifiées à l'aide du langage d'observation d'EMI-UML. Dans cet exemple, deux points d'arrêt ont été définis sur l'exécution du modèle d'IRV avec pour conditions les propositions atomiques correspondant à *ccsEngaged* et *evStop*. Dans la configuration initiale, ces deux conditions étaient fausses. Nous avons demandé à OBP2 d'atteindre une configuration où la condition du premier point d'arrêt (*ccsEngaged*) soit vraie. OBP2 a alors lancé un algorithme d'atteignabilité et a retourné une trace d'exécution où l'atome *ccsEngaged* devient vrai lors du passage de l'avant-dernière à la dernière configura-

tion. La Figure 10.4 montre l'interface graphique d'OBP2 à cet instant avec la trace d'exécution retournée. Des indicateurs colorés indiquent si les propositions atomiques sont vraies (couleur verte) ou fausses (couleur rouge) dans la configuration courante. Ces indicateurs montrent ici que *ccsEngaged* est vraie et que *evStop* est fausse. Le débogage multivers permet ainsi d'atteindre plus facilement les configurations qui ont de l'intérêt pour l'utilisateur.

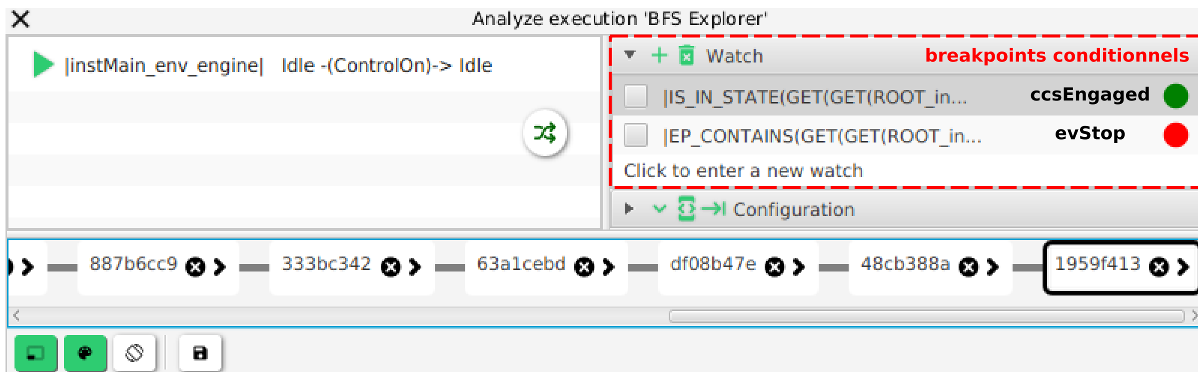



FIGURE 10.4 – Interface graphique de débogage de OBP2.

### 10.4.3 *Model-checking* LTL et détection de *deadlocks*

Pour détecter davantage de bogues logiciels et d'erreurs de conception, OBP2 peut contrôler l'exécution de l'interpréteur EMI-UML afin de détecter des *deadlocks* et vérifier des propriétés LTL. Pour cette expérimentation, nous considérons le modèle du contrôleur du passage à niveau avec les deux variantes de l'implémentation de l'*event pool* : *FifoEventPool* et *OrderedListDeferredEventPool*. Pour cette expérimentation, deux questions de recherche sont prises en compte :

**RQ#1** Les activités d'analyses permettent-elles d'étudier l'impact des points de variation sémantique sur le comportement d'un système ?

**RQ#2** L'approche EMI facilite-t-elle la compréhension des résultats d'analyse ?

Pour répondre à ces questions, les quatre exigences logicielles du passage à niveau ($E1_{CPN}$ à $E4_{CPN}$) ont été exprimées dans le formalisme LTL afin d'effectuer du *model-checking*. Les propositions atomiques sont exprimées entre "|" dans le langage d'observation d'EMI-UML et identifiées par un nom. Les expressions de ces propositions atomiques sont données dans la section E.1. Les propriétés LTL à vérifier sur le système CPN sont les suivantes :

**$P1_{CPN}$** `"[] !(|trainIsPassing| && |gateIsOpen|)"`

**$P2_{CPN}$** `"[] !(|trainIsPassing| && |roadSignIsOff|)"`

**$P3_{CPN}$** `"[] (|gateIsClosed| -> <> |gateIsOpen|)"`

**$P4_{CPN}$** `"[] (|roadSignIsOn| -> <> |roadSignIsOff|)"`

Avec l'implémentation *FifoEventPool*, l'espace d'état du modèle contient 173 configurations et 276 transitions et le *model-checker* a détecté deux *deadlocks*. La Figure 10.5 donne une représentation de cet espace d'état avec l'état initial en noir et les deux *deadlocks* détectés en rouge.

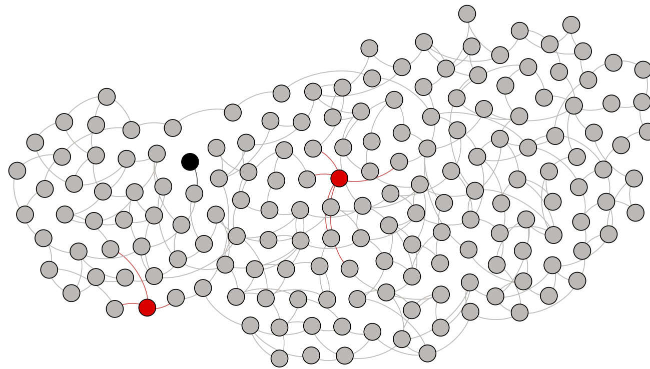


FIGURE 10.5 – Espace d'état du contrôleur de passage à niveau avec *FifoEventPool*.

Le *model-checking* avec OBP2 indique que les trois premières propriétés LTL sont vérifiées mais que la quatrième est violée. La Figure 10.6 montre le contre-exemple retourné par OBP2 pour cette dernière propriété. Ce contre-exemple indique que le feu de signalisation *roadSign* reste allumé indéfiniment sans jamais être éteint. Afin d'identifier l'erreur à l'origine de cette défaillance, le contre-exemple doit être analysé. La première chose que l'on peut remarquer est que le seul pas d'exécution disponible est “Büchi stuttering” autrement dit un pas de bégaiement. À l'instar de l'opérateur `add_implicit_transitions` pour les PUSMs, des pas de bégaiement sont également utilisés en *model-checking* LTL pour ne pas bloquer l'exécution de l'automate de Büchi en cas de *deadlock* sur le système. Le contre-exemple met donc en évidence que la propriété est violée à cause d'un *deadlock*. Un pas de bégaiement est ajouté créant ainsi une boucle d'acceptation. Pour aller plus loin dans la compréhension du problème, il faut maintenant comprendre pourquoi ce *deadlock* se produit. En analysant la configuration courante de l'interpréteur lors du *deadlock*, on s'aperçoit que l'objet *controller* est dans l'état *FarDetection* et que son *event pool* est vide. D'après le modèle UML du système, dans l'état *FarDetection*, le *controller* attend l'évènement *entranceDetection* pour pouvoir exécuter un nouveau pas. L'analyse du début de la trace du contre-exemple montre que cet évènement est bien reçu par le contrôleur mais qu'il est ensuite supprimé de l'*event pool* afin de traiter des évènements plus prioritaires mais arrivés après *entranceDetection*. L'implémentation *FifoEventPool* est donc responsable de ce *deadlock*.

L'utilisation de l'implémentation *OrderedListDeferredEventPool* résout ce problème puisque

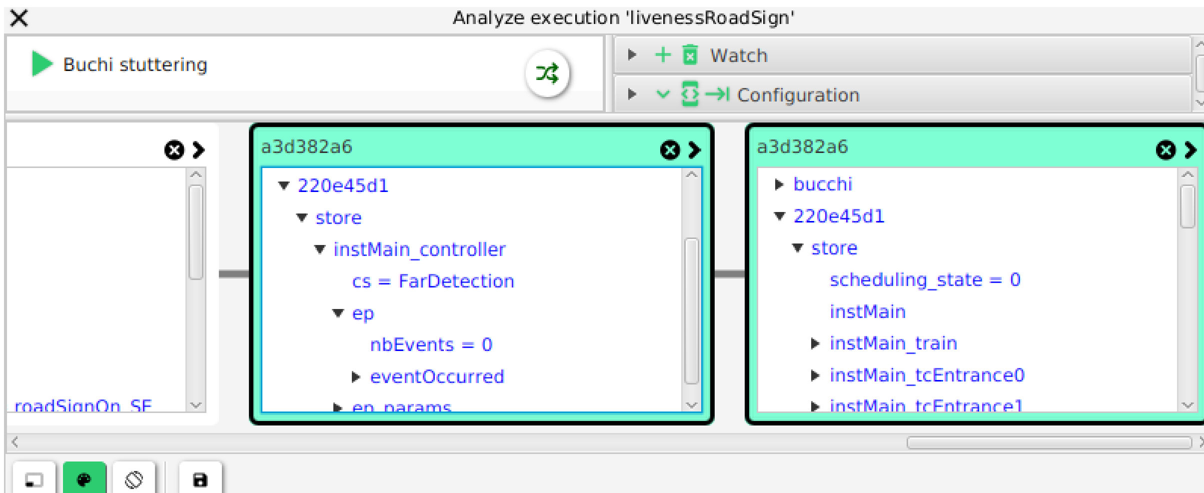



FIGURE 10.6 – Contre-exemple obtenu pour la propriété $E4_{CPN}$ avec *FifoEventPool*.

les évènements ignorés ne sont pas supprimés de l'*event pool* afin de reporter à plus tard leur traitement. Les quatre propriétés LTL sont maintenant vérifiées et l'espace d'état du modèle ne contient plus de *deadlocks* avec l'abstraction considérée pour l'environnement. L'espace d'état exploré par OBP2 contient 122 configurations et 209 transitions.

En termes de performance de *model-checking*, le couple (EMI-UML + OBP2) ne permet pas de concurrencer les outils de l'état de l'art. En dépit de la modularité apportée, la connexion TCP ralentit fortement les échanges de données entre EMI-UML et OBP2. Pour améliorer les performances d'exécution, il est possible d'utiliser une interface programmatique (plutôt que TCP) ou d'utiliser une communication par mémoire partagée. À titre indicatif, le branchement d'un algorithme BFS écrit en langage C sur l'interface STR de EMI-UML permet de réduire le temps d'exploration du modèle par un facteur 10. Pour être encore plus performant, il est possible de déployer le *model-checker* sur un FPGA et de le connecter via un bus de données à EMI-UML déployé sur un *System on Chip* (SoC). Cette expérimentation a été réalisée avec Menhir [FTL20], une version hardware du *model-checker* OBP2. Des résultats préliminaires montrent que cette architecture permet d'explorer l'espace d'état d'un modèle environ 30 fois plus rapidement qu'OBP2 connecté en TCP à EMI-UML.

En somme, cette expérimentation permet de répondre aux deux questions de recherche posées.

**Réponse à RQ#1 (Étude des points de variation sémantique)** Le choix d'une implémentation ou d'une autre pour un point de variation sémantique peut directement impacter le comportement du modèle et sa fiabilité. L'approche EMI permet d'étudier ces impacts via différentes activités d'analyse. Utiliser la même implémentation de la sémantique du langage pour toutes les activités de développement logiciel permet d'éliminer les différences sémantiques et

ainsi d'améliorer la sûreté de fonctionnement de ces systèmes.

**Réponse à RQ#2 (Compréhension des résultats d'analyse)** Les résultats de vérification sont directement exprimés avec les concepts du langage de modélisation ce qui facilite la compréhension des traces d'exécution (p. ex. des contre-exemples) et simplifie l'établissement de liens entre les données d'exécution et le modèle de conception.

### 10.4.4 *Model-checking* avec l'ordonnanceur

Les opérateurs du chapitre 6 peuvent également être utilisés dans la boucle de vérification formelle. Cette expérimentation vise à étudier l'impact de la prise en compte de l'ordonnanceur et de l'hypothèse de réactivité lors du *model-checking*. Pour réaliser cette évaluation, trois questions de recherche ont été considérées :

**RQ#3** La prise en compte de l'ordonnanceur dans la boucle de vérification aide-t-elle les ingénieurs à choisir une politique d'ordonnancement en accord avec les exigences du système considéré ?

**RQ#4** La prise en compte de l'ordonnanceur dans la boucle de vérification permet-elle de focaliser l'analyse sur les erreurs de conception essentielles ?

**RQ#5** La prise en compte de l'ordonnanceur dans la boucle de vérification permet-elle d'améliorer les performances de *model-checking* ?

Pour y répondre, l'outil OBP2 a été utilisé pour explorer l'espace d'état du modèle de contrôleur de passage à niveau, détecter des *deadlocks* et vérifier des propriétés LTL. Les deux variantes de l'implémentation de l'*event pool* (*FifoEventPool* et *OrderedListDeferredEventPool*) ont été utilisés pour étudier l'impact de l'ordonnancement sur le comportement du système.

Pour cette expérimentation, trois types d'ordonnancement ont été considérés : "sans ordonnancement" lorsque l'ordonnanceur n'est pas utilisé, "ordonnancement round-robin" et "ordonnancement à priorité fixe" lorsque l'ordonnanceur est utilisé avec les politiques d'ordonnancement du même nom. Cette expérimentation prend également en compte l'Hypothèse de Réactivité (HR) pour étudier son influence sur les résultats de vérification formelle.

Le Tableau 10.1a montre les résultats obtenus via *model-checking* avec l'implémentation *FifoEventPool* et le Tableau 10.1b ceux obtenus avec l'implémentation *OrderedListDeferred-EventPool*. Pour chaque expérimentation, ces tableaux donnent le nombre de configurations (C) et le nombre de transitions (A) de l'espace d'état du modèle, le nombre de *deadlocks* (D), et le résultat de vérification de chaque propriété. Parmi ces résultats, la première ligne de chaque tableau correspond aux résultats de *model-checking* sans ordonnancement effectué dans la section précédente. Cette première ligne sert donc de référence. Les résultats avec ordonnancement montrent qu'inclure l'ordonnanceur dans la boucle de vérification aide à réduire l'espace d'état, tend à diminuer le nombre de *deadlocks* et tend à supprimer les chemins d'exécution qui entraînent la violation de la propriété $P4_{CPN}$. Comme attendu, l'utilisation de l'hy-

| Expérimentation | C | A | D | $P1_{CPN}$ | $P2_{CPN}$ | $P3_{CPN}$ | $P4_{CPN}$ |
|---|---|---|---|---|---|---|---|
| Sans ordonnancement | 173 | 276 | 2 | ✓ | ✓ | ✓ | ✗ |
| Ordonnancement round-robin | 26 | 25 | 1 | ✓ | ✓ | ✓ | ✓ |
| Ordonnancement à priorité fixe | 21 | 21 | 0 | ✓ | ✓ | ✓ | ✓ |
| Sans ordonnancement avec HR | 50 | 54 | 2 | ✓ | ✓ | ✓ | ✗ |
| Ordonnancement round-robin avec HR | 8 | 7 | 1 | ✓ | ✓ | ✓ | ✗ |
| Ordonnancement à priorité fixe avec HR | 21 | 21 | 0 | ✓ | ✓ | ✓ | ✓ |

(a) Avec l'implémentation *FifoEventPool*.

| Expérimentation | C | A | D | $P1_{CPN}$ | $P2_{CPN}$ | $P3_{CPN}$ | $P4_{CPN}$ |
|---|---|---|---|---|---|---|---|
| Sans ordonnancement | 122 | 209 | 0 | ✓ | ✓ | ✓ | ✓ |
| Ordonnancement round-robin | 28 | 28 | 0 | ✓ | ✓ | ✓ | ✓ |
| Ordonnancement à priorité fixe | 21 | 21 | 0 | ✓ | ✓ | ✓ | ✓ |
| Sans ordonnancement avec HR | 32 | 48 | 0 | ✓ | ✓ | ✓ | ✓ |
| Ordonnancement round-robin avec HR | 27 | 27 | 0 | ✓ | ✓ | ✓ | ✓ |
| Ordonnancement à priorité fixe avec HR | 21 | 21 | 0 | ✓ | ✓ | ✓ | ✓ |

(b) Avec l'implémentation *OrderedListDeferredEventPool*.

TABLE 10.1 – Résultats du *model-checking* avec différentes implémentations de l'event pool (avec C : le nombre de configurations, A : le nombre de transitions, D : le nombre de *deadlocks*, et les résultats de vérification des propriétés).

pothèse de réactivité réduit l'espace d'état car elle diminue le nombre d'entrelacements avec les évènements venant de l'environnement.

Dans le cas spécifique de ce modèle de contrôleur de passage à niveau, l'ordonnancement à priorité fixe donne de meilleurs résultats que l'ordonnancement round-robin. Ceci peut être expliqué par le fait que l'ordonnancement à priorité fixe permet de mieux configurer les priorités des objets actifs du modèle. Il est aussi possible de remarquer que le cas "Ordonnancement round-robin avec HR" a très peu d'états dans son espace d'état. Après analyse, il s'avère que dans ce cas le scénario nominal aboutit à un *deadlock* qui bloque l'exécution du modèle et entraîne la violation de la propriété $P4_{CPN}$. Sans l'hypothèse de réactivité ("Ordonnancement round-robin"), un problème similaire est observé sauf que le *deadlock* qui entraîne la violation de $P4_{CPN}$ a été évité mais un autre *deadlock* est toujours présent. De notre perspective, il semble très intéressant pour les ingénieurs de pouvoir détecter ces problèmes dès la phase de vérification formelle. Pour les résoudre, les ingénieurs peuvent soit changer l'implémentation de l'*event pool*, la politique d'ordonnancement ou les deux.

Pour visualiser le comportement du système ordonnancé, le mécanisme de traces de EMI-UML permet également de produire des *timing diagrams* en utilisant le formalisme PlantUML.

La Figure 10.7 montre un *timing diagram* illustrant une trace d'exécution du modèle de contrôleur de passage à niveau avec un ordonnancement à priorité fixe et l'hypothèse de réactivité pour l'implémentation *FifoEventPool*.

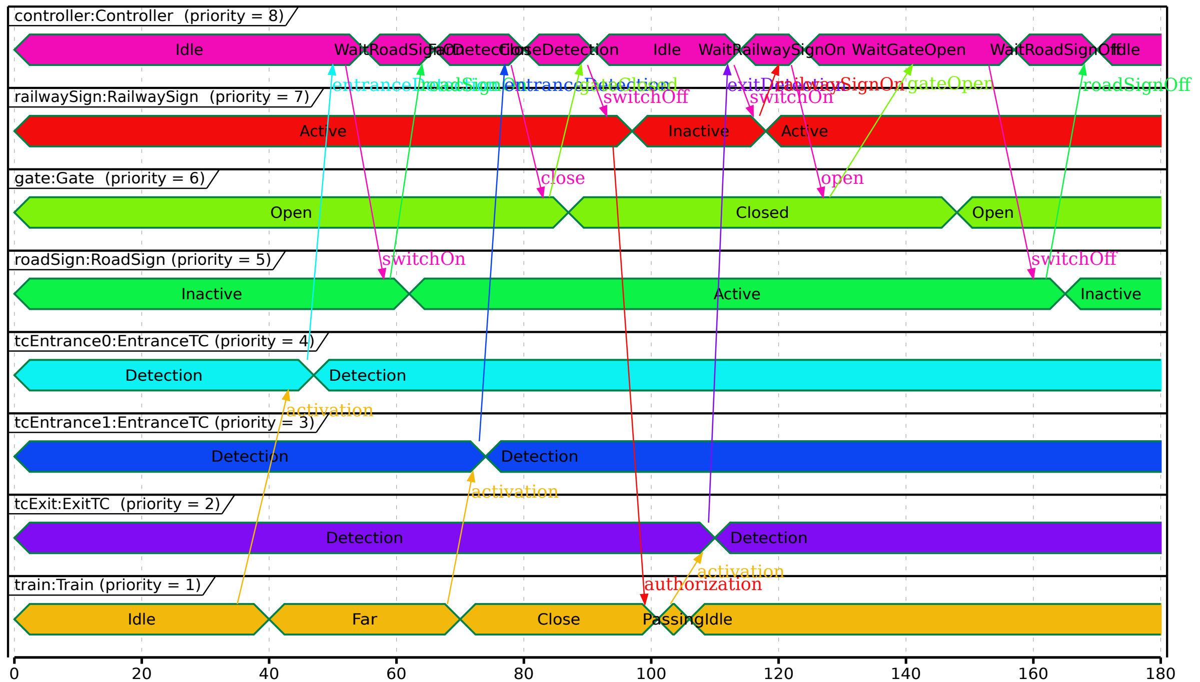



FIGURE 10.7 – *Timing diagram* d'une exécution du contrôleur de passage à niveau avec un ordonnancement à priorité fixe et l'hypothèse de réactivité pour l'implémentation *FifoEventPool*.

Avec ces résultats, on remarque également que toutes les propriétés qui sont vérifiées sans ordonnancement sont aussi vérifiées avec ordonnancement (le contraire n'étant pas vrai). C'est la raison pour laquelle les techniques de *model-checking* se sont affranchies des dépendances liées aux politiques d'ordonnancement réelles pour en considérer une abstraction. Cependant, les expérimentations réalisées montrent que l'approche proposée est intéressante à plusieurs niveaux.

**Réponse à RQ#3 (Choix d'une politique d'ordonnancement)** Inclure l'ordonnanceur dans la boucle de vérification permet de s'assurer que la politique d'ordonnancement choisie est appropriée pour le système considéré et qu'elle ne nuit pas aux exigences fonctionnelles auxquelles il doit satisfaire.

**Réponse à RQ#4 (Analyse ciblée)** La prise en compte de l'ordonnanceur dans la boucle de vérification permet de ne vérifier formellement que le comportement du système ordonnancé et donc de ne se focaliser que sur les erreurs de conception qui peuvent se produire lors de l'exécution.

**Réponse à RQ#5 (Performances)** Le fait de considérer l'ordonnanceur et l'hypothèse de

réactivité en phase de *model-checking* permet généralement de rendre les techniques de *model-checking* plus efficaces en réduisant la taille de l'espace d'état à explorer. Cette tendance a également été confirmée en appliquant l'approche à d'autres modèles UML. En général, l'espace d'état du modèle est plus petit car le non-déterminisme général de l'exécution du système a été résolu. Néanmoins, l'état d'exécution de la politique d'ordonnancement peut différencier des configurations par ailleurs équivalentes.

### 10.4.5 *Model-checking* avec des PUSMs

Cette section illustre la spécification de PUSMs pour le modèle d'interface du régulateur de vitesse et leur utilisation en *model-checking* avec OBP2 et EMI-UML. Pour cette expérimentation, deux questions de recherche ont été prises en compte :

**RQ#6** Les propriétés formelles peuvent-elles facilement être spécifiées avec des PUSMs ?

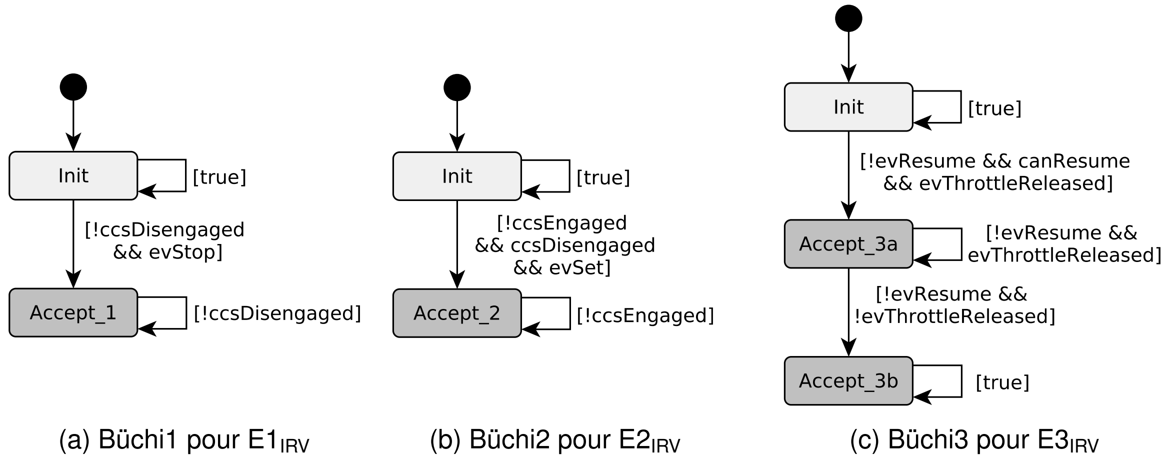
**RQ#7** Les résultats de *model-checking* obtenus avec des PUSMs sont-ils comparables à ceux obtenus par *model-checking* des propriétés LTL équivalentes ?

#### 10.4.5.1 Spécification de PUSMs

Avant de pouvoir effectuer la vérification formelle, les exigences de l'interface du régulateur de vitesse doivent être exprimées sous forme de PUSMs. Les propriétés de vivacité (correspondant aux exigences $E1_{IRV}$, $E2_{IRV}$, $E3_{IRV}$) ont été encodées sous forme d'automates de Büchi et les propriétés de sûreté (correspondant aux exigences $E4_{IRV}$, $E5_{IRV}$, $E6_{IRV}$) ont été exprimées sous forme d'automates observateurs. Les gardes de ces automates sont exprimées dans le langage d'observation d'EMI-UML. Les expressions de ces propositions atomiques sont données dans la section E.2.

La Figure 10.8 présente les machines à états des PUSMs représentant des automates de Büchi pour $E1_{IRV}$, $E2_{IRV}$ et $E3_{IRV}$. Le *model-checker* ne pouvant utiliser qu'un seul automate de Büchi à la fois, les trois PUSMs de la Figure 10.8 peuvent être combinés en un seul afin de vérifier les trois propriétés simultanément (cf. section 9.2.2). L'automate résultant est illustré sur la Figure 10.9.

Pour les automates observateurs, les machines à états des PUSMs correspondant aux exigences $E4_{IRV}$, $E5_{IRV}$ et $E6_{IRV}$ sont illustrées sur la Figure 10.10. Les automates *Observateur4*, *Observateur5* et *Observateur6* sont des automates observateurs déterministes qui peuvent être utilisés en *monitoring*. L'automate *ObservateurND* est un exemple d'automate observateur non-déterministe combinant la vérification des exigences $E5_{IRV}$ et $E6_{IRV}$. Cet automate est non-déterministe car les états *Engaged* et *Disengaged* ont deux transitions sortantes qui peuvent être tirables au même instant puisque leurs gardes peuvent être vraies en même temps.




(a) Büchi1 pour $E1_{IRV}$ (b) Büchi2 pour $E2_{IRV}$ (c) Büchi3 pour $E3_{IRV}$

FIGURE 10.8 – Machines à états des PUSMs (représentant des automates de Büchi) pour le modèle d'IRV.

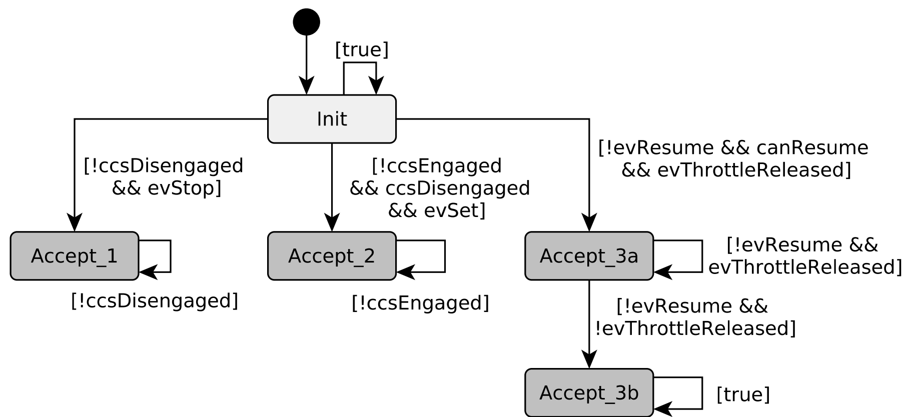
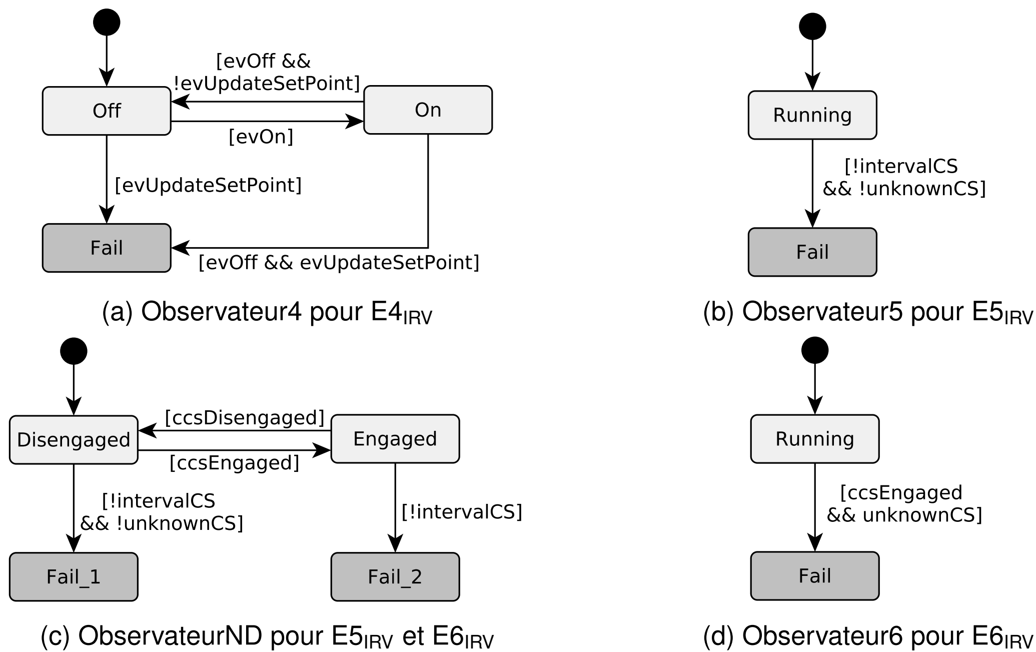



FIGURE 10.9 – Combinaison de plusieurs PUSMs en un pour vérifier $E1_{IRV}$, $E2_{IRV}$ et $E3_{IRV}$ simultanément.




(a) Observateur4 pour $E4_{IRV}$
(b) Observateur5 pour $E5_{IRV}$
(c) ObservateurND pour $E5_{IRV}$ et $E6_{IRV}$
(d) Observateur6 pour $E6_{IRV}$

FIGURE 10.10 – Machines à états des PUSMs (représentant des automates observateurs) pour le modèle d'IRV.

Tous ces PUSMs peuvent maintenant être utilisés en vérification formelle pour analyser le comportement de l'interface du régulateur de vitesse.

#### 10.4.5.2 ***Model-checking* du comportement du système**

Le *model-checking* à l'aide des PUSMs a été mis en œuvre avec l'architecture de la Figure 9.4.

Pour les propriétés de vivacité, la vérification a été réalisée en utilisant l'algorithme “nested DFS” [GS09] pour la détection de boucles d'acceptation et l'interpréteur EMI-UML pour exécuter le modèle du système et les PUSMs. Chaque automate de Büchi a été utilisé séparément pour vérifier le comportement du système et aucune défaillance n'a été détectée par OBP2. Le PUSM de la Figure 10.9 résultant de la combinaison des trois PUSMs pour $E1_{IRV}$, $E2_{IRV}$ et $E3_{IRV}$ confirme également ce résultat en *model-checking*.

Pour les propriétés de sûreté, la vérification réalisée avec un algorithme d'atteignabilité montre que les exigences $E4_{IRV}$ et $E5_{IRV}$ sont satisfaites et que l'exigence $E6_{IRV}$ est violée. En conséquence, la propriété encodée par l'*ObservateurND* qui est une combinaison de $E5_{IRV}$ et $E6_{IRV}$ est aussi violée.

Pour vérifier la validité de notre approche, ces résultats ont été comparés aux résultats obtenus par *model-checking* des propriétés LTL équivalentes. Ainsi, toutes les exigences de

l'interface du régulateur de vitesse ont été spécifiées en LTL avec des propositions atomiques exprimées à l'aide du langage d'observation d'EMI-UML.

**P1**$_{\mathbf{IRV}}$ "[] (|evStop| -> (<> |ccsDisengaged|))"

**P2**$_{\mathbf{IRV}}$ "[] ((|ccDisengaged| && |evSet|) -> (<> |ccEngaged|))"

**P3**$_{\mathbf{IRV}}$ "[] ((|canResume| && |evThrottleReleased|) ->
(|evThrottleReleased| U |evResume|))"

**P4**$_{\mathbf{IRV}}$ "(!|evUpdateSetPoint| W |evOn|) &&
([] (|evOff| -> (!|evUpdateSetPoint| W |evOn|)))"

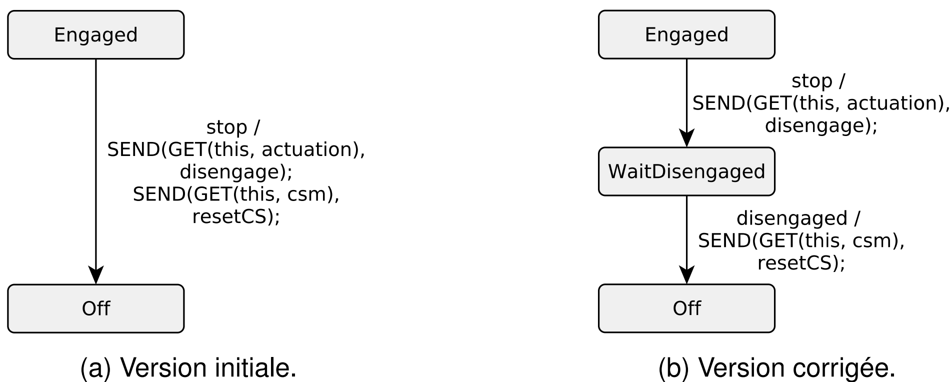
**P5**$_{\mathbf{IRV}}$ "[] (|intervalCS| or |unknownCS|)"

**P6**$_{\mathbf{IRV}}$ "[] (|ccsEngaged| -> !|unknownCS|)"

En comparaison avec les PUSMs, certaines de ces propriétés LTL ont des expressions plutôt complexes car elles requièrent la connaissance des opérateurs LTL et notamment des opérateurs temporels (p. ex. *globally*, *weak until*) (cf. RQ#6). En particulier, la propriété $P4_{IRV}$ a demandé un travail de réflexion important pour pouvoir être exprimée en LTL alors que l'automate observateur correspondant (Figure 10.10a) a été trivial à concevoir en UML.

**Réponse à RQ#6 (Expression des propriétés)** Les ingénieurs peuvent facilement spécifiées des propriétés formelles sous forme de PUSMs s'ils maîtrisent les concepts du langage UML.

Le *model-checking* de ces propriétés LTL montre que toutes les propriétés sont vérifiées excepté la propriété $P6_{IRV}$. Ces résultats basés sur les propriétés LTL sont donc identiques à ceux obtenus avec les PUSMs (cf. RQ#7). Pour comprendre pourquoi l'exigence $E6_{IRV}$ n'est pas satisfaite, il faut analyser le contre-exemple retourné par OBP2. Ce contre-exemple montre que lorsque l'objet *controller* de l'IRV va de l'état *Engaged* à l'état *Off*, les évènements *disengage* et *resetCS* sont tous les deux envoyés en même temps (cf. Figure 10.11a). Cependant, l'ordonnanceur du système peut choisir de traiter l'évènement *resetCS* en premier ce qui met




(a) Version initiale.

(b) Version corrigée.

FIGURE 10.11 – Extrait de la machine à états de l'objet *controller* de l'IRV.

la vitesse de croisière à une valeur indéfinie (c.-à-d. -1) alors que le système est toujours engagé. La Figure 10.11b montre comment corriger le problème. Un état intermédiaire, appelé ici *WaitDisengaged*, doit être ajouté pour envoyer l'évènement *disengage* en premier, attendre son acquittement et enfin envoyer l'évènement *resetCS*. Le problème était donc dû à une erreur subtile d'entrelacement d'évènements. Avec les PUSMs, la trace de l'automate de Büchi est directement exprimée avec des concepts UML ce qui simplifie encore un peu plus l'analyse du contre-exemple en comparaison au *model-checking* LTL (cf. RQ#2). Le fait d'utiliser l'ordonnanceur dans la boucle de vérification, avec une politique d'ordonnancement qui rend le traitement de l'évènement *disengage* plus prioritaire que *resetCS*, aurait également pu permettre d'éviter ce problème.

Une nouvelle itération du processus de vérification montre que toutes les propriétés sont maintenant vérifiées avec les PUSMs et avec le *model-checking* LTL. La version corrigée de ce modèle possède un espace d'état contenant 17 134 122 configurations et 29 088 210 transitions. Une comparaison entre les performances du *model-checking* basé sur les PUSMs et le *model-checking* basé sur LTL a également été menée à la fois sur le temps et la mémoire prise par le *model-checker* pour effectuer la vérification. En moyenne pour ce modèle de l'IRV, la vérification avec les PUSMs est plus efficace que la vérification LTL de 6,7% en mémoire et de 7,4% en temps. Ces résultats s'expliquent notamment par le fait que la composition synchrone est implémentée en langage C plutôt qu'en Java.

**Réponse à RQ#7 (Comparaison des résultats LTL vs PUSMs)** Sur le modèle d'IRV considéré, les résultats de *model-checking* obtenus avec des PUSMs sont identiques à ceux obtenus par *model-checking* des propriétés LTL équivalentes.

### 10.4.6 *Model-checking* des mécanismes de sûreté de fonctionnement

Au cours de nos expérimentations avec les PUSMs, nous avons également exploré la possibilité d'utiliser des mécanismes de sûreté de fonctionnement. Pour l'IRV, la disponibilité du système n'est pas critique pour la sûreté du véhicule. Ainsi, si une défaillance est détectée, une action appropriée est d'éteindre le régulateur de sorte que le conducteur reprenne le contrôle du véhicule.

Pour implémenter ce comportement, les automates observateurs doivent être couplés avec un mécanisme de sûreté de fonctionnement permettant la désactivation du système en cas de défaillance. Idéalement, ce dispositif permettant d'agir sur le système doit être séparé des automates observateurs dont le rôle est uniquement de suivre l'exécution du système et d'émettre une notification lorsqu'un comportement incorrect a été détecté. Néanmoins, dans le cadre de cette thèse, ce mécanisme a été directement combiné au sein des machines à états des automates observateurs. Pour cela, nous avons ajouté l'envoi d'un évènement “stop” à destination du contrôleur du système sur l'action *entry* de tous les états d'acceptation des automates ob-

servateurs utilisés en *monitoring*. Les automates résultants n'observent donc pas seulement l'exécution du système mais ils sont aussi capables de réagir lorsqu'une défaillance se produit. Ils ne peuvent cependant agir sur le système qu'en cas de défaillance afin de ne pas impacter le comportement nominal du système. Pour vérifier le comportement de ce mécanisme de sûreté de fonctionnement, la propriété LTL suivante peut être utilisée :

```
"[] (|failureObserved| -> [] (|evStopToController| -> (<> |ccsDisengaged|)))"
```

Cette propriété a été vérifiée avec le *model-checker* OBP2 pour l'*Observateur6* qui atteignait un état d'acceptation sur la version initiale du modèle. En conséquence, cela signifie que notre langage d'observation est suffisamment expressif pour spécifier des propriétés à propos du système monitoré et que les PUSMs peuvent être couplés à des mécanismes de sûreté de fonctionnement.

### 10.4.7 Exécution embarquée et *monitoring*

Une fois vérifiés, les modèles UML peuvent être déployés avec EMI-UML sur une cible embarquée STM32 discovery. Il suffit pour cela de remplacer le modèle abstrait de l'environnement par la connexion aux périphériques réels de la plateforme d'exécution embarquée. Grâce au continuum, les activités d'analyse (p. ex. la simulation interactive, le débogage multivers) peuvent toujours être mises en œuvre lors de l'exécution sur une cible embarquée.

Pour l'exécution réelle, il est également possible d'effectuer du *monitoring* en déployant des PUSMs sur la plateforme d'exécution. Pour évaluer la solution de *monitoring* proposée, deux questions de recherche ont été considérées.

**RQ#8** Les résultats obtenus par *monitoring* sur la trace d'exécution courante sont-ils cohérents avec ceux obtenus par *model-checking* ?

**RQ#9** La solution proposée pour le *monitoring* peut-elle passer à l'échelle ?

Pour illustrer la mise en œuvre du *monitoring*, le modèle d'interface du régulateur de vitesse est utilisé. Pour les besoins de l'expérience, ce modèle a été déployé avec un environnement abstrait et non avec les capteurs et actionneurs d'un véhicule réel. Les objets actifs de l'environnement sont ici exécutés avec la même instance de l'interpréteur EMI-UML que les objets du système. Les PUSMs représentant des automates observateurs déterministes (cf. *Observer4*, *Observer5* et *Observer6* de la Figure 10.10) ont également été déployés sur la plateforme d'exécution afin de monitorer l'exécution du système en accord avec l'architecture de la Figure 9.5. Aucune défaillance n'a été détectée sur la version corrigée du modèle UML de l'interface du régulateur de vitesse. En revanche, le *monitoring* de la version initiale du modèle montre que l'*Observateur6* aurait réussi à détecter la défaillance si elle était survenue.

**Réponse à RQ#8 (Comparaison des résultats *monitoring* vs *model-checking*)** Sur le modèle d'IRV considéré, les résultats obtenus par *monitoring* sont cohérents avec ceux obte-

nus par *model-checking*. Contrairement au *model-checking* et comme toute solution de *monitoring*, la solution proposée ne permet de détecter que les défaillances qui se produisent sur la trace d'exécution courante.

En termes de performance, l'exécution des automates observateurs induit un surcoût en comparaison à l'exécution du même modèle UML sans *monitoring*. En plus du coût des moniteurs, le *monitoring* accroît le temps d'exécution de 6,5% à cause de l'utilisation de l'opérateur de composition synchrone. Le coût des moniteurs dépend de la taille du modèle et du nombre de gardes devant être évaluées par les automates observateurs à chaque pas d'exécution. Une estimation du surcoût (en %) induit par $N$ moniteurs sur le temps d'exécution est donnée par l'équation suivante :

$$\texttt{surcoût} \approx 6.5 + \frac{100}{\texttt{nb_unitées_exécution}} \sum_{i=1}^{N} \frac{\texttt{nb_transitions}_i}{\texttt{nb_états}_i}$$

où `nb_unitées_exécution` est le nombre d'objets actifs du modèle UML, $\texttt{nb_états}_i$ est le nombre d'états (en excluant les pseudo-états et les états d'acceptation) du moniteur $i$, et $\texttt{nb_transitions}_i$ est le nombre de transitions sortantes des états considérés. Par exemple, l'utilisation d'un automate observateur avec un modèle contenant 10 objets actifs va ajouter un surcoût de 10% alors que ce surcoût serait seulement de 1% si 100 objets actifs étaient utilisés. Pour chaque moniteur, ce coût est ensuite pondéré par $\frac{\texttt{nb_transitions}}{\texttt{nb_états}}$ c.-à-d. le nombre moyen de gardes devant être évaluées à chaque pas par cet observateur. Pour le modèle d'IRV, cette équation donne un surcoût de 50,2% alors qu'en pratique nous obtenons 50.8%. En termes d'empreinte mémoire, le surcoût mesuré est de 8.2% dont approximativement 1.2% est dû à la composition synchrone et 7% aux trois moniteurs. Ces mesures ont été réalisées en comparant le temps pris pour exécuter un million de transitions et la taille de l'exécutable binaire avec et sans moniteurs. De notre perspective, ces surcoûts sont acceptables pour l'exécution de systèmes embarqués. Ces métriques doivent néanmoins être adaptées avec les contraintes spécifiques et le niveau de criticité de chaque système.

**Réponse à RQ#9 (Passage à l'échelle du *monitoring*)** À partir de l'équation de surcoût, il est possible de remarquer que cette approche de *monitoring* passe à l'échelle car le coût relatif d'un automate observateur décroît à mesure que la taille du système augmente.

### 10.4.8 Évaluation des performances d'exécution

Lors de la conception d'EMI-UML, nous avons tenté de faire des choix techniques visant à obtenir de bonnes performances d'exécution. Pour évaluer ces performances, la question de recherche suivante a été considérée.

**RQ#10** Les performances d'exécution de l'interpréteur EMI-UML sont-elles comparables à

celles des outils de l'état de l'art ?

Pour y répondre, cette section présente une expérimentation visant à comparer les performances d'exécution d'EMI-UML avec deux autres outils de la littérature : Papyrus Software Designer [Pha+17] (Papyrus SD) et Rhapsody [HK04]. Pour cette expérience, ces trois outils ont exécuté un modèle UML très simple composé de deux objets s'envoyant des évènements à tour de rôle. Le Tableau 10.2 donne pour chaque outil le temps moyen pris pour exécuter un pas d'exécution sur plus de 200 milles pas exécutés, la taille de l'exécutable binaire après *stripping* (i.e., sans les symboles de débogage) et la taille de ses dépendances, les pics d'occupation de la pile et du tas, ainsi que la taille totale des allocations réalisées.

| Outils | Papyrus SD | Rhapsody | EMI-UML |
|---|---|---|---|
| Temps moyen par pas d'exécution (en us) | 80.75 | 1.36 | 0.47 |
| Taille de l'exécutable (en ko) | 22.2 | 54.6 | 67.0 |
| Taille des dépendances (en ko) | 1681.3 | 1180.5 | 0 |
| Pic d'occupation de la pile (en ko) | 5.2 | 2.7 | 2.4 |
| Pic d'occupation du tas (en ko) | 72.6 | 29.1 | 0 |
| Total allocation (en ko) | 72.6 | 18724.9 | 0 |

TABLE 10.2 – Comparaison des performances d'exécution.

Contrairement à EMI-UML, Papyrus SD et Rhapsody utilisent des techniques de génération code (vers le langage C) qui sont en théorie plus efficace que l'interprétation. Néanmoins, les résultats obtenus montrent que EMI-UML est 2.9 fois plus performant que Rhapsody et 171.8 fois plus efficace que Papyrus SD. Ces résultats sont surprenants et nous avons donc réalisé du profilage de code pour chercher à les comprendre. Le profilage montre que ces deux outils utilisent des concepts POSIX (p. ex. des *threads*, des *message queues*) qui augmentent considérablement la taille des piles d'appels de fonctions (notamment pour la gestion des évènements) et ralentissent l'exécution. En comparaison, l'implémentation des *event pools* dans EMI-UML repose sur des files de messages réimplémentées par nos soins avec peu de fonctions. La différence de performance semble aussi s'expliquer par le fait que Papyrus SD et Rhapsody supporte un sous-ensemble d'UML plus important que EMI-UML rendant les optimisations plus difficiles. Contrairement à EMI-UML, ces outils utilisent également de l'allocation dynamique de mémoire ce qui permet d'être plus flexible mais moins efficace. Par exemple avec Rhapsody, plus de 9% du temps d'exécution est utilisé par la fonction *malloc* afin d'allouer les 18724.9 ko de mémoire nécessaire à l'exécution du modèle. Pour Papyrus SD, il semble que l'exécution est plus efficace lorsqu'elle est davantage parallélisée comme le montre l'étude en [Pha+17]. Sur le modèle simple de cette étude, les deux objets s'envoient des évènements et doivent donc s'attendre mutuellement. Avec Papyrus SD, le profilage de code montre que beaucoup de temps est consommé par la gestion des concepts POSIX et notamment des mutex. Concer-

nant les autres résultats, le pic d'occupation de la pile est le plus faible sur EMI-UML avec 2.4 ko contre 2.7 ko pour Rhapsody et 5.2 ko pour Papyrus SD. En ce qui concerne la taille de l'exécutable, EMI-UML possède l'exécutable le plus volumineux mais ce résultat est à nuancer par le fait que Papyrus SD et Rhapsody utilisent des librairies dynamiques (libpthread, librt et libm) occupant plus de 1 Mo de mémoire.

**Réponse à RQ#10 (Performances d'exécution)** Pour l'exécution concrète, les performances d'exécution de EMI-UML sont donc au-delà des attentes espérées et surpassent même les performances de certains outils de la littérature.

## 10.5 Expérimentations avec d'autres outils

Afin de généraliser l'approche et de montrer sa versatilité, des expérimentations ont été menées avec d'autres moteurs d'exécution et/ou d'autres outils d'analyse.

### 10.5.1 EMI-LOGO : un interpréteur de modèles LOGO

L'approche EMI a été mise en œuvre pour réaliser un interpréteur de modèles LOGO [AGR74] appelé EMI-LOGO. LOGO est un langage de programmation éducatif permettant de programmer le déplacement d'une tortue pour réaliser des dessins. Le langage LOGO a été créé en 1967 au laboratoire d'intelligence artificielle du MIT. Il s'agit d'une adaptation du langage LISP visant à faciliter l'apprentissage des concepts de programmation. Ce langage a été choisi pour expérimenter notre approche car il s'agit d'un des premiers langages interprétés. De plus, contrairement à UML, LOGO est un langage séquentiel (c.-à-d. avec un seul chemin d'exécution) et procédural avec des concepts algorithmiques relativement simples.

L'interpréteur EMI-LOGO a été implémenté avec le langage C comme langage hôte en utilisant la même méthodologie de conception que l'interpréteur EMI-UML. Parmi les points clés de ce prototype, l'implémentation [6] de la sémantique du langage LOGO repose sur une variante du patron de conception Interpréteur [EHV95]. Ce patron de conception est particulièrement adapté ici car le modèle forme un arbre syntaxique abstrait où chaque nœud correspond à une instruction. Pour l'implémentation d'EMI-LOGO, ce patron a été modifié pour que l'exécution soit pilotable. Plus concrètement, cela se traduit par l'ajout d'une pile d'exécution avec pour chaque instruction un compteur de programme indiquant où en est rendue l'exécution. Par exemple, pour l'instruction `FOR`, le compteur d'itération est utilisé comme compteur de programme afin que chaque itération soit vue comme un pas observable. Une autre observation intéressante est que l'état de l'interface graphique doit également être sauvegardé

6. L'implémentation de la sémantique de LOGO est inspirée de celle créée par Didier Vojtisek en `https://dvojtise.github.io/mde-crashcourse-logo/`.

dans la configuration afin de pouvoir replacer celle-ci dans l'état souhaité. Pour l'interpréteur EMI-LOGO, cela se traduit par la sauvegarde de tous les points et segments visibles ayant été parcourus par la tortue.

Pour les activités d'analyse, ce moteur d'exécution possède un langage d'observation basé sur le langage C et complété avec des opérateurs sous forme de macros C. Le *model-checker* OBP2 peut être utilisé pour piloter l'exécution de EMI-LOGO et mener diverses activités d'analyse : la simulation interactive, le débogage multivers et le *model-checking* LTL. Un mécanisme de trace implémenté dans l'interpréteur permet également d'afficher le dessin en cours de réalisation sous forme d'une image *Scalable Vector Graphics* (SVG). La Figure 10.12 présente un exemple de dessin obtenu avec EMI-LOGO à partir du programme LOGO en Listing 10.1.

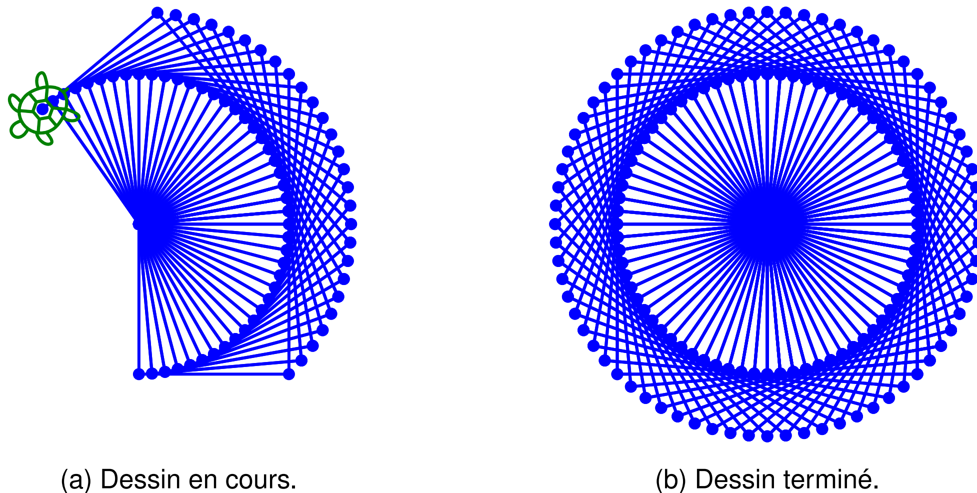
```
to drawSquare
    repeat 4 [
        forward 100
        right 90
    ]
end

clear
pendown
repeat 72 [
    drawSquare
    right 5
]
penup
```

Listing 10.1 – Exemple de programme LOGO.

### 10.5.2 AnimUML : un outil d'interprétation et d'animation de modèles UML

AnimUML [Jou+20] est un outil d'interprétation et d'animation de modèles UML permettant d'éditer, de simuler et de vérifier des modèles partiels. Cet outil, implémenté en JavaScript, à vocation à être utilisé dans les phases de conception amont pour aider les ingénieurs à concevoir leurs modèles. Des modèles même incomplets peuvent déjà être exécutés et animés avec AnimUML ce qui permet de détecter des erreurs de conception au plus tôt.

AnimUML propose à la fois une interface graphique web et un interpréteur de modèles UML implémenté en JavaScript. L'interface graphique propose diverses fonctionnalités d'édition de



(a) Dessin en cours. (b) Dessin terminé.

FIGURE 10.12 – Exemple de dessin pour le programme LOGO ci-dessus.

modèles permettant de créer ou de modifier des modèles UML. Ces modèles peuvent ensuite être exécutés sur un moteur d'exécution et pilotés par des outils d'analyse au travers de l'interface de contrôle d'exécution. La conception de l'interpréteur de modèles UML est fortement inspirée de la méthodologie de conception EMI. Il implémente la sémantique d'UML sous forme opérationnelle en se basant sensiblement sur le même sous-ensemble qu'EMI-UML. Il supporte les concepts pouvant être représentés sur les diagrammes de classes, de structures composites et de machines à états. Ce moteur d'exécution en JavaScript rajoute également des concepts annexes pour supporter l'interprétation de modèles UML partiels comme l'*Ether* pour le traitement des messages sans destinataire explicite.

Pour l'analyse de modèles, AnimUML permet également de piloter et d'observer l'exécution des modèles UML. AnimUML permet d'animer des modèles par exemple en changeant la couleur de l'état courant d'une machine à états ou en mettant en surbrillance les transitions tirables. Il offre également des mécanismes de traces interactifs permettant d'afficher des diagrammes de séquence ou des *timing diagrams* en utilisant PlantUML. La Figure 10.13 présente un diagramme de séquence réalisé avec AnimUML pour le modèle du contrôleur de passage à niveau. AnimUML utilise l'interface de contrôle d'exécution permettant à la fois de connecter des outils d'analyse externes comme le *model-checker* OBP2 ou d'autres moteurs d'exécution comme l'interpréteur EMI-UML. Ainsi, AnimUML peut servir d'interface graphique pour EMI-UML afin d'animer les modèles exécutés par cet interpréteur.

AnimUML offre enfin un outil d'exécution comparée (ou de bisimulation forte [Fer90]) permettant de comparer les espaces d'état de deux modèles s'exécutant chacun sur un moteur d'exécution. Cette fonctionnalité peut ainsi permettre de comparer les sémantiques opérationnelles de deux moteurs d'exécution ou encore d'étudier l'impact de certains changements sur

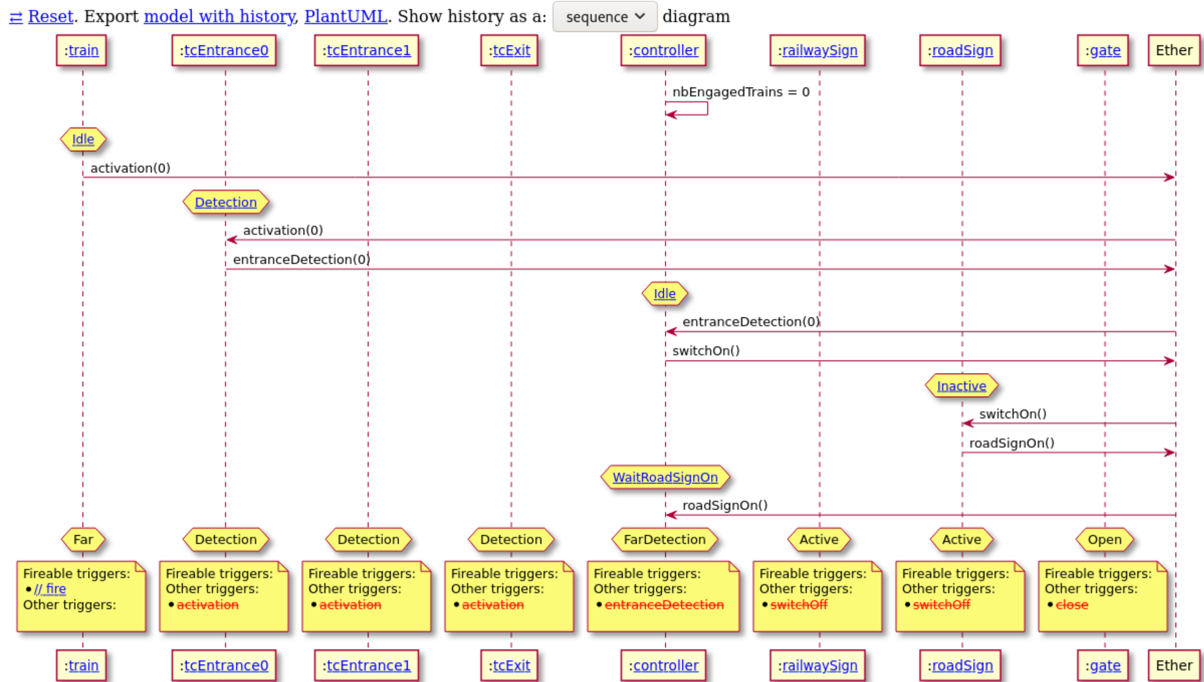



FIGURE 10.13 – Capture d’écran de l’interface AnimUML montrant un diagramme de séquence interactif pour le modèle du contrôleur de passage à niveau.

l’exécution d’un même modèle.

Pour effectuer toutes ces activités d’analyse (y compris l’exécution comparée), l’interface de contrôle d’exécution définie dans cette thèse s’est révélée suffisante pour piloter l’exécution des modèles.

### 10.5.3 Intégration d’EMI-UML dans Gemoc Studio

Pour valider l’interface de contrôle d’exécution, nous avons également intégré l’interpréteur EMI-UML comme moteur d’exécution dans Gemoc Studio. Gemoc Studio peut ainsi piloter l’exécution de modèles UML sur EMI-UML afin de mener de la simulation et du débogage omniscient.

Pour la simulation, Gemoc studio lance l’exécution du modèle comme s’il était déployé sur la plateforme d’exécution réelle. Contrairement à la simulation interactive, ce n’est pas l’utilisateur mais un ordonnanceur qui résout le non-déterminisme général de l’exécution du modèle.

Pour le débogage omniscient, la Figure 10.14 montre une partie de l’interface graphique de Gemoc Studio lors du débogage du modèle de contrôleur de passage à niveau. À gauche, l’addon *Concurrent Logical Steps Decider* permet à l’utilisateur de choisir le prochain pas d’exécution. À droite, l’addon *Multidimensional debugger* affiche un chronogramme permettant de

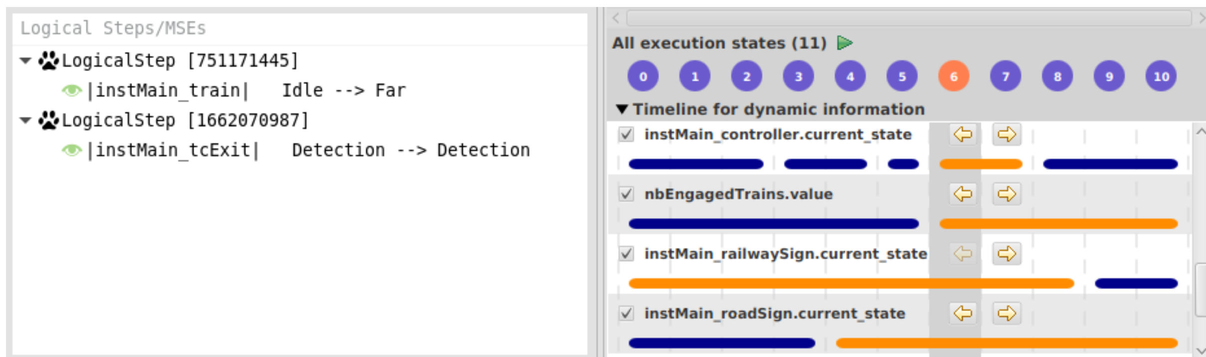



FIGURE 10.14 – Capture d'écran d'une partie de l'interface de Gemoc Studio montrant une session de débogage pour le modèle du contrôleur de passage à niveau.

visualiser les valeurs des RTDs à chaque pas d'exécution. Cette addon affiche également la trace d'exécution courante de sorte qu'il est possible de recharger une configuration précédente en cliquant sur celle-ci (p. ex. la configuration 6 sur la Figure 10.14). Contrairement à OBP2, Gemoc Studio possède une optimisation telle que l'exécution est rejouée, tant que possible, en s'appuyant sur les données de la trace d'exécution. La configuration courante de l'interpréteur n'est rechargée que si une nouvelle trace d'exécution doit être parcourue. Cette optimisation peut néanmoins rendre plus difficile la détection de problèmes de non-déterminisme d'exécution dans l'interpréteur.

Pour intégrer EMI-UML dans Gemoc Studio, l'interface de contrôle d'exécution s'est montrée appropriée pour piloter l'exécution et fournir une projection des RTDs. En ce qui concerne l'observation de l'exécution, un moteur d'exécution dans Gemoc Studio peut être couplé à un "interpréteur d'expressions" notamment pour permettre l'animation de modèles. Il serait tout à fait envisageable d'évaluer des propositions atomiques sur EMI-UML via ce mécanisme afin de répondre aux besoins des activités d'analyse.

## 10.6 Synthèse des expérimentations

Les différentes expérimentations réalisées permettent de répondre aux trois questions V#1, V#2 et V#3 définies pour valider de manière générale l'approche EMI.

**Réponse à V#1 (Conception d'un interpréteur pilotable)** L'implémentation de trois interpréteurs conformes à l'approche EMI permet de conclure que l'effort devant être produit est à la portée des ingénieurs et des chercheurs. En comparaison avec un moteur d'exécution classique, un interpréteur EMI doit implémenter l'interface de contrôle d'exécution, définir une sémantique pilotable et définir une configuration encapsulant les données d'exécution.

**Réponse à V#2 (Généricité de l'interface de contrôle d'exécution)** La diversité des ac-

tivités d'analyse réalisées montre que l'interface de contrôle d'exécution a répondu à ses attentes. Elle permet à la fois d'exposer l'exécution du système et de fournir aux outils d'analyse les informations dont ils ont besoin.

**Réponse à V#3 (Facilité de l'analyse de l'exécution de modèles)** L'approche EMI permet de faciliter l'expression des propositions atomiques, l'expression des propriétés, la compréhension des résultats d'analyse ainsi que la configuration des outils d'analyse grâce à différents opérateurs.

En plus de ces éléments de synthèse, il convient également de lister les menaces à l'encontre de la validité de l'évaluation de l'approche. Une première menace est que les trois interpréteurs conçus n'ont pas été eux-mêmes formellement vérifiés et que leur conformité par rapport à la spécification du langage de modélisation n'a pas non plus été vérifiée. Néanmoins, même si une erreur de conception existe dans l'un de ces interpréteurs, si un système donné satisfait toutes ses exigences en phase d'analyse alors l'approche permet de garantir que ces exigences seront aussi satisfaites lors de l'exécution réelle. Une seconde menace est que les expérimentations ont surtout été réalisées avec des outils développés au sein de notre équipe de recherche ne permettant pas de garantir son applicabilité en dehors de ce cadre. L'interfaçage d'EMI-UML avec Gemoc Studio est un premier pas permettant d'atténuer cette menace.

À travers cette synthèse, les différentes expérimentations réalisées permettent de conclure que l'approche EMI est applicable.

## 10.7 Contributions scientifiques

Cette section décrit chacune des contributions scientifiques revendiquées par cette thèse et apporte des arguments factuels pour les soutenir.

### 10.7.1 C#1 Contribution à l'unification de l'analyse et de l'exécution embarquée de modèles

Cette contribution s'appuie sur quatre sous-contributions afin d'unifier les activités d'analyse et l'exécution réelle sur une cible embarquée.

**C#1.1 Proposition d'une interface de contrôle d'exécution permettant de faire interagir des outils d'analyse avec des moteurs d'exécution.** L'interface de contrôle d'exécution est modulaire et séparée en plusieurs sous-interfaces (STR, APE, APC, $A_{cc}$) afin de s'adapter au mieux aux besoins de chaque activité d'analyse. Pour évaluer cette interface, trois outils d'analyse (OBP2, AnimUML et Gemoc Studio) ont été connectés à au moins un moteur d'exécution via l'interface de contrôle d'exécution. Ces différentes combinaisons d'outils ont été utilisées

pour mener diverses activités d'analyse comme la simulation interactive, l'animation, le débogage multivers, le *model-checking*, ou encore l'exécution comparée. La diversité des activités d'analyse réalisées et des outils employés montre que cette interface de contrôle d'exécution est appropriée pour piloter, observer et visualiser l'exécution de modèles.

**C#1.2 Composition modulaire du système, de l'ordonnanceur et de l'environnement.** La prise en compte de l'ordonnanceur dans la boucle de vérification permet de réduire le problème d'équivalence entre ce qui est exécuté et ce qui est vérifié. La vérification peut ainsi porter sur le système ordonnancé afin d'évaluer si la politique d'ordonnancement choisie permet au système de répondre à toutes ses exigences. Le mécanisme proposé dans cette thèse permet donc de réduire l'effort de vérification (les chemins d'exécution non choisis par l'ordonnanceur n'ont pas besoin d'être vérifiés) et d'améliorer l'efficacité des techniques de *model-checking* (l'espace d'état est généralement plus petit) tout en conservant le niveau de vérification exigé. Pour le passage à l'exécution réelle, seul le modèle abstrait de l'environnement doit être remplacé par l'environnement réel. Le modèle du système reste ainsi inchangé ce qui contribue également à l'unification entre analyse et exécution réelle.

**C#1.3 Unification du *model-checking* et du *monitoring*.** L'approche proposée permet d'utiliser en *monitoring* les mêmes automates observateurs déterministes que ceux utilisés en *model-checking*. Ces automates peuvent être déployés sans transformation ni instrumentation de code ce qui assure que la propriété monitorée est exactement celle qui a été vérifiée en *model-checking*. Cette approche permet ainsi d'assurer la continuité entre la vérification hors-ligne (*model-checking*) et la vérification en ligne (*monitoring*). Le *monitoring* est ainsi complémentaire au *model-checking* puisqu'il est toujours applicable (même en cas d'explosion combinatoire) et qu'il permet d'assurer que les hypothèses considérées pour réaliser l'abstraction de l'environnement sont bien satisfaites à l'exécution. Les expérimentations réalisées avec EMI-UML montrent que le surcoût induit par le *monitoring* est acceptable. De plus, ce surcoût décroît lorsque la taille du système augmente ce qui permet le passage à l'échelle de la solution.

**C#1.4 Élaboration d'une méthodologie de conception d'interpréteurs de modèles pilotables.** Pour mettre en œuvre cette méthodologie de conception, un interpréteur de modèles UML a été développé comme preuve de concept. Cet interpréteur EMI-UML permet à la fois l'exécution embarquée sur une cible embarquée STM32 discovery et le pilotage de l'exécution pour mener des activités d'analyse. Sa conception et le choix du langage C comme langage hôte lui offre des performances comparables à celles du code exécutable produit par les outils de génération de code de la littérature. D'autres interpréteurs de modèles ont également été développés pour attester de la généricité de l'approche. EMI-LOGO est un interpréteur de modèles pour le langage LOGO, un langage procédural avec des paradigmes différents

d'UML. AnimUML est un interpréteur de modèles UML développé en JavaScript ce qui permet d'apprécier l'applicabilité de cette méthodologie avec un autre langage hôte que le langage C.

### 10.7.2 C#2 Contribution à l'adoption des techniques de vérification formelle par les ingénieurs

Cette contribution s'appuie sur trois sous-contributions afin d'aider les ingénieurs à spécifier des propriétés, utiliser des outils de vérification formelle et analyser les résultats obtenus.

**C#2.1 Proposition d'un langage d'observation pour spécifier les propositions atomiques.** Pour faciliter l'expression des propositions atomiques, cette thèse propose un langage d'observation s'appuyant sur les concepts du langage de modélisation. La spécification des atomes ne nécessite pas de connaissances formelles et peut ainsi être effectuée par les ingénieurs. Les atomes peuvent ensuite être utilisés pour du débogage multivers ou du *model-checking*. Grâce à l'interface de contrôle d'exécution, les atomes sont directement évalués sur le moteur d'exécution du système. Ce mécanisme permet de rendre les outils plus génériques en découplant davantage les algorithmes d'analyse des préoccupations liées à l'exécution ou au langage de modélisation.

**C#2.2 Facilitation de la spécification des propriétés formelles.** L'interface de contrôle d'exécution permet de piloter n'importe quel formalisme d'automates y compris des automates spécifiés dans le langage de modélisation. Cette technique a été utilisée avec UML afin de spécifier des automates de Büchi et des automates observateurs à l'aide de machines à états UML appelés PUSMs. Avec cette approche, la spécification des propriétés formelles n'est plus une tâche réservée aux experts en méthodes formelles mais elle devient également accessible aux ingénieurs. L'interface de contrôle d'exécution permet ainsi de s'adapter aux connaissances de chacun en permettant la spécification des propriétés formelles dans divers langages.

**C#2.3 Facilitation de l'utilisation des outils de vérification formelle.** Cette thèse propose une architecture logicielle configurable permettant de connecter des outils de vérification formelle (et plus généralement des outils d'analyse) à un interpréteur de modèles. Différents opérateurs de composition, d'ordonnancement et de filtrage peuvent être utilisés pour s'adapter aux besoins de chaque système et de chaque activité d'analyse. Ces opérateurs, définis en Lean, se présentent sous forme de briques logicielles interconnectables à travers l'interface de contrôle d'exécution.

Grâce à cette interface, les outils d'analyse sont appliqués sur le modèle de conception. Les résultats de vérification sont exprimés directement en termes des concepts du langage de modélisation ce qui facilite la compréhension de ces résultats par les ingénieurs. Les expérimentations avec EMI-UML et OBP2 ont notamment montré que l'analyse des contre-exemples se trouve simplifiée.

Ces observations ont également été confirmées par l'utilisation de l'interpréteur EMI-UML dans plusieurs enseignements avec des étudiants-ingénieurs. Au cours de travaux pratiques, les étudiants ont pu produire différents cas d'études et mettre en œuvre avec succès différentes activités d'analyse (p. ex. simulation, *model-checking* LTL).

## 10.8 Synthèse

L'approche EMI permet de concevoir des interpréteurs de modèles pour différents langages de modélisation dont UML et LOGO. Le développement de ces prototypes a permis d'éprouver la méthodologie de conception mise au point dans cette thèse. Des activités d'analyse diverses et variées ont pu être menées avec ces interpréteurs en allant de la simulation interactive au *monitoring* en passant par le *model-checking* ou l'animation. Ces interpréteurs ont pu être appliqués sur différents cas d'études dont la plupart sont des systèmes embarqués. Les différentes expérimentations réalisées permettent de conclure que l'approche EMI est applicable. L'architecture candidate permet d'atteindre les objectifs fixés. Elle permet d'unifier l'exécution embarquée et l'analyse de modèles tout en offrant des performances d'exécution adéquates et des services adaptés pour les activités d'analyse. Elle permet également d'aider les ingénieurs à spécifier des propriétés formelles et à utiliser les outils de *model-checking* permettant de les vérifier. Les expérimentations réalisées montrent que l'architecture candidate est une solution probante pour résoudre les fossés sémantiques P#1 et P#2 et le problème d'équivalence P#3 tout en offrant de bonnes caractéristiques pour l'analyse et l'exécution embarquée de modèles.

# CONCLUSION

## Rappel du contexte et des problèmes considérés

Les systèmes embarqués sont de plus en plus complexes et nécessitent des besoins croissants en V&V pour assurer leur fiabilité et leur sécurité. Le coût de correction des bogues augmentant exponentiellement au cours du temps, il faut réduire au maximum le nombre de défaillances logicielles en détectant au plus tôt les erreurs de conception et les failles de sécurité. Pour répondre à ce besoin, cette thèse s'est intéressée à définir une solution permettant d'exécuter et d'analyser des modèles conformes à un langage de modélisation donné. Cette solution doit pouvoir être applicable à n'importe quel langage de modélisation et doit permettre l'usage de techniques de vérification formelle afin d'alléger les coûts de test.

L'étude de l'état de l'art a permis d'identifier trois problèmes pouvant être à l'origine de défaillances logicielles : (P#1) un fossé sémantique entre le modèle de conception et le(s) modèle(s) d'analyse, (P#2) un fossé sémantique entre le modèle de conception et le code exécutable, et (P#3) un problème d'équivalence entre ce qui est exécuté et ce qui est vérifié. Ces trois problèmes résultent de l'utilisation de transformations non prouvées (p. ex. transformation de modèles, génération de code) du modèle de conception vers différents langages cibles. Ces transformations définissent ainsi plusieurs implémentations de la sémantique du langage de modélisation sans aucune garantie d'équivalence entre elles.

## Solution proposée

Pour répondre à ces problèmes, cette thèse définit une approche sans transformation permettant d'utiliser une unique implémentation de la sémantique du langage de modélisation. Cette approche, appelée EMI, repose sur un interpréteur de modèles pour encoder la sémantique du langage de modélisation sous forme opérationnelle. Cet interpréteur est pilotable de sorte que tous les outils de développement logiciel puissent contrôler l'exécution du modèle sur l'interpréteur tout en réutilisant la sémantique qu'il encode. Ces travaux de doctorat ont permis de définir plus précisément l'approche EMI et la mise en œuvre d'un interpréteur qui s'y conforme.

L'approche proposée repose notamment sur une interface de contrôle d'exécution permettant de piloter et d'observer l'exécution de modèles sur un interpréteur. Cette interface se compose elle-même de plusieurs sous-interfaces permettant de piloter l'exécution (interface STR),

de collecter des propositions atomiques (interface APC) ou de les évaluer (interface APE), et de déterminer si l'état d'exécution courant est un état d'acceptation (interface $A_{cc}$). Un langage d'observation permet également de spécifier les propositions atomiques avec les concepts du langage de modélisation. Ces atomes sont utiles aux algorithmes d'analyse et sont évalués directement sur l'interpréteur offrant ainsi un meilleur découplage entre moteur d'exécution et outils d'analyse.

Pour mener des activités de développement logiciel, il suffit ensuite de connecter un contrôleur d'exécution à l'interpréteur de modèles via cette interface. Le contrôleur d'exécution peut facilement être changé pour s'adapter aux besoins de chaque activité. Lors de l'exécution réelle, le rôle du contrôleur d'exécution est joué par la boucle principale de la plateforme d'exécution. Lors de l'analyse de modèles, l'exécution est contrôlée soit de manière interactive par l'utilisateur grâce à l'interface graphique de l'outil d'analyse, soit de façon automatique par un algorithme d'analyse. Grâce à ce contrôleur d'exécution interchangeable, les activités d'analyse peuvent être menées sur une plateforme de développement ou sur la plateforme d'exécution réelle. Dans le premier cas, une abstraction de l'environnement du système doit être modélisée pour solliciter l'exécution du système.

Avec l'approche EMI, diverses activités d'analyse portant sur l'exécution du modèle peuvent être menées. Ce projet de thèse est notamment l'un des premiers à avoir mis en œuvre la technique de débogage multivers permettant de déboguer plusieurs chemins d'exécution en parallèle à l'aide de points d'arrêt conditionnels. Ces travaux ont également proposé une architecture logicielle permettant de mettre en œuvre des techniques de vérification formelle comme le *model-checking* ou le *monitoring*. Pour y parvenir, cette architecture repose sur un opérateur de composition synchrone permettant de composer l'exécution de l'automate d'une propriété formelle avec l'exécution du système. Cette architecture est configurable et adaptable aux besoins de chaque activité d'analyse avec différents opérateurs. Ces opérateurs permettent notamment de prendre en compte l'ordonnanceur dans la boucle de vérification ou de filtrer les actions disponibles par exemple pour considérer l'hypothèse de réactivité. L'approche EMI permet ainsi de composer le système, l'ordonnanceur et l'environnement de façon modulaire.

Cette architecture de vérification formelle a notamment été appliquée pour vérifier des propriétés LTL non temporisées par *model-checking*. Dans ce cas, les propriétés LTL sont transformées automatiquement en automates de Büchi et composées de façon synchrone avec l'exécution du système. Cette même architecture a également été employée pour vérifier des propriétés spécifiées directement en UML sous forme de PUSMs. Un PUSM possède une machine à états UML permettant d'encoder des automates observateurs ou des automates de Büchi en UML. Pour chaque propriété, le PUSM correspondant et le système peuvent alors être exécutés par deux instances du même interpréteur de modèles et composés ensemble de façon synchrone. Des algorithmes de *model-checking* peuvent alors être employés pour véri-

fier la propriété encodée par le PUSM sur l'exécution du système. La vérification formelle peut aussi continuer lors de l'exécution réelle en déployant les mêmes automates observateurs déterministes que pour le *model-checking*. Il est ainsi possible de faire du *monitoring* en utilisant une approche sans transformation et sans instrumentation de code. Les moniteurs peuvent ainsi continuer d'assurer la sûreté de fonctionnement du système lors de l'exécution réelle.

D'un point de vue plus technique, cette thèse propose une méthodologie permettant de concevoir et d'implémenter un interpréteur embarqué et pilotable conforme à l'approche EMI. Cette méthodologie consiste à implémenter tous les éléments du langage de modélisation (c.-à-d. le métamodèle et la sémantique) dans un langage hôte afin de pouvoir produire du code exécutable en utilisant les mécanismes de compilation de ce langage hôte. Le modèle de conception est également chargé dans la mémoire de la plateforme d'exécution au moment de la compilation. L'instanciation de ce modèle utilise le mécanisme de promotion afin de réaliser une instanciation physique. Les instances physiques permettent de faire le lien entre le modèle statique et les données d'exécution contenues dans la configuration.

Au travers de ces différents travaux, l'approche EMI propose deux contributions scientifiques. La première contribue à unifier l'analyse et l'exécution embarquée de modèles en utilisant le même couple (modèle + sémantique) pour toutes les activités de développement logiciel. La seconde contribue à faciliter l'adoption des techniques de vérification formelle par les ingénieurs en simplifiant notamment l'expression des propositions atomiques, l'utilisation des outils, et la compréhension des résultats d'analyse.

L'approche EMI permet de répondre aux problèmes identifiés dans cette thèse. En n'utilisant pas de transformations, elle permet d'éviter les fossés sémantiques et le problème d'équivalence entre ce qui est exécuté et ce qui est vérifié. Elle offre une solution permettant l'analyse et l'exécution embarquée de modèles pour n'importe quel langage de modélisation même s'il n'est que semi-formel. Les expérimentations avec EMI montrent que cette approche est fonctionnelle. Elle permet à la fois de préserver de bonnes performances d'exécution et une faible empreinte mémoire tout en fournissant des services pour l'analyse de modèles. Ces résultats sont notamment dus à l'utilisation d'un serveur de langages qui permet d'externaliser certains services d'analyse hors de l'interpréteur. Il fournit notamment des fonctions de projections permettant de visualiser le contenu de la configuration et des pas d'exécution.

D'un point de vue de la méthodologie de développement, l'approche EMI apporte également des avantages intéressants. Le continuum entre conception et exécution permet d'exécuter le système sur une plateforme embarquée dès les phases de conception amont mais aussi d'appliquer les outils d'analyse directement sur l'exécution embarquée du système. L'approche permet également d'alléger les efforts de test en utilisant des techniques de vérification formelle comme le *model-checking*. Les coûts de certification sont réduits puisque l'interpréteur n'a besoin d'être certifié qu'une seule fois pour l'analyse et l'exécution embarquée. Seuls les

modèles doivent ensuite être certifiés comme conformes à leurs spécifications.

L'approche EMI possède de nombreux bénéfices mais même avec cette approche ce qui est exécuté n'est pas exact à 100% à ce qui a été vérifié. Les outils d'analyse reposent sur une abstraction de l'environnement pour fermer l'exécution du système et cette thèse fait l'hypothèse que cette abstraction est conforme à l'environnement réel. Même si cette hypothèse est réaliste dans le cas des systèmes embarqués, elle reste néanmoins à vérifier. Pour assurer la sûreté de fonctionnement du système, l'approche EMI offre une solution de *monitoring* permettant de continuer la vérification des propriétés à l'exécution et de réagir en cas de défaillances. Ce mécanisme peut également être utilisé pour vérifier que les hypothèses considérées pour réaliser l'abstraction de l'environnement sont bien satisfaites lors de l'exécution réelle. Une seconde différence entre analyse et exécution embarquée est que les briques logicielles permettant d'effectuer le pilotage et l'évaluation des atomes doivent être enlevées de l'interpréteur pour l'exécution réelle afin de ne pas laisser de failles de sécurité. Pour réduire les interférences, notre architecture est telle que ces briques sont indépendantes de la sémantique opérationnelle du langage de modélisation. Par ailleurs, l'analyse et l'exécution réelle étant souvent réalisées sur des plateformes d'exécution différentes, des compilateurs différents doivent être utilisés pour produire le code exécutable. Pour le langage C, les compilateurs GCC et Clang reposent sur une forte communauté d'utilisateurs apportant ainsi confiance dans leur utilisation.

## Perspectives

Au cours de ces travaux de thèse, de nombreuses perspectives ont émergé autour de l'approche EMI.

**Mise en œuvre du *live modeling* et du *partial modeling*.** Pour assister au plus tôt les ingénieurs en phase de modélisation, il serait intéressant d'étendre l'approche EMI afin de pouvoir mettre en œuvre du *live modeling* [Bag+19a] et du *partial modeling* [Bag+19b]. Le *live modeling* permet d'adapter la modélisation du système alors même qu'il est en train de s'exécuter. Cette technique supprime le cycle d'édition, de compilation et d'exécution afin d'avoir des retours immédiats sur les changements effectués dans le modèle (c.-à-d. sans avoir besoin de redémarrer le moteur d'exécution). Le *partial modeling* permet de modéliser et d'exécuter des modèles partiels (c.-à-d. incomplets) pour voir la façon dont ils se comportent. La combinaison de ces deux approches offre encore davantage de possibilités en permettant la modélisation et la modification à l'exécution de modèles incomplets. Lors de l'exécution, les ingénieurs doivent pouvoir interagir avec l'interpréteur afin d'ajouter, de modifier ou de supprimer dynamiquement des éléments du modèle. Pour appliquer ces modifications, il semble intéressant d'explorer les

deux techniques suivantes : *(1)* enrichir l'interface de contrôle d'exécution pour éditer le modèle chargé dans l'interpréteur, ou *(2)* utiliser des techniques de chargement du modèle à la volée pour charger en mémoire les modifications effectuées.

Pour le *live modeling*, il est également indispensable d'avoir un mécanisme permettant de retrouver un état consistant. Par exemple, que se passe-t-il si on supprime l'état courant d'une machine à états ? Différentes stratégies sont envisageables comme reprendre l'exécution depuis l'état d'exécution initial ou depuis un état antérieur encore valide. Il pourrait également être intéressant de fournir aux ingénieurs le moyen de choisir les valeurs des données d'exécution afin de définir la configuration utilisée pour reprendre l'exécution. Avec ce dernier mécanisme, il semble indispensable d'avoir des outils d'exploration de l'espace d'état afin de vérifier que l'état d'exécution sélectionné est bien atteignable depuis l'état d'exécution initial du modèle (c.-à-d. qu'il est bien dans l'espace d'état du modèle).

Par ailleurs, il est nécessaire d'étendre la sémantique du langage de modélisation soit pour relâcher certaines contraintes lorsque le modèle est incomplet (cf. AnimUML [Jou+20]) ou pour adapter la sémantique d'exécution au cours du développement afin de la rendre de plus en plus conforme à sa définition (textuelle ou formelle). Avec AnimUML, il est déjà possible de choisir dynamiquement une sémantique d'exécution parmi les variantes supportées pour un point d'exécution donné. Il semble possible d'aller encore plus loin en ajoutant ou en modifiant dynamiquement certaines règles de la sémantique même si ces variantes n'avaient pas été prévues à priori.

Le *live modeling* et le *partial modeling* s'adaptent particulièrement bien avec l'approche EMI qui possède un continuum entre conception et exécution. Divers outils d'analyse (p. ex. simulation, débogage, animation) pourraient être appliqués encore plus tôt lors de la phase de conception et accompagner les ingénieurs tout au long du développement du système.

**Extension aux systèmes temps réel et/ou temporisés.** Il serait également intéressant de pouvoir exécuter des systèmes temps réel et/ou temporisés avec un interpréteur EMI. Il est nécessaire d'avoir pour cela des horloges afin de cadencer l'exécution des différentes unités d'exécution du modèle. En phase d'analyse, la notion de temps doit cependant être abstraite afin d'éviter l'explosion combinatoire. Par exemple, si une unité d'exécution doit attendre la fin d'une temporisation (p. ex. 10 secondes) avant de pouvoir exécuter un pas d'exécution, il est évident que les activités d'analyse comme le *model-checking* prendrait beaucoup trop de temps pour pouvoir être menées jusqu'à leur terme. Pour abstraire le temps, il est possible de raisonner avec une représentation logique ou symbolique (p. ex. avec des *Difference Bounds Matrix* (DBM) [Dil89] permettant de représenter les relations entre les horloges sous forme d'intervalles). Dans le cadre de l'approche EMI, l'un des principaux défis scientifiques est d'arriver à unifier (ou du moins à faire le lien entre) la représentation concrète du temps utilisée lors de

l'exécution réelle et son abstraction utilisée en *model-checking*. Comme pour l'abstraction de l'environnement, il faut alors établir une preuve que l'abstraction réalisée permet de considérer un sur-ensemble de tous les cas concrets. Pour garantir ce résultat à l'exécution, la solution de *monitoring* proposée dans cette thèse pourrait être utilisée.

Avec cette extension aux systèmes temps réel et/ou temporisés, l'approche EMI pourrait servir à mettre en œuvre diverses activités d'analyse comme la vérification de contraintes temporelles via *model-checking*. Il serait aussi possible de déterminer le *Worst Case Execution Time* (WCET) de l'exécution du système en calculant la durée d'exécution de chaque transition puis en cherchant le maximum de toutes les transitions via une exploration de l'espace d'état. Pour ce qui concerne l'exécution réelle, l'interpréteur pourrait être déployé sur un OS temps réel afin de satisfaire au mieux les contraintes temporelles du système.

**Extension aux systèmes distribués.** Une autre perspective d'application serait d'étendre l'approche EMI aux systèmes distribués (ou multicœurs) en utilisant un interpréteur EMI par nœud (ou par cœur).

Pour analyser un système distribué, une première possibilité est d'utiliser les techniques de *model-checking* compositionnel [CLM89] qui consistent à vérifier séparément chacun des nœuds du système. Il faut alors non seulement considérer une abstraction de l'environnement mais aussi une abstraction du reste du système. Pour faciliter cette tâche, des outils [Tka08] permettent de générer automatiquement ces abstractions à partir des interactions entre les nœuds.

Une seconde technique est d'appliquer le *model-checking* sur l'ensemble des nœuds du système. Dans ce cas, la configuration globale est la concaténation des configurations de chaque interpréteur EMI. Pour la reconstituer et ainsi la fournir au *model-checker*, la principale difficulté est d'arriver à synchroniser l'exécution des différents nœuds afin d'obtenir une configuration globale consistante [Bon+12]. Pour exécuter ce type de systèmes, il est également indispensable de pouvoir interconnecter les différents interpréteurs entre eux afin qu'ils puissent s'échanger des messages. Si cette considération est prise en compte dans l'environnement avec le *model-checking* compositionnel, il est nécessaire de la considérer ici d'une façon particulière. Lors de l'exécution réelle, les différents nœuds communiquent via les périphériques de leurs plateformes d'exécution. Une première solution serait donc d'ajouter dans la configuration l'état d'exécution de ces périphériques (p. ex. les registres qu'ils utilisent). Une autre solution est de remplacer ces liens de communication bas niveau par des liens de communication permettant l'échange d'évènements à un haut niveau d'abstraction. Il devient alors nécessaire de garantir que ces échanges haut niveau utilisés lors de l'analyse de modèles sont conformes aux échanges réels.

**Passage à l'échelle.** En vue d'un transfert industriel, l'approche EMI devra être mise en œuvre sur des modèles de taille plus conséquente pour évaluer la robustesse des outils et le passage à l'échelle.

Pour le *model-checking*, il est important de pouvoir repousser l'explosion de l'espace d'état afin de vérifier des systèmes plus complexes. Pour optimiser le stockage de l'espace d'état en mémoire, il serait intéressant d'utiliser des techniques de *model-checking* symbolique (p. ex. en utilisant des BDD [Ake78]), de réduction d'ordre partiel [Hol97 ; BK08], de réduction par symétrie [Bar+17 ; DP04] ou de *Past-Free[ze]* [Teo+16]. Pour réduire la complexité en temps, les expérimentations réalisées avec un *model-checker* matériel sont prometteuses. Cette piste doit être explorée plus en profondeur afin d'améliorer les performances de *model-checking*.

En ce qui concerne l'amélioration des performances d'interprétation de modèles, il semble intéressant d'explorer les possibilités offertes par les techniques d'évaluation partielle [JGS93] pour optimiser l'exécution de l'interpréteur par rapport à un modèle donné.

**Et bien d'autres perspectives.**

*Application à d'autres langages de modélisation.* L'approche EMI a été mise en œuvre sur les langages UML et LOGO au cours de cette thèse mais il serait intéressant d'appliquer l'approche à d'autres DSMLs. Ces expérimentations permettraient d'évaluer l'approche avec des langages ayant différents paradigmes (p. ex. objets, fonctions) et issus de différents secteurs d'activités (p. ex. aéronautique, ferroviaire, automobile, robotique). L'implémentation de nouveaux interpréteurs EMI permettrait également d'utiliser d'autres langages hôtes (p. ex. VHDL, WebAssembly, Rust) et d'autres patrons de conception pour l'implémentation de la sémantique.

*Diversification des activités d'analyse.* La modularité et le découplage des préoccupations dans l'approche EMI permettent de mettre en œuvre une multitude d'activités d'analyse. Cette thèse s'est principalement focalisée sur les activités de simulation, de débogage, de détection de *deadlocks*, de *model-checking* et de *monitoring*. De nombreuses autres activités pourraient théoriquement être appliquées comme du profilage, des techniques basées sur l'exécution symbolique ou concolique, ou encore des techniques d'interprétation abstraite.

*Évaluation empirique.* Pour confirmer les bénéfices offerts par l'approche EMI, une évaluation empirique pourrait être réalisée afin d'évaluer l'utilisabilité de l'approche EMI avec un panel d'ingénieurs. Cette étude permettrait notamment d'évaluer à quel point l'approche EMI facilite l'expression des propositions atomiques et la compréhension des résultats de vérification.

En conclusion, l'approche EMI permet d'unifier l'analyse et l'exécution embarquée de modèles pour n'importe quel langage de modélisation. Elle permet de détecter et de corriger les défaillances logicielles au plus tôt en s'appuyant notamment sur des techniques de vérification formelle. L'approche EMI apporte ainsi des éléments de réponse pour satisfaire les besoins en V&V engendrés par la complexité logicielle croissante des innovations de demain.

# BIBLIOGRAPHIE


[AGR74] Hal ABELSON, Nat GOODMAN et Lee RUDOLPH, « LOGO manual », in : (déc. 1974) (cité p. 180).

[Abr96] Jean-Raymond ABRIAL, *The B-Book : Assigning Programs to Meanings*, USA : Cambridge University Press, 1996, ISBN : 0521496195 (cité p. 43).

[Abr10] Jean-Raymond ABRIAL, *Modeling in Event-B : System and Software Engineering*, 1st, New York, NY, USA : Cambridge University Press, 2010, ISBN : 0521895561 (cité pp. 37, 41, 42, 43).

[Abr+91] Jean-Raymond ABRIAL, Matthew K. O. LEE, Dave NEILSON, P. N. SCHARBACH et Ib SØRENSEN, « The B-Method », in : *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2 : Tutorials*, VDM '91, Berlin, Heidelberg : Springer-Verlag, 1991, p. 398-405, ISBN : 3540548688 (cité p. 43).

[Ake78] Sheldon B. AKERS, « Binary Decision Diagrams », in : *IEEE Transactions on Computers* 27.6 (juin 1978), p. 509-516, ISSN : 0018-9340, DOI : 10.1109/TC.1978.1675141 (cité pp. 39, 195).

[Alu+98] Rajeev ALUR, Thomas A. HENZINGER, Freddy Y. C. MANG, Shaz QADEER, Sriram K. RAJAMANI et Serdar TASIRAN, « MOCHA : Modularity in Model Checking », in : *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, London, UK, UK : Springer-Verlag, 1998, p. 521-525, ISBN : 3540646086 (cité p. 46).

[And09] Charles ANDRÉ, *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*, Research Report RR-6925, INRIA, 2009, p. 37 (cité p. 28).

[Are+10] Thorsten ARENDT, Enrico BIERMANN, Stefan JURACK, Christian KRAUSE et Gabriele TAENTZER, « Henshin : Advanced Concepts and Tools for In-Place EMF Model Transformations », in : *Model Driven Engineering Languages and Systems*, sous la dir. de Dorina C. PETRIU, Nicolas ROUQUETTE et Øystein HAUGEN, Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 121-135, ISBN : 978-3-642-16145-2, DOI : 10.1007/978-3-642-16145-2_9 (cité p. 28).

[Arm07] Joe ARMSTRONG, *Programming Erlang : Software for a Concurrent World*, Pragmatic Bookshelf, 2007, ISBN : 193778553X (cité p. 100).

[AK03] Colin ATKINSON et Thomas KUHNE, « Model-Driven Development : A Metamodeling Foundation », in : *IEEE Software* 20.5 (sept. 2003), p. 36-41, ISSN : 0740-7459, DOI : 10.1109/MS.2003.1231149 (cité p. 131).

[Aue+07] Joshua AUERBACH, David F. BACON, Bob BLAINEY, Perry CHENG, Michael DAWSON, Mike FULTON, David GROVE, Darren HART et Mark STOODLEY, « Design and Implementation of a Comprehensive Real-time Java Virtual Machine », in : *Proceedings of the 7th ACM &Amp ; IEEE International Conference on Embedded Software*, EMSOFT '07, Salzburg, Austria : Association for Computing Machinery, 2007, p. 249-258, ISBN : 978-1-59593-825-1, DOI : 10.1145/1289927.1289967 (cité p. 29).

[Bab+12] Tomáš BABIAK, Mojmír KŘETÍNSKÝ, Vojtěch ŘEHÁK et Jan STREJČEK, « LTL to Büchi Automata Translation : Fast and More Deterministic », in : *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, sous la dir. de Cormac FLANAGAN et Barbara KÖNIG, TACAS'12, Tallinn, Estonia : Springer-Verlag, 2012, p. 95-109, ISBN : 9783642287558, DOI : 10.1007/978-3-642-28756-5_8 (cité pp. 97, 146).

[BHD17] Mojtaba BAGHERZADEH, Nicolas HILI et Juergen DINGEL, « Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems », in : *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, Paderborn, Germany : Association for Computing Machinery, 2017, p. 419-430, ISBN : 9781450351058, DOI : 10.1145/3106237.3106278 (cité p. 48).

[Bag+19a] Mojtaba BAGHERZADEH, Karim JAHED, Benoit COMBEMALE et Juergen DINGEL, « Live-UMLRT : A Tool for Live Modeling of UML-RT Models », in : *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Munich, Germany : IEEE, sept. 2019, p. 743-747, DOI : 10.1109/MODELS-C.2019.00115 (cité p. 192).

[Bag+19b] Mojtaba BAGHERZADEH, Karim JAHED, Nafiseh KAHANI et Juergen DINGEL, « PMExec : An Execution Engine of Partial UML-RT Models », in : *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, San Diego, California : IEEE Press, 2019, p. 1178-1181, ISBN : 9781728125084, DOI : 10.1109/ASE.2019.00131 (cité p. 192).

[BK08] Christel BAIER et Joost-Pieter KATOEN, *Principles of Model Checking (Representation and Mind Series)*, The MIT Press, 2008, ISBN : 026202649X (cité pp. 1, 4, 33, 36, 38, 39, 40, 88, 92, 133, 195).

[Bak+06] Jason BAKER, Antonio CUNEI, Chapman FLACK, Filip PIZLO, Marek PROCHAZKA, Jan VITEK, Austin ARMBRUSTER, Edward PLA et David HOLMES, « A Real-time Java Virtual Machine for Avionics - An Experience Report », in : *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, Washington, DC, USA : IEEE Computer Society, 2006, p. 384-396, ISBN : 0769525164, DOI : 10.1109/RTAS.2006.7 (cité p. 29).

[Bar+17] Zuzana BARANOVÁ, Jiří BARNAT, Katarína KEJSTOVÁ, Tadeáš KUČERA, Henrich LAUKO, Jan MRÁZEK, Petr ROČKAI et Vladimír ŠTILL, « Model Checking of C and C++ with DIVINE 4 », in : *Automated Technology for Verification and Analysis*, sous la dir. de Deepak D'SOUZA et K. NARAYAN KUMAR, Cham : Springer International Publishing, 2017, p. 201-207, ISBN : 978-3-319-68167-2, DOI : 10.1007/978-3-319-68167-2_14 (cité pp. 33, 40, 50, 57, 60, 90, 92, 96, 101, 195).

[BLS11] Andreas BAUER, Martin LEUCKER et Christian SCHALLHART, « Runtime Verification for LTL and TLTL », in : *ACM Transactions on Software Engineering and Methodology* 20.*4* (sept. 2011), p. 1-64, ISSN : 1049-331X, DOI : 10.1145/2000799.2000800 (cité pp. 88, 145).

[Ben+19] Raounak BENABIDALLAH, Salah SADOU, Brendan LE TRIONNAIRE et Isabelle BORNE, « Designing a Code Vulnerability Meta-scanner », in : *Information Security Practice and Experience*, sous la dir. de Swee-Huay HENG et Javier LOPEZ, Cham : Springer International Publishing, 2019, p. 194-210, ISBN : 978-3-030-34339-2, DOI : 10.1007/978-3-030-34339-2_11 (cité p. 33).

[Ber07] Gérard BERRY, « SCADE : Synchronous Design and Validation of Embedded Control Software », in : *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, sous la dir. de S. RAMESH et Prahladavaradan SAMPATH, Dordrecht : Springer Netherlands, 2007, p. 19-33, ISBN : 978-1-4020-6254-4, DOI : 10.1007/978-1-4020-6254-4_2 (cité pp. 26, 58, 100).

[Ber+08] Bernard BERTHOMIEU, Jean-Paul BODEVEIX, Patrick FARAIL, Mamoun FILALI, Hubert GARAVEL, Pierre GAUFILLET, Frederic LANG et François VERNADAT, « Fiacre : an Intermediate Language for Model Verification in the Topcased Environment », in : *4th European Congress ERTS Embedded Real Time Software (ERTS 2008)*, Toulouse, France, jan. 2008 (cité pp. 46, 101).

[BV06] Bernard BERTHOMIEU et Francois VERNADAT, « Time Petri Nets Analysis with TINA », in : *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*, QEST '06, USA : IEEE Computer Society, 2006, p. 123-124, ISBN : 0769526659, DOI : 10.1109/QEST.2006.56 (cité p. 46).

[BC10] Yves Bertot et Pierre Castran, *Interactive Theorem Proving and Program Development : Coq'Art The Calculus of Inductive Constructions*, 1st, Springer Publishing Company, Incorporated, 2010, isbn : 3642058809 (cité pp. 25, 34).

[Bes19] Valentin Besnard, « Unification de la Vérification et de l'Exécution Embarquée de Modèles », in : *Actes des 18èmes journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, AFADL '19, Toulouse, France, juin 2019, p. 93-100 (cité p. 10).

[Bes+17] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier et Ciprian Teodorov, « Towards one Model Interpreter for Both Design and Deployment », in : *3rd International Workshop on Executable Modeling (EXE 2017)*, Austin, United States, sept. 2017 (cité p. 9).

[Bes+19a] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault et Ciprian Teodorov, « EMI : Un Interpréteur de Modèles Embarqué pour l'Exécution et la Vérification de Modèles UML », in : *Actes des 18èmes journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels*, AFADL '19, Toulouse, France, juin 2019, p. 101-104 (cité p. 10).

[Bes+18a] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov et Philippe Dhaussy, « Embedded UML Model Execution to Bridge the Gap Between Design and Runtime », in : *Software Technologies : Applications and Foundations*, sous la dir. de Manuel Mazzara, Iulian Ober et Gwen Salaün, Cham : Springer International Publishing, 2018, p. 519-528, isbn : 978-3-030-04771-9, doi : 10.1007/978-3-030-04771-9_38 (cité p. 10).

[Bes+18b] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov et Philippe Dhaussy, *One Model Interpreter for Simulation, Verification, and Execution of UML Models*, Posters of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '18), oct. 2018 (cité p. 11).

[Bes+18c] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov et Philippe Dhaussy, « Unified LTL Verification and Embedded Execution of UML Models », in : *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, MODELS '18, Copenhagen, Denmark : Association for Computing Machinery, oct. 2018, p. 112-122, isbn : 9781450349499, doi : 10.1145/3239372.3239395 (cité pp. 9, 80, 136, 137, 257).

[Bes+20] Valentin Besnard, Frédéric Jouault, Matthias Brun, Ciprian Teodorov et Philippe Dhaussy, « Modular Deployment of UML Models for V&V Activities and Embedded Execution », in : *Proceedings of the 17th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, Virtual Event, Canada, oct. 2020,

ISBN : 978-1-4503-8135-2/20/10, DOI : 10.1145/3417990.3419227 (cité pp. 10, 139, 261).

[Bes+18d] Valentin BESNARD, Frédéric JOUAULT, Théo LE CALVAR et Massimo TISI, « The TTC 2018 Social Media Case, by ATL and AOF », in : *11th Transformation Tool Contest, co-located with the 2018 Software Technologies : Applications and Foundations (STAF 2018)*, Toulouse, France, juin 2018 (cité p. 10).

[Bes+19b] Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, « A Model Checkable UML Soccer Player », in : *3rd Workshop on Model-Driven Engineering Tools*, Munich, Germany, sept. 2019, p. 211-220 (cité pp. 10, 161).

[Bes+19c] Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, *Verifying and Monitoring UML Models with Observer Automata*, Posters of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS '19), sept. 2019 (cité p. 11).

[Bes+19d] Valentin BESNARD, Ciprian TEODOROV, Frédéric JOUAULT, Matthias BRUN et Philippe DHAUSSY, « Verifying and Monitoring UML Models with Observer Automata : A Transformation-Free Approach », in : *ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS '19)*, MODELS '19, Munich, Germany, sept. 2019, p. 161-171, DOI : 10.1109/MODELS.2019.000-5 (cité pp. 9, 80, 141, 259).

[Béz04] Jean BÉZIVIN, « In search of a Basic Principle for Model-Driven Engineering », in : *Novatica Journal, Special Issue* 5.*2* (2004), p. 21-24 (cité pp. 229, 230, 231, 232).

[BL98] Jean BÉZIVIN et Richard LEMESLE, « Ontology-based Layered Semantics for Precise OA&D Modeling », in : *Proceedings of the Workshops on Object-Oriented Technology*, sous la dir. de Jan BOSCH et Stuart MITCHELL, ECOOP '97, Berlin, Heidelberg : Springer-Verlag, 1998, p. 151-154, ISBN : 3540640398, DOI : 10.1007/3-540-69687-3_32 (cité p. 131).

[Bje05] Per BJESSE, « What is Formal Verification ? », in : *SIGDA Newsletter* 35.*24* (déc. 2005), ISSN : 0163-5743, DOI : 10.1145/1113792.1113794 (cité p. 34).

[BR15] Denis BOGDANAS et Grigore ROŞU, « K-Java : A Complete Semantics of Java », in : *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, Mumbai, India : Association for Computing Machinery, 2015, p. 445-456, ISBN : 9781450333009, DOI : 10.1145/2676726.2676982 (cité pp. 29, 49).

[Bon+12] Frédéric BONIOL, Hugues CASSÉ, Eric NOULARD et Claire PAGETTI, « Deterministic Execution Model on COTS Hardware », in : *Proceedings of the 25th International Conference on Architecture of Computing Systems*, ARCS'12, Munich, Germany : Springer-Verlag, 2012, p. 98-110, ISBN : 9783642282928, DOI : 10.1007/978-3-642-28293-5_9 (cité p. 194).

[BW14] Frédéric BONIOL et Virginie WIELS, « The Landing Gear System Case Study », in : *ABZ 2014 : The Landing Gear Case Study*, Cham : Springer International Publishing, 2014, p. 1-18, ISBN : 978-3-319-07512-9 (cité p. 161).

[Boo86] Grady BOOCH, « Object-Oriented Development », in : *IEEE Transactions on Software Engineering* 12.*1* (jan. 1986), p. 211-221, ISSN : 0098-5589, DOI : 10.1109/TSE.1986.6312937 (cité p. 235).

[Bou15] Erwan BOUSSE, « Execution trace management to support dynamic V&V for executable DSMLs », thèse de doct., Université de Rennes 1, 2015 (cité pp. 33, 45, 230).

[Bou+15] Erwan BOUSSE, Jonathan CORLEY, Benoit COMBEMALE, Jeff GRAY et Benoit BAUDRY, « Supporting Efficient and Advanced Omniscient Debugging for xDSMLs », in : *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, Pittsburgh, PA, USA : Association for Computing Machinery, 2015, p. 137-148, ISBN : 9781450336864 (cité pp. 33, 48).

[Bou+16] Erwan BOUSSE, Thomas DEGUEULE, Didier VOJTISEK, Tanja MAYERHOFER, Julien DEANTONI et Benoit COMBEMALE, « Execution Framework of the GEMOC Studio (Tool Demo) », in : *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, Amsterdam, Netherlands : Association for Computing Machinery, 2016, p. 84-89, ISBN : 9781450344470, DOI : 10.1145/2997364.2997384 (cité pp. 28, 48, 50, 127, 133).

[Bou+17] Erwan BOUSSE, Dorian LEROY, Benoit COMBEMALE, Manuel WIMMER et Benoit BAUDRY, « Omniscient debugging for executable DSLs », in : *Journal of Systems and Software* 137 (mar. 2017), p. 261-288, ISSN : 0164-1212, DOI : 10.1016/j.jss.2017.11.025 (cité pp. 33, 48, 117).

[BMW17] Erwan BOUSSE, Tanja MAYERHOFER et Manuel WIMMER, « Domain-Level Debugging for Compiled DSLs with the GEMOC Studio (Tool Demo) », in : *1rst International Workshop on Debugging in Model-Driven Engineering (MDEbug 2017)*, Austin, United States, sept. 2017 (cité p. 46).

[BW19] Erwan BOUSSE et Manuel WIMMER, « Domain-Level Observation and Control for Compiled Executable DSLs », in : *IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Munich, Germany, sept. 2019 (cité p. 46).

[Bra+00] Guillaume BRAT, Klaus HAVELUND, Seungjoon PARK et Willem VISSER, « Java PathFinder - Second Generation of a Java Model Checker », in : *Proceedings of the Workshop on Advances in Verification*, 2000 (cité pp. 48, 63).

[BGT20] Mihal BRUMBULLI, Emmanuel GAUDIN et Ciprian TEODOROV, « Automatic Verification of BPMN Models », in : *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, jan. 2020 (cité pp. 33, 57).

[Bru+18] Hugo BRUNELIERE, Romina ERAMO, Abel GÓMEZ, Valentin BESNARD, Jean Michel BRUEL, Martin GOGOLLA, Andreas KÄSTNER et Adrian RUTLE, « Model-Driven Engineering for Design-Runtime Interaction in Complex Systems : Scientific Challenges and Roadmap », in : *Software Technologies : Applications and Foundations*, sous la dir. de Manuel MAZZARA, Iulian OBER et Gwen SALAÜN, Cham : Springer International Publishing, 2018, p. 536-543, ISBN : 978-3-030-04771-9, DOI : 10.1007/978-3-030-04771-9_40 (cité p. 10).

[Buc+01] Joseph BUCK, Soonhoi HA, Edward A. LEE et David G. MESSERSCHMITT, « Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems », in : *Readings in Hardware/Software Co-Design*, USA : Kluwer Academic Publishers, 2001, p. 527-543, ISBN : 1558607021 (cité p. 114).

[Bur+05] Sven BURMESTER, Holger GIESE, Martin HIRSCH, Daniela SCHILLING et Matthias TICHY, « The Fujaba Real-time Tool Suite : Model-driven Development of Safety-critical, Real-time Systems », in : *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, St. Louis, MO, USA : ACM, 2005, p. 670-671, ISBN : 1581139632, DOI : 10.1145/1062455.1062601 (cité pp. 25, 238).

[CDE08] Cristian CADAR, Daniel DUNBAR et Dawson ENGLER, « KLEE : Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs », in : *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, San Diego, California : USENIX Association, 2008, p. 209-224 (cité p. 33).

[CS13] Cristian CADAR et Koushik SEN, « Symbolic Execution for Software Testing : Three Decades Later », in : *Communications of the ACM* 56.*2* (fév. 2013), p. 82-90, ISSN : 0001-0782, DOI : 10.1145/2408776.2408795 (cité p. 33).

[Car+09] A. CARACAS, T. KRAMP, M. BAENTSCH, M. OESTREICHER, T. EIRICH et I. ROMANOV, « Mote Runner : A Multi-language Virtual Machine for Small Embedded Devices », in : *Proceedings of the 2009 Third International Conference on Sensor Technologies and Applications*, SENSORCOMM '09, Washington, DC, USA : IEEE Computer Society, 2009, p. 117-125, ISBN : 9780769536699, DOI : 10.1109/SENSORCOMM.2009.27 (cité p. 29).

[Cav+14] Roberto CAVADA, Alessandro CIMATTI, Michele DORIGATTI, Alberto GRIGGIO, Alessandro MARIOTTI, Andrea MICHELI, Sergio MOVER, Marco ROVERI et Stefano TONETTA, « The NuXmv Symbolic Model Checker », in : *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, Berlin, Heidelberg : Springer-Verlag, 2014, p. 334-342, ISBN : 9783319088662, DOI : 10.1007/978-3-319-08867-9_22 (cité p. 46).

[CMB12] Asma CHARFI SMAOUI, Chokri MRAIDHA et Pierre BOULET, « An Optimized Compilation of UML State Machines », in : *Proceedings of the 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '12, USA : IEEE Computer Society, avr. 2012, p. 172-179, ISBN : 9780769546438, DOI : 10.1109/ISORC.2012.30 (cité pp. 25, 238).

[Cha+09] Franck CHAUVEL, Olivier BARAIS, Jean-Marc JÉZÉQUEL et Isabelle BORNE, « Un processus à base de modèles pour les systèmes auto-adaptatifs », in : *REE - Revue de l'électricité électronique 2* (2009), p. 38-44 (cité p. 238).

[CDR04] Feng CHEN, Marcelo D'AMORIM et Grigore ROŞU, « A Formal Monitoring-Based Framework for Software Development and Analysis », in : *Formal Methods and Software Engineering*, sous la dir. de Jim DAVIES, Wolfram SCHULTE et Mike BARNETT, Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 357-372, ISBN : 978-3-540-30482-1, DOI : 10.1007/978-3-540-30482-1_31 (cité p. 34).

[Cic14] Federico CICCOZZI, « From Models to Code and Back : A Round-trip Approach for Model-driven Engineering of Embedded Systems », thèse de doct., Mälardalen University, Embedded Systems, 2014, p. 140, ISBN : 978-91-7485-129-8 (cité pp. 45, 46).

[Cic18] Federico CICCOZZI, « Unicomp : A Semantics-aware Model Compiler for Optimised Predictable Software », in : *Proceedings of the 40th International Conference on Software Engineering : New Ideas and Emerging Results*, ICSE-NIER '18, Gothenburg, Sweden : Association for Computing Machinery, 2018, p. 41-44, ISBN : 9781450356626, DOI : 10.1145/3183399.3183406 (cité pp. 25, 238).

[CMS18] Federico CICCOZZI, Ivano MALAVOLTA et Bran SELIC, « Execution of UML Models : A Systematic Review of Research and Practice », in : *Software & Systems Modeling* 18.*3* (juin 2018), p. 2313-2360, ISSN : 1619-1366, DOI : 10.1007/s10270-018-0675-4 (cité pp. 237, 239).

[Cim+02] Alessandro CIMATTI, Edmund M. CLARKE, Enrico GIUNCHIGLIA, Fausto GIUNCHIGLIA, Marco PISTORE, Marco ROVERI, Roberto SEBASTIANI et Armando TACCHELLA, « NuSMV 2 : An OpenSource Tool for Symbolic Model Checking », in : *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, Berlin, Heidelberg : Springer-Verlag, 2002, p. 359-364, ISBN : 3540439978 (cité pp. 46, 96).

[CE81] Edmund M. CLARKE et E. Allen EMERSON, « Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic », in : *Logics of Programs*, sous la dir. de Dexter KOZEN, Berlin, Heidelberg : Springer-Verlag, 1981, p. 52-71, ISBN : 978-3-540-39047-3 (cité p. 36).

[CGL94] Edmund M. CLARKE, Orna GRUMBERG et David E. LONG, « Model Checking and Abstraction », in : *ACM Transactions on Programming Languages and Systems* 16.*5* (sept. 1994), p. 1512-1542, ISSN : 0164-0925, DOI : 10.1145/186025.186051 (cité p. 33).

[CLM89] Edmund M. CLARKE, David E. LONG et Kenneth L. MCMILLAN, « Compositional Model Checking », in : *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, USA : IEEE Press, juin 1989, p. 353-362, ISBN : 0818619546, DOI : 10.1109/LICS.1989.39190 (cité pp. 109, 194).

[Cla+05] Edmund CLARKE, Ansgar FEHNKER, Sumit Kumar JHA et Helmut VEITH, « Temporal Logic Model Checking », in : *Handbook of Networked and Embedded Control Systems*, sous la dir. de Dimitrios HRISTU-VARSAKELIS et William S. LEVINE, Boston, MA : Birkhäuser Boston, 2005, p. 539-558, ISBN : 978-0-8176-4404-8, DOI : 10.1007/0-8176-4404-0_23 (cité p. 35).

[Cla+03] Manuel CLAVEL, Francisco DURÁN, Steven EKER, Patrick LINCOLN, Narciso MARTÍ-OLIET, José MESEGUER et Carolyn TALCOTT, « The Maude 2.0 System », in : *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, RTA'03, Valencia, Spain : Springer-Verlag, 2003, p. 76-87, ISBN : 3540402543 (cité pp. 46, 48, 96).

[Cle09] CLEARSY SYSTEM ENGINEERING, *Obligations de preuve Manuel de référence version 3.7*, 2009 (cité p. 42).

[Cli+00] Curtis CLIFTON, Gary T. LEAVENS, Craig CHAMBERS et Todd MILLSTEIN, « MultiJava : Modular Open Classes and Symmetric Multiple Dispatch for Java », in : *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, Minneapolis, Minnesota, USA : Association for Computing Machinery, 2000, p. 130-145, ISBN : 158113200X, DOI : 10.1145/353171.353181 (cité p. 28).

[Com08] Benoît COMBEMALE, « Approche de métamodélisation pour la simulation et la vérification de modèle : application à l'ingénierie des procédés », thèse de doct., Institut National Polytechnique de Toulouse - INPT, 2008, 1 vol. (195 p.) (Cité pp. 16, 17, 18, 32, 33, 230, 232).

[Com+09] Benoît COMBEMALE, Xavier CRÉGUT, Pierre-Loïc GAROCHE et Xavier THIRIOUX, « Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification », in : *Journal of Software (JSW)* 4.*9* (nov. 2009), p. 943-958 (cité pp. 26, 58).

[Com+13] Benoît COMBEMALE, Julien DE ANTONI, Matias Vara LARSEN, Frédéric MALLET, Olivier BARAIS, Benoit BAUDRY et Robert B. FRANCE, « Reifying Concurrency for Executable Metamodeling », in : *Software Language Engineering*, sous la dir. de Martin ERWIG, Richard F. PAIGE et Eric VAN WYK, Cham : Springer International Publishing, 2013, p. 365-384, ISBN : 978-3-319-02654-1, DOI : 10.1007/978-3-319-02654-1_20 (cité p. 28).

[CGR11] Benoît COMBEMALE, Laure GONNORD et Vlad RUSU, « A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics », in : *Modelling Foundations and Applications*, sous la dir. de Robert B. FRANCE, Jochen M. KUESTER, Behzad BORDBAR et Richard F. PAIGE, Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, p. 35-51, ISBN : 978-3-642-21470-7 (cité p. 46).

[Con+18] Sylvain CONCHON, Albin COQUEREAU, Mohamed IGUERNLALA et Alain MEBSOUT, « Alt-Ergo 2.2 », in : *SMT Workshop : International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, juil. 2018 (cité p. 34).

[Cor+00] James C. CORBETT, Matthew B. DWYER, John HATCLIFF, Shawn LAUBACH, Corina S. PĂSĂREANU, ROBBY et Hongjun ZHENG, « Bandera : Extracting Finite-State Models from Java Source Code », in : *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, Limerick, Ireland : Association for Computing Machinery, 2000, p. 439-448, ISBN : 1581132069, DOI : 10.1145/337180.337234 (cité p. 46).

[CC77] Patrick COUSOT et Radhia 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*, POPL '77, Los Angeles, California : Association for Computing Machinery, 1977, p. 238-252, ISBN : 9781450373500, DOI : 10.1145/512950.512973 (cité p. 33).

[CD08] Michelle L. CRANE et Juergen DINGEL, « Towards a UML Virtual Machine : Implementing an Interpreter for UML 2 Actions and Activities », in : *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research : Meeting of Minds*, CASCON '08, Ontario, Canada : Association for Computing Machinery, 2008, ISBN : 9781450378826, DOI : 10.1145/1463788.1463799 (cité pp. 27, 238).

[Dea+15] Julien DEANTONI, Issa Papa DIALLO, Ciprian TEODOROV, Joel CHAMPEAU et Benoit COMBEMALE, « Towards a Meta-language for the Concurrency Concern in DSLs », in : *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, Grenoble, France : EDA Consortium, 2015, p. 313-316, ISBN : 9783981537048 (cité p. 28).

[DM12] Julien DEANTONI et Frédéric MALLET, « TimeSquare : Treat Your Models with Logical Time », in : *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, Prague, Czech Republic : Springer-Verlag, 2012, p. 34-41, ISBN : 9783642305603, DOI : 10.1007/978-3-642-30561-0_4 (cité p. 28).

[Dév+15] Gergely DÉVAI, Máté KARÁCSONY, Boldizsár NÉMETH, Róbert KITLEI et Tamás KOZSIK, « UML Model Execution via Code Generation », in : *1st International Workshop on Executable Modeling (EXE 2015)*, 2015 (cité p. 29).

[DRB11] P. DHAUSSY, J. ROGER et F. BONIOL, « Reducing State Explosion with Context Modeling for Model-Checking », in : *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, nov. 2011, p. 130-137, DOI : 10.1109/HASE.2011.24 (cité p. 3).

[DLT14] Philippe DHAUSSY, Luka LE ROUX et Ciprian TEODOROV, « Vérification formelle de propriétés : Application de l'outil OBP au cas d'étude CCS », in : *Génie logiciel* 109 (juin 2014) (cité p. 260).

[Dha+12] Philippe DHAUSSY, Jean-Charles ROGER, Luka LEROUX et Frédéric BONIOL, « Context Aware Model Exploration with OBP tool to Improve Model-Checking », in : *ERTS 2012*, Toulouse, France, fév. 2012 (cité pp. 3, 37, 45, 90, 92).

[Dil89] David L. DILL, « Timing Assumptions and Verification of Finite-state Concurrent Systems », in : *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France : Springer-Verlag, 1989, p. 197-212, ISBN : 3540521488, DOI : 10.1007/3-540-52148-8_17 (cité p. 193).

[DSH94] Christophe DIOT, Robert de SIMONE et Christian HUITEMA, « Communication Protocols Development Using ESTEREL », in : *First International HIPPARCH workshop. INRIA Sophia Antipolis*, déc. 1994, p. 15-16 (cité p. 109).

[Dre+09] Zoé DREY, Cyril FAUCHER, Franck FLEUREY, Vincent MAHÉ et Didier VOJTISEK, *Kermeta language - Reference manual*, 2009 (cité p. 28).

[DT16] Zoé DREY et Ciprian TEODOROV, « Object-oriented Design Pattern for DSL Program Monitoring », in : *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, Amsterdam, Netherlands : Association for Computing Machinery, 2016, p. 70-83, ISBN : 9781450344470, DOI : 10.1145/2997364.2997373 (cité p. 56).

[Dru00] Doron DRUSINSKY, « The Temporal Rover and the ATG Rover », in : *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, sous la dir. de Klaus HAVELUND, John PENIX et Willem VISSER, Berlin, Heidelberg : Springer-Verlag, 2000, p. 323-330, ISBN : 3540410309, DOI : 10.1007/10722468_19 (cité p. 34).

[Dub+13] Catherine DUBOIS, Michalis FAMELIS, Martin GOGOLLA, Leonel NOBREGA, Ileana OBER, Martina SEIDL et Markus VÖLTER, « Research Questions for Validation and Verification in the Context of Model-Based Engineering », in : *International Workshop on Model Driven Engineering, Verification and Validation - MoDeVVA 2013*, t. 1069, Miami, USA : CEUR Workshop Proceedings, oct. 2013, p. 67-76 (cité p. 56).

[DP04] Alexandre DURET-LUTZ et Denis POITRENAUD, « SPOT : An Extensible Model Checking Library Using Transition-Based Generalized Büchi Automata », in : *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '04, Washington, DC, USA : IEEE Computer Society, 2004, p. 76-83, ISBN : 0769522513, DOI : 10.1109/MASCOT.2004.1348184 (cité pp. 50, 61, 90, 92, 96, 97, 195).

[DK92] Eugène DÜRR et Jan van KATWIJK, « VDM++ : A Formal Specification Language for Object-Oriented Designs », in : *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems*, TOOLS 7, Dort-

mund, Germany : Prentice Hall International (UK) Ltd., 1992, p. 63-77, ISBN : 0139174362 (cité p. 43).

[DAC98] Matthew B. DWYER, George S. AVRUNIN et James C. CORBETT, « Patterns in Property Specifications for Finite-State Verification », in : *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, Los Angeles, California, USA : Association for Computing Machinery, 1998, p. 411-420, ISBN : 1581130740, DOI : 10.1145/302405.302672 (cité p. 96).

[ER12] Chucky ELLISON et Grigore ROŞU, « An Executable Formal Semantics of C with Applications », in : *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, Philadelphia, PA, USA : Association for Computing Machinery, 2012, p. 533-544, ISBN : 9781450310833, DOI : 10.1145/2103656.2103719 (cité p. 49).

[EHV95] RichardHelm ERICH GAMMA, Richard HELM et John VLISSIDES, « Design Patterns : Elements of Reusable Object-Oriented Software », in : (1995) (cité pp. 27, 134, 135, 142, 180).

[EB10] Moritz EYSHOLDT et Heiko BEHRENS, « Xtext : Implement Your Language Faster than the Quick and Dirty Way », in : *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, Reno/Tahoe, Nevada, USA : Association for Computing Machinery, 2010, p. 307-309, ISBN : 9781450302401, DOI : 10.1145/1869542.1869625 (cité p. 16).

[Far+06] Patrick FARAIL, Pierre GAUFILLET, Agusti CANALS, Christophe LE CAMUS, David SCIAMMA, Pierre MICHEL, Xavier CRÉGUT et Marc PANTEL, « The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design », in : *Embedded Real Time Software (ERTS)*, Toulouse, fév. 2006 (cité p. 46).

[Fer90] Jean-Claude FERNANDEZ, « An Implementation of an Efficient Algorithm for Bisimulation Equivalence », in : *Science of Computer Programming* 13.*2–3* (avr. 1990), p. 219-236, ISSN : 0167-6423, DOI : 10.1016/0167-6423(90)90071-K (cité pp. 58, 182).

[Fer+92] Jean-Claude FERNANDEZ, Hubert GARAVEL, Laurent MOUNIER, Anne RASSE, Carlos RODRIGUEZ et Joseph SIFAKIS, « A Toolbox for the Verification of LOTOS Programs », in : *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, Melbourne, Australia : Association for Computing Machinery, 1992, p. 246-259, ISBN : 0897915046, DOI : 10.1145/143062.143124 (cité p. 34).

[FDD17] Jacopo FERRETTI, Licia DI PIETRO et Carmelo DE MARIA, « Open-source automated external defibrillator », in : *HardwareX* 2 (2017), p. 61-70, ISSN : 2468-0672, DOI : 10.1016/j.ohx.2017.09.001 (cité p. 161).

[FP13] Jean-Christophe FILLIÂTRE et Andrei PASKEVICH, « Why3 — Where Programs Meet Provers », in : *Programming Languages and Systems*, sous la dir. de Matthias FELLEISEN et Philippa GARDNER, ESOP'13, Rome, Italy : Springer-Verlag, 2013, p. 125-128, ISBN : 9783642370359, DOI : 10.1007/978-3-642-37036-6_8 (cité p. 34).

[FW99] Clemens FISCHER et Heike WEHRHEIM, « Model-Checking CSP-OZ Specifications with FDR », in : *Proceedings of the 1st International Conference on Integrated Formal Methods*, sous la dir. de Keijiro ARAKI, Andy GALLOWAY et Kenji TAGUCHI, IFM '99, Berlin, Heidelberg : Springer-Verlag, 1999, p. 315-334, ISBN : 1852331070, DOI : 10.1007/978-1-4471-0851-1_17 (cité p. 46).

[FLS+08] John FITZGERALD, Peter Gorm LARSEN, Shin SAHARA et al., « VDMTools : Advances in Support for Formal Modeling in VDM », in : *SIGPLAN Notices* 43.*2* (fév. 2008), p. 3-11, ISSN : 0362-1340, DOI : 10.1145/1361213.1361214 (cité p. 43).

[FTL20] Émilien FOURNIER, Ciprian TEODOROV et Loïc LAGADEC, « Menhir : Generic High-Speed FPGA Model-Checker », in : *Proceedings of the 23th EUROMICRO Conference on Digital System Design*, DSD '20, USA : IEEE Computer Society, août 2020 (cité p. 168).

[Fra+98] Robert B. FRANCE, Andy EVANS, Kevin LANO et Bernhard RUMPE, « The UML as a Formal Modeling Notation », in : *Computer Standards & Interfaces* 19.*7* (1998), p. 325-334, ISSN : 0920-5489, DOI : 10.1016/S0920-5489(98)00020-8 (cité pp. 25, 237, 240).

[FM89] P. K. D. FROOME et B. Q. MONAHAN, « Specbox : a toolkit for BSI-VDM », in : *Second International Conference on Software Engineering for Real Time Systems, 1989.* Sept. 1989, p. 50-54 (cité p. 43).

[FMS08] Lidia FUENTES, Jorge MANRIQUE et Pablo SÁNCHEZ, « Execution and Simulation of (Profiled) UML Models Using PóPulo », in : *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, Leipzig, Germany : Association for Computing Machinery, 2008, p. 75-81, ISBN : 9781605580258, DOI : 10.1145/1370731.1370749 (cité pp. 27, 238).

[GS09] Andreas GAISER et Stefan SCHWOON, « Comparison of Algorithms for Checking Emptiness on Büchi Automata », in : *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, sous la dir.

de Petr HLINENÝ, Václav MATYÁŠ et Tomáš VOJNAR, t. 13, OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, p. 18-26, ISBN : 978-3-939897-15-6, DOI : 10.4230/DROPS.MEMICS.2009.2349 (cité pp. 39, 78, 97, 152, 174).

[Gar+13] Hubert GARAVEL, Frédéric LANG, Radu MATEESCU et Wendelin SERWE, « CADP 2011 : A Toolbox for the Construction and Analysis of Distributed Processes », in : *International Journal on Software Tools for Technology Transfer* 15.*2* (2013), p. 89-107, ISSN : 1433-2779, DOI : 10.1007/s10009-012-0244-z (cité p. 46).

[Gar+05] Guillaume GARDEY, Didier LIME, Morgan MAGNIN et Olivier H. ROUX, « Romeo : A Tool for Analyzing Time Petri Nets », in : *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, Edinburgh, Scotland, UK : Springer-Verlag, 2005, p. 418-423, ISBN : 3540272313, DOI : 10.1007/11513988_41 (cité p. 46).

[GO01] Paul GASTIN et Denis ODDOUX, « Fast LTL to Büchi Automata Translation », in : *Proceedings of the 13th International Conference on Computer Aided Verification*, sous la dir. de Gérard BERRY, Hubert COMON et Alain FINKEL, CAV '01, Berlin, Heidelberg : Springer-Verlag, 2001, p. 53-65, ISBN : 3540423451, DOI : 10.1007/3-540-44585-4_6 (cité p. 146).

[GFL05] Frédéric GERVAIS, Marc FRAPPIER et Régine LALEAU, *Vous avez dit raffinement ?*, Research Report CEDRIC-05-829, CEDRIC Lab/CNAM, 2005 (cité p. 41).

[GMB09] Tahar GHERBI, Djamel MESLATI et Isabelle BORNE, « MDE between Promises and Challenges », in : *Proceedings of the UKSim 2009 : 11th International Conference on Computer Modelling and Simulation*, UKSIM '09, USA : IEEE Computer Society, 2009, p. 152-155, ISBN : 9780769535937, DOI : 10.1109/UKSIM.2009.13 (cité p. 232).

[GM04] Stefania GNESI et Franco MAZZANTI, « On the fly model checking of communicating UML State Machines », in : *Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2004)*, mai 2004, p. 331-338 (cité pp. 48, 240).

[GM05] Stefania GNESI et Franco MAZZANTI, « A model checking verification environment for UML Statecharts », in : *Proceedings of XLIII Congresso*, 2005 (cité pp. 48, 240).

[GDT14] Teofilo GONZALEZ, Jorge DIAZ-HERRERA et Allen TUCKER, *Computing Handbook, Third Edition : Computer Science and Software Engineering*, 3rd, Chapman & Hall/CRC, 2014, ISBN : 1439898529 (cité pp. 19, 20).

[HK04] David HAREL et Hillel KUGLER, « The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) », in : *Integration of Software Specification Techniques for Applications in Engineering : Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, sous la dir. d'Hartmut EHRIG, Werner DAMM, Jörg DESEL, Martin GROSSE-RHODE, Wolfgang REIF, Eckehard SCHNIEDER et Engelbert WESTKÄMPER, Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 325-354, ISBN : 978-3-540-27863-4, DOI : 10.1007/978-3-540-27863-4_19 (cité pp. 25, 179, 235, 238, 239).

[Hat+03] John HATCLIFF, Xinghua DENG, Matthew B. DWYER, Georg JUNG et Venkatesh Prasad RANGANATH, « Cadena : An Integrated Development, Analysis, and Verification Environment for Component-Based Systems », in : *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, Portland, Oregon : IEEE Computer Society, 2003, p. 160-173, ISBN : 076951877X, DOI : 10.1109/ICSE.2003.1201197 (cité p. 46).

[HER15] Chris HATHHORN, Chucky ELLISON et Grigore ROŞU, « Defining the Undefinedness of C », in : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, Portland, OR, USA : Association for Computing Machinery, 2015, p. 336-345, ISBN : 9781450334686, DOI : 10.1145/2737924.2737979 (cité p. 29).

[HP00] Klaus HAVELUND et Thomas PRESSBURGER, « Model checking Java programs using Java PathFinder », in : *International Journal on Software Tools for Technology Transfer* 2.4 (mar. 2000), p. 366-381, ISSN : 1433-2779, DOI : 10.1007/s100090050043 (cité p. 46).

[Heg+10] Ábel HEGEDÜS, Gabor BERGMANN, István RÁTH et Dániel VARRÓ, « Back-annotation of Simulation Traces with Change-Driven Model Transformations », in : *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, USA : IEEE Computer Society, sept. 2010, p. 145-155, ISBN : 9780769541532, DOI : 10.1109/SEFM.2010.28 (cité p. 45).

[HDB17] Nicolas HILI, Juergen DINGEL et Alain BEAULIEU, « Modelling and Code Generation for Real-Time Embedded Systems with UML-RT and Papyrus-RT », in : *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, ICSE-C '17, Buenos Aires, Argentina : IEEE Press, mai 2017, p. 509-510, ISBN : 9781538615898, DOI : 10.1109/ICSE-C.2017.168 (cité pp. 25, 48, 238).

[Hoa78] C. A. R. HOARE, « Communicating Sequential Processes », in : *Communications of the ACM* 21.*8* (août 1978), p. 666-677, ISSN : 0001-0782, DOI : 10.1145/359576.359585 (cité pp. 101, 102).

[Hol97] Gerard J. HOLZMANN, « The model checker SPIN », in : *IEEE Transactions on Software Engineering* 23.*5* (mai 1997), p. 279-295, ISSN : 0098-5589, DOI : 10.1109/32.588521 (cité pp. 33, 40, 45, 59, 90, 92, 101, 195, 239).

[Hol00] Gerard J. HOLZMANN, « Logic Verification of ANSI-C Code with SPIN », in : *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, sous la dir. de Klaus HAVELUND, John PENIX et Willem VISSER, Berlin, Heidelberg : Springer-Verlag, 2000, p. 131-147, ISBN : 3540410309, DOI : 10.1007/10722468_8 (cité p. 96).

[ISO15] ISO, *Systèmes de management de la qualité — Principes essentiels et vocabulaire*, 2015, URL : https://www.iso.org/fr/standard/45481.html (cité p. 32).

[JSS00] Daniel JACKSON, Ian SCHECHTER et Hya SHLYAHTER, « Alcoa : The Alloy Constraint Analyzer », in : *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, Limerick, Ireland : Association for Computing Machinery, 2000, p. 730-733, ISBN : 1581132069, DOI : 10.1145/337180.337616 (cité p. 46).

[Jac+92] Ivar JACOBSON, Magnus CHRISTERSON, Patrik JONSSON et Gunnar OVERGAARD, *Object Oriented Software Engineering : A Use Case Driven Approach*, Addison-Wesley, 1992 (cité p. 235).

[Jäg+16] Sven JÄGER, Ralph MASCHOTTA, Tino JUNGEBLOUD, Alexander WICHMANN et Armin ZIMMERMANN, « An EMF-like UML generator for C++ », in : *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, fév. 2016, p. 309-316 (cité pp. 25, 238).

[JBF11] Jean-Marc JÉZÉQUEL, Olivier BARAIS et Franck FLEUREY, « Model Driven Language Engineering with Kermeta », in : *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, sous la dir. de João M. FERNANDES, Ralf LÄMMEL, Joost VISSER et João SARAIVA, GTTSE'09, Braga, Portugal : Springer-Verlag, 2011, p. 201-221, ISBN : 3642180221, DOI : 10.1007/978-3-642-18023-1_5 (cité pp. 27, 28).

[Jéz+15] Jean-Marc JÉZÉQUEL, Benoit COMBEMALE, Olivier BARAIS, Martin MONPERRUS et François FOUQUET, « Mashup of metalanguages and its implementation in the Kermeta language workbench », in : *Software & Systems Modeling* 14.*2* (mai

2015), p. 905-920, ISSN : 1619-1366, DOI : 10.1007/s10270-013-0354-4 (cité p. 135).

[Jon95] Cliff B JONES, *Systematic software development using VDM*, t. 2, Prentice Hall International, juin 1995 (cité p. 43).

[JGS93] Neil D. JONES, Carsten K. GOMARD et Peter SESTOFT, *Partial Evaluation and Automatic Program Generation*, USA : Prentice-Hall, Inc., 1993, ISBN : 0130202495 (cité p. 195).

[Jou05] Frédéric JOUAULT, « Loosely Coupled Traceability for ATL », in : *European Conference on Model Driven Architecture (ECMDA) Workshop on Traceability*, Germany, 2005, p. 29-37 (cité p. 45).

[Jou06] Frédéric JOUAULT, « Contribution à l'étude des langages de transformation de modèles », thèse de doct., Université de Nantes, 2006, 1 vol. (175 p.) (Cité pp. 24, 230, 231, 232, 233).

[Jou+20] Frédéric JOUAULT, Valentin BESNARD, Théo LE CALVAR, Ciprian TEODOROV, Matthias BRUN et Jérôme DELATOUR, « Designing, Animating, and Verifying Partial UML Models », in : *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20)*, MODELS '20, Virtual Event, Canada, oct. 2020, ISBN : 978-1-4503-7019-6/20/10, DOI : 10.1145/3365438.3410967 (cité pp. 9, 181, 193).

[JB06] Frédéric JOUAULT et Jean BÉZIVIN, « KM3 : A DSL for Metamodel Specification », in : *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, sous la dir. de Roberto GORRIERI et Heike WEHRHEIM, FMOODS'06, Bologna, Italy : Springer-Verlag, 2006, p. 171-185, ISBN : 354034893X, DOI : 10.1007/11768869_14 (cité pp. 131, 230).

[JBK06] Frédéric JOUAULT, Jean BÉZIVIN et Ivan KURTEV, « TCS : A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering », in : *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, Portland, Oregon, USA : Association for Computing Machinery, 2006, p. 249-254, ISBN : 1595932372, DOI : 10.1145/1173706.1173744 (cité p. 16).

[JD14] Frédéric JOUAULT et Jérôme DELATOUR, « Towards Fixing Sketchy UML Models by Leveraging Textual Notations : Application to Real-Time Embedded Systems », in : *OCL 2014*, sous la dir. d'Achim D. BRUCKER, Carolina DANIA, Geri GEORG et Martin GOGOLLA, t. 1285, OCL and Textual Modeling : Applications and Case Studies, Valencia, Spain, sept. 2014, p. 73-82 (cité p. 146).

[JK06] Frédéric JOUAULT et Ivan KURTEV, « Transforming Models with ATL », in : *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, sous la dir. de Jean-Michel BRUEL, MoDELS'05, Montego Bay, Jamaica : Springer-Verlag, 2006, p. 128-138, ISBN : 3540317805, DOI : 10.1007/11663430_14 (cité pp. 46, 233).

[Jou+14] Frédéric JOUAULT, Ciprian TEODOROV, Jérôme DELATOUR, Luka LE ROUX et Philippe DHAUSSY, « Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD », in : *Génie logiciel* 109 (juin 2014), p. 21-27 (cité pp. 146, 240).

[Kah87] G. KAHN, « Natural semantics », in : *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, sous la dir. de Franz J. BRANDENBURG, Guy VIDAL-NAQUET et Martin WIRSING, Berlin, Heidelberg : Springer-Verlag, 1987, p. 22-39, ISBN : 978-3-540-47419-7, DOI : 10.1007/BFb0039592 (cité p. 19).

[Kan+15] Gijs KANT, Alfons LAARMAN, Jeroen MEIJER, Jaco POL, Stefan BLOM et Tom DIJK, « LTSmin : High-Performance Language-Independent Model Checking », in : *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg : Springer-Verlag, 2015, p. 692-707, ISBN : 9783662466803, DOI : 10.1007/978-3-662-46681-0_61 (cité pp. 50, 61, 96, 123).

[KV10] Lennart C. L. KATS et Eelco VISSER, « The Spoofax Language Workbench : Rules for Declarative Specification of Languages and IDEs », in : *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, Reno/Tahoe, Nevada, USA : Association for Computing Machinery, 2010, p. 444-463, ISBN : 9781450302036, DOI : 10.1145/1869459.1869497 (cité p. 234).

[Kim+04] MoonZoo KIM, Mahesh VISWANATHAN, Sampath KANNAN, Insup LEE et Oleg SOKOLSKY, « Java-MaC : A Run-Time Assurance Approach for Java Programs », in : *Formal Methods in System Design* 24.2 (mar. 2004), p. 129-155, ISSN : 0925-9856, DOI : 10.1023/B:FORM.0000017719.43755.7c (cité p. 34).

[KRW04] Ekkart KINDLER, Vladimir RUBIN et Robert WAGNER, « An Adaptable TGG Interpreter for In-Memory Model Transformation », in : *Proceedings of the 2nd International Fujaba Days*, sept. 2004, p. 35-38 (cité p. 29).

[Kir+15] Florent KIRCHNER, Nikolai KOSMATOV, Virgile PREVOSTO, Julien SIGNOLES et Boris YAKOBOWSKI, « Frama-C : A software analysis perspective », in : *Formal As-*

*pects of Computing* 27.*3* (mai 2015), p. 573-609, ISSN : 1433-299X, DOI : 10.1007/s00165-014-0326-7 (cité pp. 21, 33).

[KDH07] Andrei KIRSHIN, Dolev DOTAN et Alan HARTMAN, « A UML Simulator Based on a Generic Model Execution Engine », in : *Models in Software Engineering : Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, sous la dir. de Thomas KÜHNE, Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 324-326, ISBN : 978-3-540-69489-2, DOI : 10.1007/978-3-540-69489-2_40 (cité pp. 25, 238).

[Kle07] Anneke KLEPPE, « A Language Description is More than a Metamodel », in : *Fourth international workshop on Software Language Engineering*, t. 1, oct. 2007 (cité pp. 16, 18, 19, 20, 21).

[KMR02] Alexander KNAPP, Stephan MERZ et Christopher RAUH, « Model Checking Timed UML State Machines and Collaborations », in : *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, sous la dir. de Werner DAMM et Ernst-Rüdiger OLDEROG, FTRTFT '02, Berlin, Heidelberg : Springer-Verlag, 2002, p. 395-414, ISBN : 3540441654, DOI : 10.1007/3-540-45739-9_23 (cité pp. 46, 239).

[KW06] Alexander KNAPP et Jochen WUTTKE, « Model Checking of UML 2.0 Interactions », in : *Proceedings of the 2006 International Conference on Models in Software Engineering*, sous la dir. de Thomas KÜHNE, MoDELS'06, Genoa, Italy : Springer-Verlag, 2006, p. 42-51, ISBN : 9783540694885, DOI : 10.1007/978-3-540-69489-2_6 (cité pp. 46, 239).

[KPP08] Dimitrios S. KOLOVOS, Richard F. PAIGE et Fiona A. POLACK, « The Epsilon Transformation Language », in : *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, Zurich, Switzerland : Springer-Verlag, 2008, p. 46-60, ISBN : 9783540699262, DOI : 10.1007/978-3-540-69927-9_4 (cité pp. 46, 233).

[Kri63] Saul A. KRIPKE, « Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi », in : *Mathematical Logic Quarterly* 9.*5-6* (1963), p. 67-96, DOI : 10.1002/malq.19630090502 (cité pp. 39, 90).

[KNP09] Marta KWIATKOWSKA, Gethin NORMAN et David PARKER, « PRISM : Probabilistic Model Checking for Performance and Reliability Analysis », in : *SIGMETRICS Performance Evaluation Review* 36.*4* (mar. 2009), p. 40-45, ISSN : 0163-5999, DOI : 10.1145/1530873.1530882 (cité p. 46).

[LM17] Leslie LAMPORT et Stephan MERZ, *Auxiliary Variables in TLA+*, Research Report, Inria Nancy - Grand Est (Villers-lès-Nancy, France) ; Microsoft Research, mai 2017 (cité p. 35).

[Lan90] Kevin LANO, « Z++, An Object-Orientated Extension to Z », in : *Proceedings of the Fifth Annual Z User Meeting on Z User Workshop*, Berlin, Heidelberg : Springer-Verlag, 1990, p. 151-172, ISBN : 3540196722 (cité p. 43).

[LH00] Kevin LANO et Howard HAUGHTON, *Specification in B : An Introduction Using the B Toolkit*, USA : World Scientific Publishing Co., Inc., 2000, ISBN : 1860940188 (cité p. 43).

[Lan+09] Agnes LANUSSE, Yann TANGUY, Huascar ESPINOZA, Chokri MRAIDHA, Sebastien GERARD, Patrick TESSIER, Remi SCHNEKENBURGER, Hubert DUBOIS et François TERRIER, « Papyrus UML : an open source toolset for MDA », in : *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, juin 2009, p. 1-4 (cité p. 239).

[Lap96] Jean-Claude LAPRIE, *Guide de la sûreté de la fonctionnement*, Cépaduès, 1996, ISBN : 978-2-85428-382-2 (cité p. 32).

[LPY97] Kim G. LARSEN, Paul PETTERSSON et Wang YI, « Uppaal in a Nutshell », in : *International Journal on Software Tools for Technology Transfer* 1.*1–2* (déc. 1997), p. 134-152, ISSN : 1433-2779, DOI : 10.1007/s100090050010 (cité p. 46).

[LMM99] Diego LATELLA, Istvan MAJZIK et Mieke MASSINK, « Towards a Formal Operational Semantics of UML Statechart Diagrams », in : *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, sous la dir. de Paolo CIANCARINI, Alessandro FANTECHI et Robert GORRIERI, Boston, MA : Springer US, 1999, p. 331-347, ISBN : 978-0-387-35562-7, DOI : 10.1007/978-0-387-35562-7_25 (cité pp. 237, 240).

[Lat+15] Florent LATOMBE, Xavier CRÉGUT, Benoit COMBEMALE, Julien DEANTONI et Marc PANTEL, « Weaving Concurrency in Executable Domain-specific Modeling Languages », in : *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, Pittsburgh, PA, USA : ACM, 2015, p. 125-136, ISBN : 9781450336864 (cité p. 28).

[LBG13] Yoann LAURENT, Reda BENDRAOU et Marie-Pierre GERVAIS, « Executing and Debugging UML Models : An fUML Extension », in : *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, Coimbra, Portugal : Association for Computing Machinery, 2013, p. 1095-1102, ISBN : 9781450316569, DOI : 10.1145/2480362.2480569 (cité p. 239).

[LDC18] Manuel LEDUC, Thomas DEGUEULE et Benoit COMBEMALE, « Modular Language Composition for the Masses », in : *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, Boston, MA, USA : Association for Computing Machinery, 2018, p. 47-59, ISBN : 9781450360296, DOI : 10.1145/3276604.3276622 (cité p. 27).

[Led+17] Manuel LEDUC, Thomas DEGUEULE, Benoit COMBEMALE, Tijs van der STORM et Olivier BARAIS, « Revisiting Visitors for Modular Extension of Executable DSMLs », in : *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, MODELS '17, Austin, Texas : IEEE Press, sept. 2017, p. 112-122, ISBN : 9781538634929, DOI : 10.1109/MODELS.2017.23 (cité pp. 27, 135).

[LNH06] Daniel LEROUX, Martin NALLY et Kenneth HUSSEY, « Rational Software Architect : A tool for domain-specific modeling », in : *IBM Systems Journal* 45.*3* (juil. 2006), p. 555-568, ISSN : 0018-8670, DOI : 10.1147/sj.453.0555 (cité pp. 25, 238, 239).

[LDD14] Luka LEROUX, Jérôme DELATOUR et Philippe DHAUSSY, « Modélisation UML d'un régulateur de vitesse automobile », in : *Génie logiciel* 109 (juin 2014) (cité p. 260).

[Ler09] Xavier LEROY, « Formal Verification of a Realistic Compiler », in : *Communications of the ACM* 52.*7* (juil. 2009), p. 107-115, ISSN : 0001-0782, DOI : 10.1145/1538788.1538814 (cité p. 25).

[LB08] Michael LEUSCHEL et Michael BUTLER, « ProB : an automated analysis toolset for the B method », in : *International Journal on Software Tools for Technology Transfer* 10.*2* (mar. 2008), p. 185-203, ISSN : 1433-2779, DOI : 10.1007/s10009-007-0063-9 (cité pp. 43, 46, 96).

[LC02] Philip LEVIS et David CULLER, « Maté : A Tiny Virtual Machine for Sensor Networks », in : *SIGPLAN Not.* 37.*10* (oct. 2002), p. 85-95, ISSN : 0362-1340, DOI : 10.1145/605432.605407 (cité p. 29).

[LP99] Johan LILIUS et Ivan Porres PALTOR, « vUML : a tool for verifying UML models », in : *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, USA : IEEE Computer Society, oct. 1999, p. 255-258, DOI : 10.1109/ASE.1999.802301 (cité pp. 25, 46, 239).

[Lim+09] Didier LIME, Olivier H. ROUX, Charlotte SEIDNER et Louis-Marie TRAONOUEZ, « Romeo : A Parametric Model-Checker for Petri Nets with Stopwatches », in : *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, sous la dir. de Stefan KOWALEWSKI et Anna PHILIPPOU, TACAS '09, York, UK : Springer-Verlag, 2009, p. 54-57, ISBN : 9783642007675, DOI : 10.1007/978-3-642-00768-2_6 (cité p. 46).

[Liu+13a] Shuang LIU, Yang LIU, Etienne ANDRÉ, Christine CHOPPY, Jun SUN, Bimlesh WADHWA et Jin Song DONG, « A Formal Semantics for the Complete Syntax of UML State Machines with Communications », in : *Integrated Formal Methods*, Berlin, Heidelberg : Springer Berlin Heidelberg, jan. 2013, p. 331-346, ISBN : 978-3-642-38613-8 (cité pp. 25, 237, 240).

[Liu+13b] Shuang LIU, Yang LIU, Jun SUN, Manchun ZHENG, Bimlesh WADHWA et Jin Song DONG, « USMMC : A Self-contained Model Checker for UML State Machines », in : *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, Saint Petersburg, Russia : Association for Computing Machinery, 2013, p. 623-626, ISBN : 9781450322379, DOI : 10.1145/2491411.2494595 (cité pp. 48, 240).

[ML12] Tanja MAYERHOFER et Philip LANGER, « Moliz : A Model Execution Framework for UML Models », in : *Proceedings of the 2nd International Master Class on Model-Driven Engineering : Modeling Wizards*, MW '12, Innsbruck, Austria : Association for Computing Machinery, 2012, 3 :1-3 :2, ISBN : 9781450318532, DOI : 10.1145/2448076.2448079 (cité pp. 27, 239).

[MLW12] Tanja MAYERHOFER, Philip LANGER et Manuel WIMMER, « Towards xMOF : Executable DSMLs Based on fUML », in : *Proceedings of the 2012 Workshop on Domain-specific Modeling*, DSM '12, Tucson, Arizona, USA : Association for Computing Machinery, 2012, p. 1-6, ISBN : 9781450316347, DOI : 10.1145/2420918.2420920 (cité p. 28).

[May+13] Tanja MAYERHOFER, Philip LANGER, Manuel WIMMER et Gerti KAPPEL, « xMOF : Executable DSMLs Based on fUML », in : *Software Language Engineering*, sous la dir. de Martin ERWIG, Richard F. PAIGE et Eric VAN WYK, Cham : Springer International Publishing, 2013, p. 56-75, ISBN : 978-3-319-02654-1, DOI : 10.1007/978-3-319-02654-1_4 (cité p. 28).

[McM92] Kenneth Lauchlin MCMILLAN, « Symbolic Model Checking : An Approach to the State Explosion Problem », thèse de doct., Pittsburgh, PA, USA : Carnegie Mellon University, 1992 (cité p. 46).

[Miy+19] Alvaro MIYAZAWA, Pedro RIBEIRO, Wei LI, Ana CAVALCANTI, Jon TIMMIS et Jim WOODCOCK, « RoboChart : modelling and verification of the functional behaviour of robotic applications », in : *Software & Systems Modeling* 18.5 (oct. 2019), p. 3097-3149, ISSN : 1619-1366, DOI : 10.1007/s10270-018-00710-z (cité p. 59).

[Moh15] Mattias MOHLIN, « Modeling Real-Time Applications in RSARTE », in : *IBM, White Paper* (2015) (cité pp. 25, 238).

[Mos06] Peter D. MOSSES, « Formal Semantics of Programming Languages : — An Overview — », in : *Electronic Notes in Theoretical Computer Science* 148.*1* (fév. 2006), Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), p. 41-73, ISSN : 1571-0661, DOI : `10.1016/j.entcs.2005.12.012` (cité pp. 16, 18, 19).

[MB08] Leonardo de MOURA et Nikolaj BJØRNER, « Z3 : An Efficient SMT Solver », in : *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, sous la dir. de C. R. RAMAKRISHNAN et Jakob REHOF, TACAS'08/ETAPS'08, Budapest, Hungary : Springer-Verlag, 2008, p. 337-340, ISBN : 978-3-540-78800-3, DOI : `10.1007/978-3-540-78800-3_24` (cité p. 34).

[Mou+15] Leonardo de MOURA, Soonho KONG, Jeremy AVIGAD, Floris van DOORN et Jakob von RAUMER, « The Lean Theorem Prover (System Description) », in : *Automated Deduction - CADE-25*, sous la dir. d'Amy P. FELTY et Aart MIDDELDORP, Cham : Springer International Publishing, 2015, p. 378-388, ISBN : 978-3-319-21401-6, DOI : `10.1007/978-3-319-21401-6_26` (cité p. 75).

[MG00] Pierre-Alain MULLER et Nathalie GAERTNER, *Modélisation objet avec UML*, Eyrolles, t. 514, Paris, France, 2000 (cité p. 235).

[MO07] John MULLINS et Raveca OARGA, « Model Checking of Extended OCL Constraints on UML Models in SOCLe », in : *Proceedings of the 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, sous la dir. de Marcello M. BONSANGUE et Einar Broch JOHNSEN, FMOODS'07, Paphos, Cyprus : Springer-Verlag, 2007, p. 59-75, ISBN : 9783540729198, DOI : `10.1007/978-3-540-72952-5_4` (cité p. 240).

[OGO04] Iulian OBER, Susanne GRAF et Ileana OBER, « Validation of UML Models via a Mapping to Communicating Extended Timed Automata », in : *Model Checking Software*, sous la dir. de Susanne GRAF et Laurent MOUNIER, Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 127-145, ISBN : 978-3-540-24732-6, DOI : `10.1007/978-3-540-24732-6_9` (cité p. 239).

[OGO06] Iulian OBER, Susanne GRAF et Ileana OBER, « Validating timed UML models by simulation and verification », in : *International Journal on Software Tools for Technology Transfer* 8.*2* (avr. 2006), p. 128-145, ISSN : 1433-2779, DOI : `10.1007/s10009-005-0205-x` (cité pp. 4, 88, 90, 239).

[OC12] Bruno C. d. S. OLIVEIRA et William R. COOK, « Extensibility for the Masses », in : *ECOOP 2012 – Object-Oriented Programming*, sous la dir. de James NOBLE,

Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 2-27, ISBN : 978-3-642-31057-7, DOI : 10.1007/978-3-642-31057-7_2 (cité pp. 27, 135).

[OMG15] OMG, *XML Metadata Interchange*, juin 2015, URL : https://www.omg.org/spec/XMI/2.5.1/PDF (cité p. 17).

[OMG16] OMG, *MOF Query/View/Transformation version 1.3*, juin 2016, URL : https://www.omg.org/spec/QVT/1.3/PDF (cité pp. 46, 233).

[OMG17a] OMG, *Action Language for Foundational UML (Alf)*, juil. 2017, URL : https://www.omg.org/spec/ALF/1.1/PDF (cité p. 160).

[OMG17b] OMG, *Precise Semantics of UML State Machines*, fév. 2017, URL : https://www.omg.org/spec/PSSM/1.0/PDF (cité pp. 136, 160, 237).

[OMG17c] OMG, *Semantics of a Foundational Subset for Executable UML Models*, oct. 2017, URL : https://www.omg.org/spec/FUML/1.4/PDF (cité pp. 27, 160, 237).

[OMG17d] OMG, *Unified Modeling Language*, déc. 2017, URL : https://www.omg.org/spec/UML/2.5.1/PDF (cité pp. 6, 18, 100, 136, 235, 236).

[OMG19] OMG, *Precise Semantics of UML Composite Structures*, juin 2019, URL : https://www.omg.org/spec/PSCS/1.2/PDF (cité pp. 136, 160, 237).

[OMG] OMG, *Modeling and Analysis of Real-time and Embedded systems*, URL : https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf (cité p. 28).

[ORS92] Sam OWRE, John M. RUSHBY et Natarajan SHANKAR, « PVS : A prototype verification system », in : *Proceedings of the 11th International Conference on Automated Deduction : Automated Deduction*, sous la dir. de Deepak KAPUR, CADE-11, Berlin, Heidelberg : Springer-Verlag, 1992, p. 748-752, ISBN : 3540556028, DOI : 10.1007/3-540-55602-8_217 (cité p. 34).

[Par13] Terence PARR, *The definitive ANTLR 4 reference*, The Pragmatic Programmers, 2013, ISBN : 9781934356999 (cité p. 232).

[PR10] Corina S. PĂSĂREANU et Neha RUNGTA, « Symbolic PathFinder : Symbolic Execution of Java Bytecode », in : *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, Antwerp, Belgium : Association for Computing Machinery, 2010, p. 179-180, ISBN : 978-1-4503-0116-9, DOI : 10.1145/1858996.1859035 (cité p. 48).

[Pau94] Lawrence C. PAULSON, *Isabelle a Generic Theorem Prover*, sous la dir. de Springer Science & Business MEDIA, t. 828, Springer-Verlag Berlin Heidelberg, 1994, ISBN : 978-3-540-58244-1, DOI : 10.1007/BFb0030541 (cité p. 34).

[Per+17] Rui PEREIRA, Marco COUTO, Francisco RIBEIRO, Rui RUA, Jácome CUNHA, João Paulo FERNANDES et João SARAIVA, « Energy Efficiency across Programming Languages : How Do Energy, Time, and Memory Relate ? », in : *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, Vancouver, BC, Canada : Association for Computing Machinery, 2017, p. 256-267, ISBN : 9781450355254, DOI : 10.1145/3136014.3136031 (cité p. 159).

[Pha+17] Van Cam PHAM, Ansgar RADERMACHER, Sébastien GÉRARD et Shuai LI, « Complete Code Generation from UML State Machine », in : *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, Porto, Portugal : SCITEPRESS - Science et Technology Publications, Lda, 2017, p. 208-219, ISBN : 9789897582103, DOI : 10.5220/0006274502080219 (cité pp. 25, 179, 238).

[PM03a] Gergely PINTÉR et István MAJZIK, « Automatic Code Generation Based on Formally Analyzed UML Statechart Models », in : *Formal Methods for Railway Operation and Control Systems (Proceedings of the FORMS-2003 Conference)*, sous la dir. de G. TARNAI et E. SCHNIEDER, Budapest, Hungary : L'Harmattan, mai 2003, p. 45-52 (cité p. 25).

[PM03b] Gergely PINTÉR et István MAJZIK, « Program Code Generation based on UML Statechart Models », in : *Periodica Polytechnica Electrical Engineering* 47.*3–4* (2003), p. 187-204 (cité p. 25).

[PL92] Nico PLAT et Peter Gorm LARSEN, « An Overview of the ISO/VDM-SL Standard », in : *SIGPLAN Notices* 27.*8* (août 1992), p. 76-82, ISSN : 0362-1340, DOI : 10.1145/142137.142153 (cité p. 43).

[Plo04] Gordon D. PLOTKIN, « A Structural Approach to Operational Semantics », in : *Journal of Logic and Algebraic Programming 60-61* (2004), p. 7-139 (cité p. 19).

[PD16] Ernesto POSSE et Juergen DINGEL, « An Executable Formal Semantics for UML-RT », in : *Software & Systems Modeling* 15.*1* (fév. 2016), p. 179-217, ISSN : 1619-1366, DOI : 10.1007/s10270-014-0399-z (cité p. 57).

[PTP07] Guillaume POTHIER, Éric TANTER et José PIQUER, « Scalable Omniscient Debugging », in : *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, Montreal, Quebec, Canada : Association for Computing Machinery, 2007, p. 535-552, ISBN : 9781595937865, DOI : 10.1145/1297027.1297067 (cité p. 117).

[QS08] J. P. QUEILLE et J. SIFAKIS, « Specification and Verification of Concurrent Systems in CESAR », in : *25 Years of Model Checking : History, Achievements, Perspectives*, Berlin, Heidelberg : Springer-Verlag, 2008, p. 216-230, ISBN : 9783540698494, DOI : 10.1007/978-3-540-69850-0_13 (cité p. 36).

[Reg10] John REGEHR, *A Guide to Undefined Behaviors in C and C++*, 2010, URL : https://blog.regehr.org/archives/213 (cité p. 161).

[Rev+18] Sébastien REVOL, Géry DELOG, Arnaud CUCCURRU, Jérémie TATIBOUËT, Charles RIVET, Bruno MARQUES et Pauline DEVILLE, *Papyrus : Moka Overview*, 2018, URL : https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution (cité pp. 27, 239).

[RG00] Mark RICHTERS et Martin GOGOLLA, « Validating UML Models and OCL Constraints », in : *Proceedings of the 3rd International Conference on The Unified Modeling Language : Advancing the Standard*, UML'00, York, UK : Springer-Verlag, 2000, p. 265-277, ISBN : 354041133X (cité p. 239).

[Rie+01] Dirk RIEHLE, Steven FRALEIGH, Dirk BUCKA-LASSEN et Nosa OMOROGBE, « The Architecture of a UML Virtual Machine », in : *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, Tampa Bay, FL, USA : Association for Computing Machinery, 2001, p. 327-341, ISBN : 1581133359, DOI : 10.1145/504282.504306 (cité pp. 27, 238).

[RDH03] ROBBY, Matthew B. DWYER et John HATCLIFF, « Bogor : An Extensible and Highly-modular Software Model Checking Framework », in : *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, Helsinki, Finland : Association for Computing Machinery, 2003, p. 267-276, ISBN : 1581137435, DOI : 10.1145/940071.940107 (cité pp. 46, 56).

[Roč+19] Petr ROČKAI, Zuzana BARANOVÁ, Jan MRÁZEK, Katarína KEJSTOVÁ et Jiří BARNAT, « Reproducible Execution of POSIX Programs with DiOS », in : *Software Engineering and Formal Methods*, sous la dir. de Peter Csaba ÖLVECZKY et Gwen SALAÜN, Cham : Springer International Publishing, 2019, p. 333-349, ISBN : 978-3-030-30446-1, DOI : 10.1007/978-3-030-30446-1_18 (cité p. 60).

[RŞ10] Grigore ROŞU et Traian Florin ŞERBĂNUŢĂ, « An Overview of the K Semantic Framework », in : *Journal of Logic and Algebraic Programming* 79.6 (2010), p. 397-434, ISSN : 1567-8326, DOI : 10.1016/j.jlap.2010.03.012 (cité pp. 29, 48).

[Rum+91] James RUMBAUGH, Michael BLAHA, William PREMERLANI, Frederick EDDY et William LORENSEN, *Object-Oriented Modeling and Design*, t. 199, 1, Englewood Cliffs, USA : Prentice-Hall, Inc., 1991, ISBN : 0136298419 (cité p. 235).

[Sad+19] Andrey SADOVYKH, Dragos TRUSCAN, Wasif AFZAL, Hugo BRUNELIERE, Adnan ASHRAF, Abel GÓMEZ, Alexandra ESPINOSA, Gunnar WIDFORSS, Pierluigi PIERINI, Elizabeta FOURNERET et Alessandra BAGNATO, « MegaM@Rt2 Project : Mega-Modelling at Runtime - Intermediate Results and Research Challenges », in : *Software Technology : Methods and Tools*, t. 11771, Lecture Notes in Computer Science (LNCS), Innopolis, Russia : Springer, Cham, oct. 2019, p. 393-405, ISBN : 978-3-030-29852-4, DOI : 10.1007/978-3-030-29852-4_33 (cité p. 46).

[SM05a] Tim SCHATTKOWSKY et Wolfgang MUELLER, « A UML virtual machine for embedded systems », in : *Proceedings of International Conference on Information Systems-New Generations (ISNG)*, USA, avr. 2005 (cité pp. 27, 238).

[SM05b] Tim SCHATTKOWSKY et Wolfgang MULLER, « Transformation of UML State Machines for Direct Execution », in : *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, Washington, DC, USA : IEEE Computer Society, 2005, p. 117-124, ISBN : 0-7695-2443-5, DOI : 10.1109/VLHCC.2005.64 (cité p. 25).

[Shu14] Liu SHUANG, *Formalizing UML State Machines Semantics for Formal Analysis — A survey*, mar. 2014 (cité pp. 237, 240).

[SC05] Doug SIMON et Cristina CIFUENTES, « The Squawk Virtual Machine : Java™ on the Bare Metal », in : *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, San Diego, CA, USA : Association for Computing Machinery, 2005, p. 150-151, ISBN : 1595931937, DOI : 10.1145/1094855.1094908 (cité p. 29).

[SK95] Kenneth SLONNEGER et Barry KURTZ, *Formal Syntax and Semantics of Programming Languages : A Laboratory Based Approach*, 1st, USA : Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN : 0201656973 (cité pp. 16, 18, 19, 20).

[Smi12] Graeme SMITH, *The Object-Z specification language*, t. 1, Springer Science & Business Media, 2012 (cité p. 43).

[SB06] Colin SNOOK et Michael BUTLER, « UML-B : Formal Modeling and Design Aided by UML », in : *ACM Transactions on Software Engineering and Methodology* 15.*1* (jan. 2006), p. 92-122, ISSN : 1049-331X, DOI : 10.1145/1125808.1125811 (cité p. 240).

[Spi92] J. M. SPIVEY, *The Z Notation : A Reference Manual*, GBR : Prentice Hall International (UK) Ltd., 1992, ISBN : 0139785299 (cité p. 43).

[Ste+09] David STEINBERG, Frank BUDINSKY, Marcelo PATERNOSTRO et Ed MERKS, *EMF : Eclipse Modeling Framework 2.0*, 2nd, Addison-Wesley Professional, 2009, ISBN : 0321331885 (cité pp. 129, 232, 234).

[TDL17] Ciprian TEODOROV, Philippe DHAUSSY et Luka LE ROUX, « Environment-Driven Reachability for Timed Systems », in : *International Journal on Software Tools for Technology Transfer* 19.*2* (avr. 2017), p. 229-245, ISSN : 1433-2779, DOI : `10.1007/s10009-015-0401-2` (cité p. 45).

[Teo+16] Ciprian TEODOROV, Luka LE ROUX, Zoé DREY et Philippe DHAUSSY, « Past-Free[ze] reachability analysis : reaching further with DAG-directed exhaustive state-space analysis », in : *Software Testing, Verification and Reliability* 26.*7* (nov. 2016), p. 516-542, ISSN : 0960-0833, DOI : `10.1002/stvr.1611` (cité pp. 40, 45, 195).

[Thi+07] Xavier THIRIOUX, Benoît COMBEMALE, Xavier CRÉGUT et Pierre-Loïc GAROCHE, « A Framework to Formalise the MDE Foundations », in : *International Workshop on Towers of Models (TOWERS 2007)*, Zurich, Switzerland : University of York, juin 2007, p. 14-30 (cité p. 230).

[Tka08] Oksana TKACHUK, « Domain-specific environment generation for modular software model checking », thèse de doct., Kansas State University, 2008 (cité pp. 37, 40, 109, 194).

[Tor+19] Carmen TORRES LOPEZ, Robbert GURDEEP SINGH, Stefan MARR, Elisa GONZALEZ BOIX et Christophe SCHOLLIERS, « Multiverse Debugging : Non-deterministic Debugging for Non-deterministic Programs », in : *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, sous la dir. d'Alastair F. DONALDSON, t. 134, Leibniz International Proceedings in Informatics (LIPIcs) 27, Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, p. 1-30, ISBN : 978-3-95977-111-5, DOI : `10.4230/LIPIcs.ECOOP.2019.27` (cité pp. 33, 48, 83).

[Var+15] Matias Ezequiel VARA LARSEN, Julien DEANTONI, Benoit COMBEMALE et Frédéric MALLET, « A Behavioral Coordination Operator Language (BCOoL) », in : *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems*, sous la dir. de Timothy LETHBRIDGE, Jordi CABOT et Alexander EGYED, MODELS '15 18, Ottawa, Canada : IEEE Press, sept. 2015, p. 186-195, ISBN : 9781467369084 (cité p. 114).

[Voe+12] Markus VOELTER, Daniel RATIU, Bernhard SCHAETZ et Bernd KOLB, « Mbeddr : An Extensible C-based Programming Language and IDE for Embedded Systems », in : *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications : Software for Humanity*, SPLASH '12, Tucson, Arizona, USA : Association for Computing Machinery, 2012, p. 121-140, ISBN : 9781450315630, DOI : 10.1145/2384716.2384767 (cité pp. 46, 58).

[WO16] Yanlin WANG et Bruno C. d. S. OLIVEIRA, « The Expression Problem, Trivially ! », in : *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, Málaga, Spain : Association for Computing Machinery, 2016, p. 37-41, ISBN : 9781450339957, DOI : 10.1145/2889443.2889448 (cité pp. 27, 135).

[Wir96] Niklaus WIRTH, « Extended Backus-Naur Form (EBNF) », in : *Iso/Iec* 14977.*2996* (1996) (cité p. 232).

[WD96] Jim WOODCOCK et Jim DAVIES, *Using Z : Specification, Refinement, and Proof*, USA : Prentice-Hall, Inc., 1996, ISBN : 0139484728 (cité p. 43).

[XRK99] Spyros XANTHAKIS, Pascal REGNIER et Constantin KARAPOULIOS, *Le test des logiciels*, Hermes Science Publications, 1999, ISBN : 978-2-7462-0083-8 (cité p. 33).

[YML99] Yuan YU, Panagiotis MANOLIOS et Leslie LAMPORT, « Model Checking TLA+ Specifications », in : *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, sous la dir. de Laurence PIERRE et Thomas KROPF, CHARME '99, Berlin, Heidelberg : Springer-Verlag, 1999, p. 54-66, ISBN : 3540665595, DOI : 10.1007/3-540-48153-2_6 (cité p. 46).

[Zel05] Andreas ZELLER, « Why Program Fail », in : 1st, Elsevier, oct. 2005, ISBN : 9781558608665 (cité p. 33).

[Zsc18] Steffen ZSCHALER, « Adding a HenshinEngine to GEMOC Studio : An experience report », in : *6th International Workshop on The Globalization of Modeling Languages (GEMOC'18)*, Copenhagen, Danmark, août 2018 (cité p. 28).

# Annexes

ANNEXE A

# LE LANGAGE UML ET SES FONDEMENTS EN IDM

---

## A.1 Introduction

Dans le cadre de cette thèse, l'Ingénierie Dirigée par les Modèles (IDM) est utilisée comme moyen d'abstraction afin de définir des modèles. Cette branche du génie logiciel organise les différentes activités de développement d'un logiciel autour d'un modèle ou d'un ensemble de modèles. Elle offre un cadre méthodologique et un ensemble d'outils permettant notamment la modélisation, l'exécution et l'analyse de modèles.

Le langage UML est l'un des principaux langages de modélisation définis par la communauté de l'IDM. Il s'agit d'un langage généraliste principalement utilisé pour la modélisation de systèmes logiciels. Ce langage unifié offre un support de communication commun entre tous les ingénieurs intervenant dans le développement d'un même système. Il permet également de visualiser le système sous différents points de vue à l'aide de multiples diagrammes. Grâce à ces caractéristiques, le langage UML s'est imposé comme standard *de facto* dans le domaine industriel. C'est notamment pour cette raison qu'UML a été choisi comme exemple d'application principal des travaux de cette thèse.

Dans cette annexe, la section A.2 présente les grands principes de l'IDM puis la section A.3 expose les caractéristiques du langage UML.

## A.2 L'ingénierie dirigée par les modèles

L'Ingénierie Dirigée par les Modèles (IDM) (ou en anglais Model Driven Engineering *(MDE)*) est un courant scientifique visant à orienter les pratiques d'ingénierie autour du concept de *modèle*. Dans l'ingénierie du logiciel, cette méthodologie basée sur le paradigme du "tout est modèle" [Béz04] fait suite au paradigme du "tout est objet" de la programmation orientée objet des années 80. Cette notion de modèle a ainsi fait apparaître de nouvelles possibilités (p. ex. les transformations de modèles) mais a aussi profondément modifié les activités du génie logiciel (p. ex. la conception, l'exécution, la vérification). Les modèles ne sont plus simplement

utilisés à des fins documentaires ou illustratives mais pour diverses activités tout au long du cycle de vie du logiciel. L'IDM permet ainsi d'exploiter tout le potentiel des modèles.

### A.2.1 Principes de l'IDM

L'IDM repose sur le concept de modèle comme entité de première classe. À ce jour, il n'existe pas de définition universelle pour définir un modèle du point de vue de l'IDM. Plusieurs travaux [JB06 ; Thi+07] ont néanmoins proposé une formalisation des grands principes de l'IDM et ont ainsi permis de mieux caractériser la notion de modèle. En se basant sur les définitions données dans [Com08 ; Jou06], cette thèse considère qu'un modèle peut être défini de la façon suivante :

**Définition A.1 (Modèle).** Un modèle est une abstraction d'un système qui est créée dans une intention particulière afin d'obtenir une représentation de ce système pour le but énoncé.

#### A.2.1.1 Les relations clés de l'IDM

Tout comme les langages orientés objet avec les relations d'héritage ("hérite de") et d'instanciation ("instance de"), l'IDM est également fondée sur deux relations essentielles [Béz04].

La première est appelée **relation de représentation** ("représentation de"). Le modèle est une représentation du système qu'il modélise. La communauté de l'IDM ne fournit pas de critères permettant de définir précisément ce qu'est un "bon" modèle ou une bonne représentation d'un système. Cependant, l'abstraction réalisée doit être pertinente par rapport au besoin énoncé afin que le modèle puisse se substituer au système réel dans un cadre précis [Com08]. Un "bon" modèle doit donc satisfaire le principe de substituabilité défini comme suit :

**Définition A.2 (Principe de substituabilité).** "*Un modèle doit être suffisant et nécessaire pour permettre de répondre à certaines questions en lieu et place du système qu'il est censé représenter, exactement de la même façon que le système aurait répondu lui-même.*" [Com08]

La seconde relation est appelée **relation de conformité** ("conforme à"). Un modèle doit en effet être conforme au métamodèle c.-à-d. à l'ensemble des concepts et relations qui permettent de le définir. Le métamodèle représente les concepts du langage de modélisation ainsi que les relations entre ces concepts (c.-à-d. la syntaxe abstraite). Il ne permet pas pour autant de définir la signification (c.-à-d. la sémantique) ni la représentation visuelle (c.-à-d. la syntaxe concrète) des concepts de ce langage. En toute rigueur, un modèle est dit conforme à son métamodèle mais, par abus de langage, il est aussi commun de dire qu'un modèle est conforme à son langage de modélisation [Bou15].

En IDM, un modèle a donc deux facettes. Il est à la fois une représentation plus ou moins abstraite d'un système mais aussi un "programme" conforme à un langage. La Figure A.1

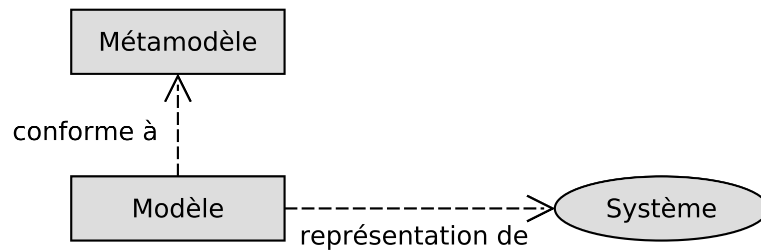



FIGURE A.1 – Les deux relations de base en IDM.

illustre ces deux relations par rapport au système à modéliser et au métamodèle du langage de modélisation. Ces deux relations sont de natures différentes mais leur complémentarité permet de faire le lien entre un langage de modélisation et les caractéristiques des systèmes pouvant être modélisés par des modèles conformes à ce langage [Jou06].

#### A.2.1.2 L'approche MDA

L'initiative *Model Driven Architecture* (MDA®) lancée en 2000 par l'*Object Management Group* (OMG®) présente un ensemble de pratiques et d'outils permettant la mise en œuvre de l'IDM. La création du MDA a été déclenchée suite à l'émergence de nombreux langages de modélisation dans divers domaines (p. ex. la modélisation objet, les procédés de développement, les entrepôts de données, la gestion des flux) [Béz04]. Pour éviter des incompatibilités entre les métamodèles de ces différents langages, il était nécessaire d'édifier un cadre commun pour les définir. L'OMG a apporté une réponse à ce problème avec le standard *Meta Object Facility* (MOF™) qui est au cœur du MDA. MOF est un métamétamodèle décrivant les concepts d'un langage permettant de définir des langages de modélisation.

L'architecture en 4 couches du MDA est présentée sur la Figure A.2. Les modèles sont définis dans la couche *M1*. Les concepts et les relations permettant de les construire sont définis, au niveau *M2*, par les métamodèles de ces langages (p. ex. UML pour les logiciels, *Software & Systems Process Engineering Metamodel* (SPEM™) pour les procédés de développement, SysML pour les systèmes). Ces métamodèles sont eux-mêmes définis à partir de concepts et de relations décrits dans le métamétamodèle de la couche *M3*. Dans le cadre du MDA, le métamétamodèle est appelé MOF. Il définit un langage de métamodélisation c.-à-d. un langage permettant de décrire des langages de modélisation. Pour d'éviter d'avoir un nombre infini de couches, le métamétamodèle est conforme à lui-même. Ainsi, le langage MOF peut être décrit en termes de ses propres concepts. C'est le principe de méta-circularité. La *couche M0* représente quant à elle le "monde réel" c.-à-d. l'exécution des systèmes. Contrairement aux couches supérieures, elle n'est donc pas considérée comme un niveau de modélisation. Pour cette rai-

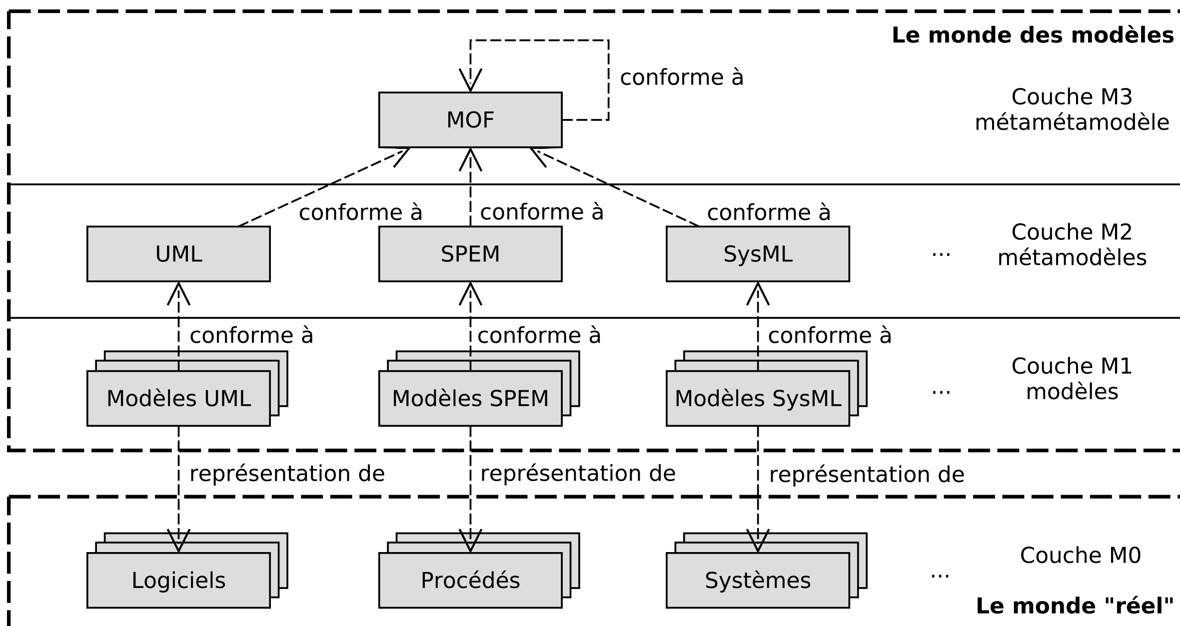



FIGURE A.2 – L'architecture en 4 couches du MDA.

son, l'architecture du MDA est considérée, plus rigoureusement, comme une architecture en 3+1 couches [Béz04]. Cette architecture est notamment mise en pratique dans EMF [Ste+09] avec le langage Ecore comme implémentation de MOF.

Cette hiérarchie en couches des modèles n'est pas propre au MDA ni à l'IDM. D'autres domaines de l'informatique utilisent cette même hiérarchie mais avec un métamétamodèle différent et d'autres outils de développement. Chaque hiérarchie en couches définit un espace technique. Par exemple, l'espace technique des grammaires se base sur des langages tels que *Extended Backus-Naur Form* (EBNF) [Wir96] ou *ANother Tool for Language Recognition* (ANTLR) [Par13] alors que celui des documents *eXtensible Markup Language* (XML™) utilise les *XML Schemas* ou les *Document Type Definitions* (DTDs). D'après [Jou06], un espace technique est défini comme suit :

**Définition A.3 (Espace technique).** Un espace technique est "*un système de représentation de modèles ainsi qu'un ensemble de solutions techniques pour les manipuler.*" [Jou06]

Afin de décrire le cycle de développement d'un logiciel, le MDA recommande l'emploi de trois types de modèles [Com08 ; GMB09] :

— *Computational Independent Model* (CIM) : Les modèles représentant les exigences logicielles indépendamment de toutes considérations technologiques ;
— *Plateform Independent Model* (PIM) : Les modèles servant à concevoir et analyser le système en prenant en compte les choix technologiques mais de façon indépendante

de la cible d'exécution ;
— *Plateform Specific Model* (PSM) : Les modèles correspondant à l'implémentation du modèle pour une plateforme d'exécution donnée.

L'utilisation de ces différents modèles permet à partir des exigences de produire du code exécutable en passant par des modèles à différents niveaux d'abstraction. À chaque étape, le modèle est raffiné pour prendre en compte de nouveaux détails du système ou certains choix technologiques. Le passage d'un niveau d'abstraction à un autre est réalisé par une transformation de modèles. Pour le déploiement sur une plateforme d'exécution, le MDA préconise l'emploi d'un cycle de développement en Y permettant d'une part de prendre en compte les exigences métiers (c.-à-d. les contraintes et besoins du domaine) et d'autre part les exigences de la cible d'exécution. En effet, à partir du PIM et d'un modèle de description de la plateforme d'exécution appelé *Plateform Description Model* (PDM), il est possible d'obtenir le PSM. Avec le MDA, il est ainsi possible de raisonner sur le système étudié à différents niveaux d'abstraction.

#### A.2.1.3 Transformation de modèles

La transformation de modèles est une des techniques clés de l'IDM. Elle repose sur un ensemble de règles (déclaratives, impératives ou hybrides) permettant de traduire un modèle source en un modèle cible [Jou06]. Cette technique est illustrée sur la Figure A.3. Les règles de transformations s'appuient sur les concepts du *Métamodèle source* et ceux du *Métamodèle cible* pour décrire la transformation et ainsi établir une correspondance entre les concepts de ces deux métamodèles. Puisqu'en IDM tout est modèle, ces règles sont capturées par un *Modèle de transformation* conforme au métamodèle du langage de transformation (*Métamodèle de transformation*). Pour décrire des transformations, plusieurs langages ont été conçus dont ATL (*ATLAS Transformation Language*) [JK06], ETL (*EPSILON Transformation Language*) [KPP08] ou encore QVT (*Query/View/Transformation*) [OMG16], le standard de l'OMG. À l'aide du modèle de transformation, le *Moteur de transformation* peut ensuite appliquer les règles de transformation sur un *Modèle source* afin de produire un *Modèle cible*.

On distingue deux types de transformation de modèles : les transformations exogènes si les métamodèles source et cible sont différents, et les transformations endogènes s'ils sont identiques. Dans ce dernier cas, la transformation peut être réalisée en place (aussi appelé mode *refining*) afin d'appliquer les changements directement sur le modèle d'entrée. Pour l'exécution de modèles, les générateurs de code implémentent typiquement des transformations exogènes qui entraînent l'apparition de fossés sémantique. Ce sont ces transformations exogènes non-prouvées que l'approche EMI cherche à tout prix à éviter. Les interpréteurs implémentent quant à eux des transformations endogènes.

Quel que soit le type de transformation utilisé, cet outil est essentiel en IDM pour la mise en œuvre de nombreuses activités de développement logiciel (p. ex. la génération de code).

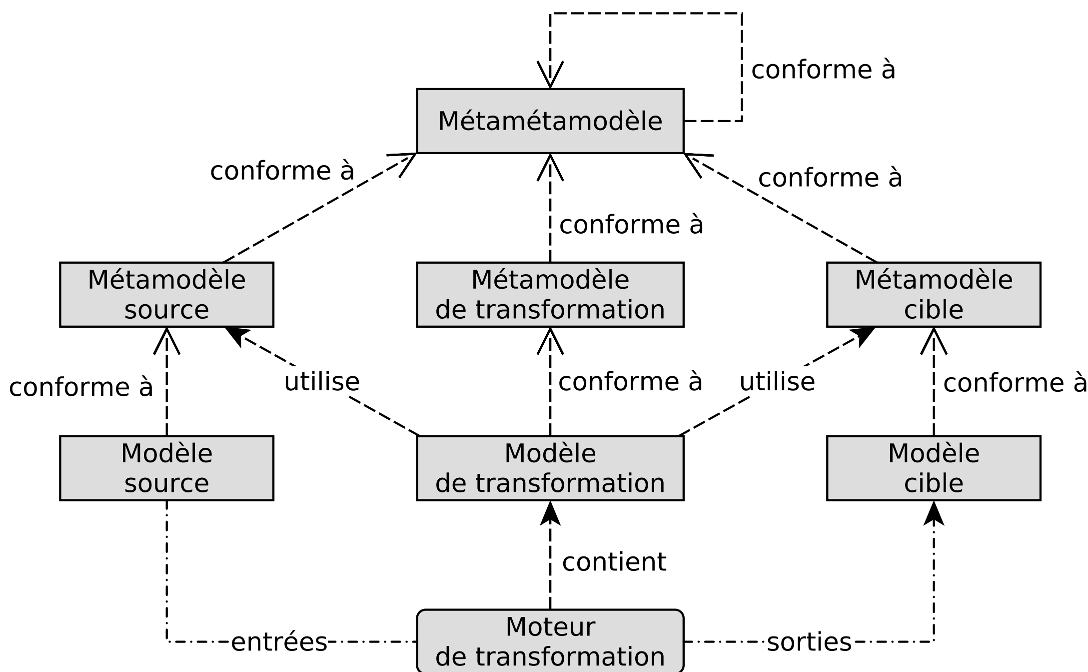



FIGURE A.3 – Principe d'une transformation de modèles en IDM.

### A.2.2 Les langages en IDM

L'IDM est une forme particulière de l'ingénierie des langages dans le sens où elle permet de définir des langages et de concevoir des modèles conformes à ces langages. L'IDM étant centrée sur la notion de modèles, ces langages sont donc des langages de modélisation. Comme tous les autres langages (cf. section 1.2), ils possèdent à la fois une syntaxe abstraite, des syntaxes concrètes, et une sémantique. Le principal point de différenciation concerne la syntaxe abstraite qui en IDM est décrite à l'aide d'un métamodèle. Différents EDI (p. ex. EMF [Ste+09], MPS [1], Spoofax [2] [KV10]) permettent de définir des métamodèles et d'offrir des services permettant de manipuler des langages de modélisation.

En opposition avec les langages de programmation génériques (GPLs), l'IDM a contribué à l'émergence de langages dédiés à un domaine (DSLs). Ces langages s'appuient directement sur les concepts métiers d'un domaine pour prendre en compte les spécificités (p. ex. le vocabulaire, les besoins, les contraintes) de ce domaine particulier. De même pour les langages de modélisation, on distingue les langages de modélisation dédiés à un domaine (DSMLs) et les langages de modélisation génériques (*General Purpose Modeling Language*s (GPMLs)). La frontière entre ces deux catégories reste parfois floue et dépend du point de vue adopté.

1. Meta Programming System (MPS) : `https://www.jetbrains.com/mps/`.
2. Spoofax : `http://www.metaborg.org/en/latest/index.html`.

Par exemple, le langage UML est parfois qualifié de GPML à cause de la richesse et de la variété des concepts de son métamodèle. Cependant, il reste particulièrement adapté pour la modélisation de logiciels et peut être vu dans ce sens comme un DSML. Dans cette thèse, les termes langage de modélisation et DSML seront utilisés de façon interchangeable. De plus, si un moteur d'exécution a été défini pour un DSML donné, alors ce langage est exécutable. On le qualifie alors de DSML eXécutable (xDSML).

La section suivante présente plus en détail le langage UML qui constitue un exemple pertinent pour décrire comment les principes de l'IDM ont été appliqués sur un langage de modélisation.

## A.3 Le langage UML

Le langage UML est l'un des langages de modélisation les plus utilisés pour la conception de logiciels. À ce titre, il a été choisi comme cas d'application principal pour illustrer la mise en œuvre de la solution conçue dans cette thèse. La section A.3.1 commence par un point de vue sur l'historique du langage avant d'évoquer le standard UML dans la section A.3.2. Enfin, les sections A.3.3 et A.3.4 présentent respectivement les différents outils permettant l'exécution et l'analyse des modèles UML.

### A.3.1 Historique

Pour faire face à la prolifération des méthodes objets au début des années 90, le langage *Unified Modeling Language* (UML) [OMG17d] a été créé afin d'unifier les différentes pratiques de développement logiciel. Le processus de réflexion autour de ce langage unificateur a débuté en 1995 avant d'être standardisé en 1997 par l'OMG dans sa version 1.1. Ce langage est principalement basé sur les concepts de trois méthodes objets : Booch [Boo86], OMT [Rum+91], et OOSE [Jac+92] mais il a également subi les influences d'autres méthodes de développement (p. ex. les automates de Harel [HK04]). UML est un langage qui offre une notation unifiée et non pas une méthode de développement [MG00]. L'objectif de ce langage est de pouvoir être utilisé avec n'importe quelle méthode (p. ex. le cycle en V, les méthodes agiles) et pour toutes les phases du cycle de développement d 'un système. Aujourd'hui, le langage UML s'est imposé comme standard *de facto* pour la conception logicielle au sein du monde industriel.

### A.3.2 Le standard UML

La dernière version du standard UML est la version 2.5.1 [OMG17d]. Il permet de modéliser à la fois *(i)* la partie structurelle et *(ii)*) la partie comportementale des logiciels mais il fournit également *(iii)* certains concepts de modélisation complémentaires. Ces trois domaines de la

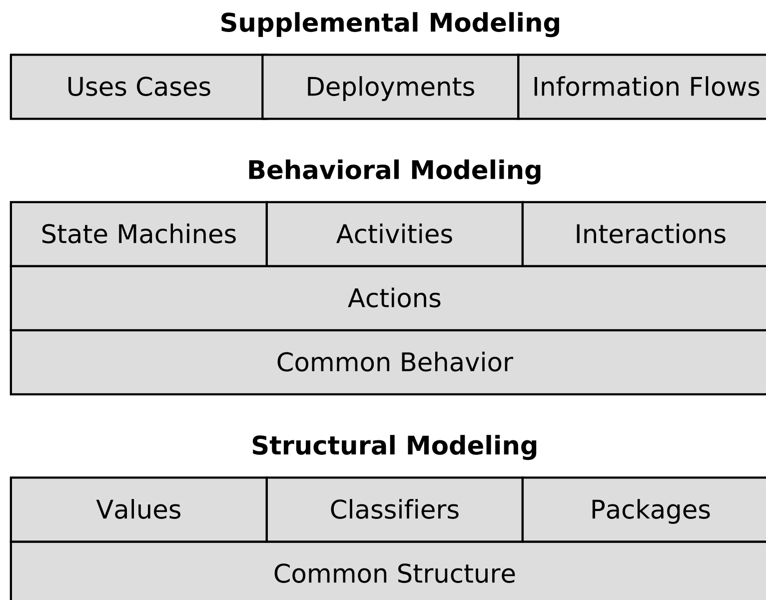



FIGURE A.4 – Vue d'ensemble des concepts d'UML.

sémantique UML sont présentés sur la Figure A.4 issue de [OMG17d]. La partie structurelle (*Structural Modeling*) repose sur un certain nombre de concepts fondamentaux comme les types ou les espaces de nommage (*namespaces*) définis dans la *Common Structure*. Trois catégories d'éléments s'appuient sur cette base commune : *Value* pour la définition des littéraux et des expressions, *Classifiers* pour la définition des types de données, des classes, des signaux, des interfaces et des composants, et enfin *Packages* pour la définition des packages et des profils. Les profils offrent la possibilité d'étendre le langage UML avec de nouveaux concepts. La partie comportementale (*Behavioral Modeling*) se base également sur les concepts fondamentaux de *Common Behavior* pour permettre l'exécution des comportements et la communication par évènements. Les *Actions* permettent ensuite de décrire finement le comportement du système. C'est ce que l'on appelle communément le langage d'action. À l'aide de ces actions, il est ensuite possible de définir des machines à états (*State Machines*), des activités (*Activities*) ou des interactions (*Interactions*). Une troisième partie (*Supplemental Modeling*) fournit des concepts de modélisation supplémentaires afin de décrire les cas d'utilisation du système (*Uses cases*), le déploiement sur des plateformes d'exécution (*Deployments*) et les échanges d'information entre les entités du système (*Information Flows*).

Grâce à tous ces concepts, UML permet la conception de systèmes logiciels sous forme de modèles. Afin d'offrir différents points de vue sur ces modèles, plusieurs types de diagrammes ont été définis dans UML dont :

— Les diagrammes de classes qui représentent les classes (c.-à-d. les types des objets)

et leurs relations ;
— Les diagrammes de structures composites qui décrivent la structure des composants logiciels du système ;
— Les diagrammes d'états-transitions qui offrent une représentation des machines à états ;
— Les diagrammes de séquence qui donnent une vue temporelle des interactions entre les objets.

Chacun de ces diagrammes représente donc une vue du système par rapport à une intention donnée. Leur utilisation permet de faciliter la conception des logiciels et de les documenter. Cette notation unifiée offre ainsi un langage commun pour tous les ingénieurs et les développeurs facilitant ainsi leurs interactions.

Le standard UML a aussi pour objectif de préciser la sémantique des différents concepts de modélisation (Figure A.4). Cette sémantique est décrite dans le standard de l'OMG mais uniquement via un langage naturel. UML n'est donc qu'un langage semi-formel car la signification de certains concepts reste floue voir indéterminée. Ce langage possède par exemple de nombreux points de variation sémantique c.-à-d. que différentes interprétations ou implémentations d'un même concept peuvent être considérées comme correctes.

Pour permettre l'exécution des modèles UML, il est cependant nécessaire de définir une sémantique précise. À ce titre, plusieurs travaux [Liu+13a ; Shu14 ; Fra+98 ; LMM99] ont proposé une formalisation d'UML afin de lever toute ambiguïté sur sa sémantique. Afin de préserver l'expressivité d'UML, l'OMG a préféré définir de nouveaux standards permettant de préciser la sémantique d'un sous-ensemble d'UML :

— fUML [OMG17c] définit le sous-ensemble de base permettant l'exécution de modèles UML en se basant sur les activités ;
— PSCS [OMG19] précise la signification des structures composites notamment pour permettre l'instanciation des objets du modèle à partir de ces structures composites ;
— PSSM [OMG17b] précise la sémantique des machines à états pour la spécification du comportement des objets.

Dans la même lignée, d'autres standards devraient voir le jour pour préciser d'autres aspects d'UML comme la prise en compte du temps via *Precise Semantics of Time* (PSOT). Ces nouveaux standards vont ainsi permettre d'accroître le sous-ensemble exécutable d'UML.

### A.3.3 Techniques d'exécution de modèles UML

Le langage UML étant l'un des langages de modélisation les plus populaires, de nombreux outils ont été développés afin de pouvoir exécuter des modèles conformes à ce langage. La revue systématique réalisée en [CMS18] dresse une liste des principaux outils d'exécution de modèles UML développés dans les milieux académique et industriel. Cette section présente

quelques-uns de ces outils capturant la sémantique d'UML sous forme translationnelle ou opérationnelle.

**Avec une sémantique translationnelle.** Les EDI Papyrus Software Designer [Pha+17], Rational Software Architect [LNH06] et Rhapsody [HK04] proposent des outils de modélisation permettant de concevoir des modèles UML puis de générer du code (p. ex. C, C++, Java) à partir de ces modèles. Ils proposent également des outils de simulation, de débogage, et d'animation afin de faciliter le développement de ces modèles. En suivant la même philosophie, les outils Papyrus-RT [HDB17], Rational Software Architect Real Time Edition [Moh15], et Fujaba [Bur+05] permettent d'exécuter des modèles de systèmes temps réels en s'appuyant sur le langage UML-RT, une variante d'UML pour le temps réel. Une autre variante d'UML appelée xtUML (*eXecutable Translatable UML*) s'appuie sur une sémantique translationnelle pour exécuter des modèles avec l'outil BridgePoint [3]. D'autres outils comme UML4CPP [Jäg+16], le simulateur de modèles en [KDH07] ou Fujaba [Bur+05] permettent également d'exécuter et de simuler des modèles UML en se basant sur des techniques de génération de code. Certains de ces générateurs de code (p. ex. Rhapsody, UML4CPP) suivent même les recommandations de la norme MISRA qui spécifie des règles de programmation permettant d'améliorer la sûreté de fonctionnement et la fiabilité des logiciels programmés en C et/ou C++. Les compilateurs de modèles GUML [CMB12] et Unicomp [Cic18] permettent également de générer du code exécutable à partir de modèles UML. GUML définit un front-end pour GCC [4] permettant de générer du code machine optimisé pour l'exécution de machines à états UML. Unicomp permet quant à lui de générer du code LLVM [5] pour les modèles fUML.

**Avec une sémantique opérationnelle.** Le projet en [Rie+01] présente l'architecture d'une machine virtuelle UML et sa mise en œuvre au sein d'un prototype. Avec cette architecture, une modification du modèle est immédiatement mise en œuvre sur le moteur d'exécution permettant ainsi un retour rapide à l'utilisateur. L'interprétation de modèles peut se baser sur les machines à états UML comme pour UVM [SM05a] qui vise l'exécution de systèmes embarqués ou pour l'interpréteur en [Cha+09] qui vise l'exécution de modèles UML auto-adaptatifs. Des interpréteurs de modèles ont également été mis en œuvre pour exécuter les activités et les actions de UML 2.0. ACTi [CD08] est une machine virtuelle UML offrant diverses capacités d'analyse dont la détection de *deadlocks* ou la vérification d'assertions. Pópulo [FMS08] est un interpréteur et un débogueur de modèles UML pouvant facilement être étendu pour personnaliser le langage d'action. L'interprétation de modèles a aussi été choisie pour réaliser

3. xtUML avec BridgePoint : `https://xtuml.org/`.
4. GCC : `https://gcc.gnu.org/`.
5. LLVM : `https://llvm.org/`.

l'implémentation de référence [6] du standard fUML. Les outils Moka [Rev+18] et Moliz [ML12] permettent également d'interpréter des modèles fUML afin d'appliquer des techniques de V&V dynamiques comme de la simulation, du débogage et du test. Ces deux interpréteurs peuvent être utilisés avec des modèles fUML créés avec Papyrus [Lan+09]. De façon similaire, l'approche en [LBG13] propose d'étendre l'implémentation de référence de fUML afin que l'exécution puisse être pilotée et observée pour réaliser du débogage de modèles.

### A.3.4 Techniques d'analyse de modèles UML

Pour analyser des modèles UML, différentes techniques et outils ont été développés. Pour le *model-checking*, deux approches sont principalement utilisées : la première consiste à transformer les modèles UML vers des langages formels (cf. section 2.5.2) alors que la seconde vise à formaliser UML pour développer des *model-checkers* dédiés (cf. section 2.5.3). Il est également possible d'utiliser les approches par raffinement en utilisant le profil UML-B.

**Outils d'analyse non formels.** En dehors du cadre de la vérification formelle, un certain nombre d'outils permettent l'analyse de modèles UML. Ces modèles peuvent notamment être validés d'un point de vue structurel à l'aide de contraintes *Object Constraint Language* (OCL™) comme dans [RG00]. Les outils Rhapsody [HK04] et Rational Software Architect [LNH06] permettent d'aller plus loin en offrant la possibilité de simuler et de déboguer ces modèles. Les interpréteurs Moka [Rev+18] et Moliz [ML12] permettent également de mettre en œuvre ces activités d'analyse pour des modèles fUML. La revue systématique en [CMS18] montre que l'activité de V&V la plus pratiquée sur les modèles UML est la simulation alors que l'utilisation des méthodes formelles reste assez marginale. Néanmoins, la méthode de vérification formelle la plus utilisée est le *model-checking*.

**Model-checking via la transformation vers un langage formel.** Les outils Hugo/RT [KW06 ; KMR02] et vUML [LP99] transforment des modèles UML en PROMELA afin de pouvoir les vérifier avec le *model-checker* Spin [Hol97]. Avec Hugo/RT, des diagrammes UML d'interactions (p. ex. des diagrammes de séquence ou de collaboration) sont transformés en automates observateurs pour vérifier que les interactions entre les machines à états du modèle UML sont conformes aux scénarios décrits par les diagrammes d'interaction. vUML permet, quant à lui, de vérifier des propriétés de sûreté et de détecter des *deadlocks* et des *livelocks*. Si un contre-exemple est trouvé, l'outil produit un diagramme de séquence UML indiquant comment reproduire l'erreur. D'autres langages formels peuvent également être utilisés pour la vérification de modèles UML. Dans [OGO04 ; OGO06], une correspondance entre les modèles UML et les

6. fUML Ref. Impl. : `http://modeldriven.github.io/fUML-Reference-Implementation/`.

automates temporisés du formalisme IF a été définie. Les modèles IF obtenus peuvent ensuite être simulés ou vérifiés en exprimant des propriétés de sûreté sous forme d'automates observateurs en UML. Le projet en [Jou+14] montre comment transformer des modèles UML en Fiacre afin de pouvoir appliquer le *model-checker* OBP1. De façon similaire, dans [MO07], des modèles UML sont transformés dans un formalisme spécifique de systèmes de transitions en passant par le langage intermédiaire *Abstract State Machine* (ASM). Des propriétés formelles peuvent ensuite être exprimées en *Extended Object Constraint Language* (EOCL) qui étend le langage OCL avec des opérateurs temporels.

**Model-checking via l'utilisation d'outils dédiés.** Pour construire des outils de vérification formelle dédiés au langage UML, une étape indispensable est de définir plus précisément la sémantique du langage. Pour ce faire, plusieurs travaux dont [Fra+98 ; LMM99] ont réussi à formaliser un sous-ensemble du langage. Plus récemment, une sémantique formelle pour le sous-ensemble des machines à états UML a été définie dans [Liu+13a ; Shu14]. Ce travail a ensuite permis de développer le *model-checker* USMMC [Liu+13b] dédié à la vérification de modèles UML. De façon comparable, le *model-checker* UMC [GM04 ; GM05] permet la vérification de modèles UML à l'aide de propriétés exprimées dans divers formalismes basés sur la logique temporelle et le $\mu$-*calculus*.

**Vérification formelle par raffinement.** En ce qui concerne les approches par raffinement, un profil UML appelé UML-B a été défini [SB06]. Ce profil permet d'un côté de profiter des fondements mathématiques de B pour préciser la sémantique d'UML et de l'autre de s'affranchir de la complexité de ces concepts pour la modélisation. Le modèle obtenu est ensuite traduit en B afin de prouver formellement les raffinements réalisés.

## A.4 Synthèse

L'IDM définit un cadre méthodologique et des outils pour modéliser, exécuter et analyser des modèles plutôt que des programmes. Les modèles permettent de se placer à un plus haut niveau d'abstraction afin de se concentrer davantage sur la conception et le fonctionnement du système plutôt que sur son implémentation. L'IDM peut donc être utilisée comme moyen d'abstraction pour concevoir des systèmes logiciels sous forme de modèles. Cette ingénierie permet également de concevoir des langages et de définir des outils d'exécution capturant leurs sémantiques.

Le langage UML s'inscrit dans cette démarche d'ingénierie des modèles puisque sa syntaxe abstraite est définie sous forme de métamodèle. De nombreux outils industriels et académiques ont déjà été développés pour pouvoir exécuter des modèles UML principalement sous la forme

d'interpréteurs ou de générateurs de code. En ce qui concerne les activités de V&V, l'analyse de modèles UML est majoritairement réalisée via des outils de simulation ou de débogage. L'application de méthodes formelles comme le *model-checking* reste complexe car le langage UML n'est que semi-formel.

ANNEXE B

# FORMALISATION EN LEAN

Cette annexe présente le code Lean correspondant à toutes définitions formelles présentées dans cette thèse. Ce code peut être utilisé avec la version web de l'éditeur Lean v3 disponible à l'adresse suivante : `https://leanprover.github.io/live/latest/`.

```
namespace emi

variables
    (C : Type) -- configurations
    (A : Type) -- actions
    (L : Type) -- atomic propositions
    (S : Type) -- filtering policy execution states
    (Vc : Type) -- view of a configuration
    (Va : Type) -- view of an action

--------------------------------------------------------------------------------
--                        The execution control interface                     --
--------------------------------------------------------------------------------

-- STR structure (STR: Semantic Transition Relation)
structure STR (C A : Type) :=
    (initial : set C)
    (actions : C → set A)
    (execute : C → A → set C)

-- APE structure (APE: Atomic Proposition Evaluator)
structure APE (C A L : Type) :=
    (eval : L → C → A → C → bool)
```

```
-- APC structure (APC: Atomic Proposition Collector)
structure APC (C A L : Type) :=
    (eval : C → A → set L)

-- Acc structure (Acc: Acceptance)
structure Acc (C : Type) :=
    (accepting : set C)

-- P structure (P: Projection)
structure P (C A : Type) :=
    (projectC : C → Vc)
    (projectA : A → Va)

-----------------------------------------------------------------------------
--                        The EMI to STR conversion                       --
-----------------------------------------------------------------------------

-- Abstract types needed for defining EMI (more complex in reality)
def EMIModel := ℕ
def EMIDynamicMemory := list ℕ
def EMITransition := ℕ

-- EMI memory structure
structure EMI : Type :=
    (model : EMIModel)
    (dynamic : EMIDynamicMemory)

-- EMIState monad
@[reducible] def EMIState := state EMI

-- EMI API functions (not described here because their implementation depends
   on the language semantics and on implementation choices)
def get_configuration : EMIState EMIDynamicMemory:= sorry
def set_configuration (n : EMIDynamicMemory) : EMIState unit := sorry
def get_executable_steps : EMIState (set EMITransition) := sorry
def execute_step (t : EMITransition) : EMIState unit := sorry
def reset : EMIState unit := sorry
```

```
--Adapter functions for STR
def emi_initial : EMIState EMIDynamicMemory :=
do reset, c ← get_configuration, return c

def emi_actions (c : EMIDynamicMemory) : EMIState (set EMITransition) :=
do set_configuration c, a ← get_executable_steps, return a

def emi_execute (c : EMIDynamicMemory) (t : EMITransition) : EMIState
    EMIDynamicMemory :=
do set_configuration c, execute_step t, x ← get_configuration, return x

-- Conversion from EMI to STR
def EMI2STR (emi : EMI) : @STR EMIDynamicMemory EMITransition  :=
{
    initial := { prod.fst (emi_initial.run emi) },
    actions := λ c, prod.fst ((emi_actions c).run emi),
    execute := λ c a, { prod.fst ((emi_execute c a).run emi) }
}

------------------------------------------------------------------------------
--                                Debugging                                 --
------------------------------------------------------------------------------

-- DebugConfig structure
structure DebugConfig (C : Type) :=
    (current : option C)
    (trace : set C)
    (options : set C)

-- DebugAction inductive type
inductive DebugAction {C A : Type}
    | step : A → DebugAction
    | select : C → DebugAction
    | jump : C → DebugAction
    | run_to_breakpoint : DebugAction
```

```
-- Formal definition for interactive multiverse debugging
def interactive_multiverse_debugging
    (o : STR C A)
    (find_counterexample : set C → STR C A → set C → list C)
    (breakpoints : set C)
: STR (DebugConfig C) (@DebugAction C A) :=
{
    initial := { { DebugConfig . current := none,
        trace := ∅, options := o.initial } },
    actions := λ dc, match dc.current with
        | (some c) :=
            let
                oa := { x : @DebugAction C A  |
                    ∀ a ∈ (o.actions c), x = DebugAction.step a },
                ja := { x : @DebugAction C A  |
                    ∀ c ∈ dc.trace,    x = DebugAction.jump c }
            in
                oa ∪ ja ∪ { DebugAction.run_to_breakpoint }
        | none := { sa | ∀ c ∈ dc.options, sa = DebugAction.select c }
        end,
    execute := λ dc da, match dc.current, da with
        | (some c), (DebugAction.step a) := { { DebugConfig . current := none,
            trace :=    dc.trace, options := o.execute c a} }
        | (none)   , (DebugAction.step a) := ∅
        | _        , (DebugAction.select c) := { { DebugConfig . current := c,
            trace := {c} ∪ dc.trace, options := ∅ } }
        | _        , (DebugAction.jump c) := { { DebugConfig . current := c,
            trace := {c} ∪ dc.trace, options := ∅ } }
        | (some c), (DebugAction.run_to_breakpoint) :=
            match (find_counterexample { c } o breakpoints) with
                | []         := { { DebugConfig . current := dc.current,
                    trace := dc.trace, options := ∅ } }
                | a::ctrace := { { DebugConfig . current := a,
                    trace := {a} ∪ { x | x ∈ ctrace } ∪ dc.trace,
                    options := ∅ } }
            end
        | none, (DebugAction.run_to_breakpoint) :=
```

```
            match (find_counterexample dc.options o breakpoints) with
                | []         := { { DebugConfig . current := dc.current,
                    trace := dc.trace, options := ∅ } }
                | a::ctrace := { { DebugConfig . current := a,
                    trace := {a} ∪ { x | x ∈ ctrace } ∪ dc.trace,
                    options := ∅ } }
        end
    end
}

-------------------------------------------------------------------------------
--                          The STR to TR conversion                        --
-------------------------------------------------------------------------------

-- TR structure
structure TR (C : Type) :=
    (initial : set C)
    (next : C → set C)
    (accepting : set C)

-- Conversion from STR to TR
def STR2TR
    (str : STR C A)
    (acc : Acc C)
: @TR C  :=
{
    initial := str.initial,
    next := λ c, { n | ∀ a ∈ str.actions c, ∀ t ∈ str.execute c a, n = t },
    accepting := acc.accepting
}
```

```
-----------------------------------------------------------------------------
--                              Operators                                  --
-----------------------------------------------------------------------------

-- Type to know if some actions are available or if there is a deadlock
universe u
inductive completed (α : Type u)
    | deadlock {} : completed
    | some     : α → completed

-- Operator used to complete a STR by adding implicit transitions
def add_implicit_transitions
    (str : STR C A)
    [∀ c, decidable (str.actions c = ∅)]
    [∀ c, decidable (∀ (a : A), a ∈ str.actions c → str.execute c a = ∅)]
: STR C (completed A) :=
{
    initial := str.initial,
    actions := λ c,
        if (str.actions c = ∅) ∨
        (∀ a ∈ str.actions c, str.execute c a = ∅) then
            singleton completed.deadlock
        else
            { x | ∀ a ∈ str.actions c, x = completed.some a },
    execute := λ c oa, match oa with
        | completed.deadlock  := singleton c
        | completed.some a := { oc | ∀ t ∈ str.execute c a, oc = t }
        end
}

-- Synchronous composition operator
def synchronous_composition (C₁ C₂ A₁ A₂ L₁ : Type)
    (lhs : STR C₁ A₁)
    (ape : APE C₁ A₁ L₁)
    (rhs : STR C₂ A₂)
    (apc : APC C₂ A₂ L₁)
: STR (C₁ × C₂) (A₁ × A₂) :=
```

```
{
    initial := { c | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial), c = (c₁, c₂) },
    actions := λ c, { a |
        match c with
        | (c₁, c₂) := ∀ (a₁ ∈ lhs.actions c₁) (a₂ ∈ rhs.actions c₂)
            (t₁ ∈ lhs.execute c₁ a₁) (t₂ ∈ rhs.execute c₂ a₂),
            match t₁, t₂ : ∀ t₁ t₂, Prop with
            | t₁, t₂ := ∀ l ∈ apc.eval c₂ a₂,
                ape.eval l c₁ a₁ t₁ = tt → a = (a₁, a₂)
            end
        end
    },
    execute := λ c a, { t |
        match c, a with
        | (c₁, c₂), (a₁, a₂) :=  ∀ (t₁ ∈ lhs.execute c₁ a₁)
            (t₂ ∈ rhs.execute c₂ a₂), t = (t₁, t₂)
        end
    }
}

-- Filtering policy
structure FilteringPolicy (C A S : Type) :=
    (initial : S)
    (selector : set A)
    (apply : S → C → set A → set (S × A))
    (subset : ∀ s c 𝔸 (sa ∈ (apply s c 𝔸)), prod.snd sa ∈ 𝔸)

def StatelessFilteringPolicy (C A : Type) := FilteringPolicy C A unit

-- Scheduling policy
structure SchedulingPolicy (C A S : Type) extends (FilteringPolicy C A S) :=
    (unique : ∀ s c 𝔸 (a ∈ (apply s c 𝔸)) (b ∈ (apply s c 𝔸)), a = b)

def StatelessSchedulingPolicy (C A : Type) := SchedulingPolicy C A unit
```

```
-- Filtering operator
def filter
    (m : STR C A)
    (s : FilteringPolicy C A S)
: STR (C × S) (S × A) :=
{
    initial := { cs | ∀ (c ∈ m.initial), cs = (c, s.initial) },
    actions := λ cs,
        let toFilter := { a ∈ m.actions (prod.fst cs) | s.selector a },
            toForward := { fa | ∀ a ∈ m.actions (prod.fst cs),
                ¬ s.selector a → fa = (prod.snd cs, a) }
        in (s.apply (prod.snd cs) (prod.fst cs) toFilter) ∪ toForward,
    execute := λ cs sa,
        let r := m.execute (prod.fst cs) (prod.snd sa)
        in { x | ∀ c ∈ r, x = (c, (prod.fst sa)) }
}

-- Used to cast a scheduling policy into a filtering policy with the ↑ notation
instance scheduling_to_filtering :
    has_coe (SchedulingPolicy C A S)(FilteringPolicy C A S) := ⟨
   SchedulingPolicy.to_FilteringPolicy ⟩

-- Scheduling operator
def scheduler
    (m : STR C A)
    (s : SchedulingPolicy C A S)
: STR (C × S) (S × A) := filter C A S m ↑s

-- Asynchronous composition operator
def asynchronous_composition  { C₁ A₁ C₂ A₂ : Type }
    (lhs : STR C₁ A₁)
    (rhs : STR C₂ A₂)
: STR (C₁ × C₂) (A₁ ⊕ A₂) :=
{
    initial := { c | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial), c = (c₁, c₂) },
    actions := λ c, { a | match c : ∀ C, Prop with
        | (c₁, c₂) := ∀ (a₁ ∈ lhs.actions c₁) (a₂ ∈ rhs.actions c₂),
```

```
            a = sum.inl a₁ ∨ a = sum.inr a₂
        end },
    execute := λ c a, { a' | match c : ∀ C, Prop with
        | (c₁, c₂) := match a with
            | (sum.inl a₁) := ∀ c₁' ∈ lhs.execute c₁ a₁, a' = (c₁', c₂ )
            | (sum.inr a₂) := ∀ c₂' ∈ rhs.execute c₂ a₂, a' = (c₁ , c₂')
            end
        end }
}

notation lhs `⊗ₐ` rhs := asynchronous_composition lhs rhs

-------------------------------------------------------------------------------
--                   Software architecture formalisation                     --
-------------------------------------------------------------------------------

-- Runtime execution with statefull scheduling policy
def runtime_execution
    (system : STR C A)
    (scheduling_policy : SchedulingPolicy C A S)
: STR (C × S) ( S × A) := scheduler C A S system scheduling_policy

-- Model-checking with filtering
def model_checking_with_filtering
    (sys_and_env : STR C A)
    (filtering_policy : FilteringPolicy C A S)
: STR (C × S) (S × A) := filter C A S sys_and_env filtering_policy

-- Model-checking with scheduler in the verification loop
def model_checking_with_scheduling (S₁ S₂ : Type)
    (sys_and_env : STR C A)
    (scheduling_policy : SchedulingPolicy C A S₁)
    (filtering_policy : FilteringPolicy (C×S₁) (S₁×A) S₂)
: STR ((C×S₁)×S₂) (S₂×(S₁×A)) :=
    filter (C×S₁) (S₁×A) S₂
        (scheduler C A S₁ sys_and_env scheduling_policy)
        filtering_policy
```

```
-- Model-checking with decoupled environment
def model_checking_with_decoupled_env (C₁ C₂ A₁ A₂ S₁ S₂ : Type)
    (system : STR C₁ A₁)
    (env : STR C₂ A₂)
    (scheduling_policy : SchedulingPolicy C₁ A₁ S₁)
    (filtering_policy : FilteringPolicy ((C₁×S₁)×C₂) ((S₁×A₁) ⊕ A₂) S₂)
: STR (((C₁×S₁)×C₂)×S₂) (S₂×((S₁×A₁) ⊕ A₂))  :=
    filter ((C₁ × S₁) × C₂) ((S₁×A₁) ⊕ A₂) S₂
        ((scheduler C₁ A₁ S₁ system scheduling_policy) ⊗ₐ env)
        filtering_policy

end emi
```
Listing B.1 – Code Lean des définitions formelles de ce manuscrit.

ANNEXE C

# LANGAGE D'ACTION ET LANGAGE D'OBSERVATION POUR LES MODÈLES UML

---

Cette annexe présente les opérateurs du langage d'action et du langage d'observation utilisés pour la modélisation de modèles UML s'exécutant sur l'interpréteur EMI-UML. Ces opérateurs sont répartis en quatre sous-langages en fonction de s'ils accèdent en lecture ou en écriture aux données d'exécution issues du modèle ou définies en interne dans le moteur d'exécution. Ces quatre sous-langages sont le langage d'expression, le langage d'effet, l'extension du langage d'expression, et l'extension du langage d'effet. Le langage d'action regroupe les opérateurs du langage d'expression et ceux du langage d'effet. Le langage d'observation regroupe les opérateurs du langage d'expression et de son extension. Pour rappel, ces deux langages se basent sur des données externes déclarées et définies dans des modèles UML. La Figure C.1 présente le métamodèle des références globales pouvant être utilisées dans un modèle UML conforme au sous-ensemble d'UML considéré dans cette thèse. Des références globales sont fournies pour :

- *ClassRef* : référence globale pour les classes ;
- *ObjectRef* : référence globale pour les objets ;
- *StateRef* : référence globale pour les états des machines à états UML ;
- *TransitionRef* : référence globale pour les transitions des machines à états UML ;
- *EventRef* : référence globale pour les évènements ;
- *PortRef* : référence globale pour les ports.

Pour le prototype d'interpréteur EMI-UML, le langage d'action et le langage d'observation sont basés sur le langage C, le langage hôte de ce prototype. Ils définissent un ensemble d'opérateurs définis à l'aide de macros C pour manipuler les données d'exécution du modèle. Pour présenter ces opérateurs, nous allons maintenant décrire le métamodèle du langage d'expression, celui du langage d'effet et celui de l'extension du langage d'expression qui composent le langage d'action et le langage d'observation de notre prototype EMI-UML.

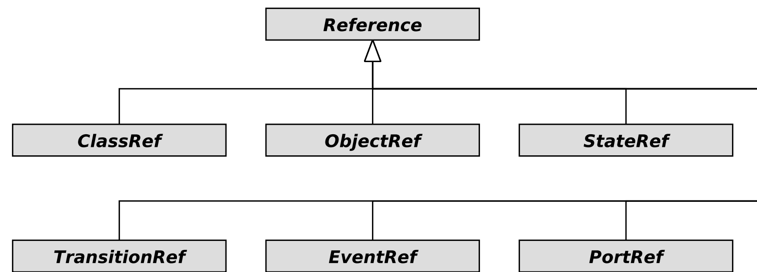



FIGURE C.1 – Métamodèle des références pour UML.

## C.1 Langage d'expression

Le langage d'expression est utilisé dans nos modèles UML, principalement pour exprimer les gardes sur les transitions des machines à états. La Figure C.2 présente le métamodèle de ce langage qui ajoute quatre opérateurs aux constructions du langage C :

— `This` (avec pour macro C `this`) permet d'accéder au contexte d'exécution courant (typiquement le contexte d'un objet) ;
— `Get` est utilisé pour naviguer le modèle (p. ex. pour récupérer la valeur d'un attribut) ;
— `At` est employé pour accéder à un élément d'une collection à un index donné ;
— `Call` permet d'appeler des fonctions sans effet de bords.

De manière générale, les macros C correspondantes à ces opérateurs peuvent être obtenues en mettant le nom de la métaclasse en majuscule et en changeant le style de définition de *camel case* à *snake case*. Par exemple, la macro associée à la métaclasse `Get` est `GET`.

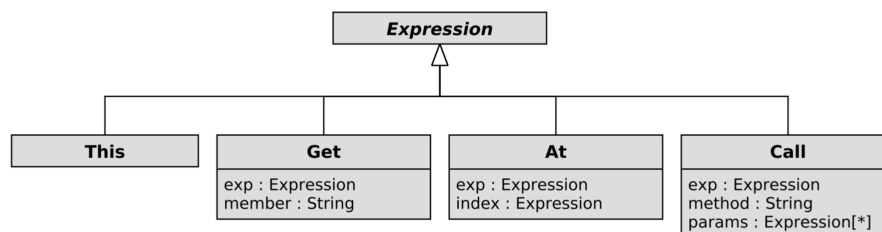



FIGURE C.2 – Métamodèle des opérateurs supplémentaires fournis par le langage d'expression pour UML.

## C.2 Langage d'effet

Le langage d'effet est essentiellement utilisé dans les modèles UML pour définir les effets sur les transitions des machines à états. N'importe quelle instruction C peut être utilisée pour effectuer cette tâche. La Figure C.3 qui illustre le métamodèle du langage d'effet montre que des opérateurs supplémentaires peuvent également être utilisés :

— `Set` permet d'affecter une nouvelle valeur à un attribut du modèle ;
— `SetAt` permet d'affecter une valeur à un index spécifique d'une collection ;
— `Inc`, `IncAt`, `Dec` et `DecAt` peuvent être utilisés pour incrémenter ou décrémenter un attribut du modèle. Ces quatre opérateurs ne sont cependant pas illustrés sur la Figure C.3 car ils fournissent seulement du sucre syntaxique basé sur les opérateurs `Set` et `SetAt` ;
— `Send` permet d'envoyer un évènement à un autre objet du modèle UML ;
— `Call` est utilisé pour appeler des méthodes avec effets de bord.

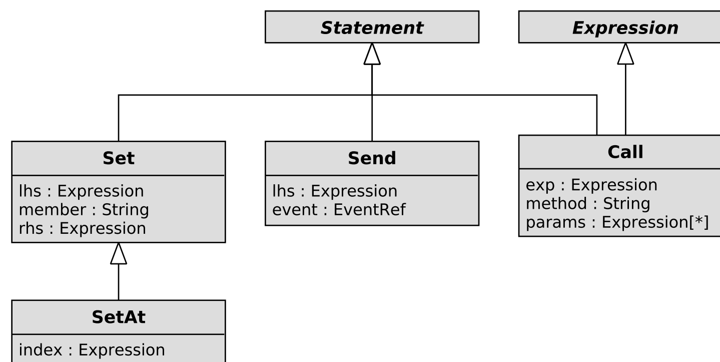



FIGURE C.3 – Métamodèle des opérateurs supplémentaires fournis par le langage d'effet pour UML.

## C.3 Extension du langage d'expression

Ce langage offre des opérateurs permettant d'accéder en lecture seule aux données d'exécution interne de l'interpréteur EMI-UML. Ce langage est utilisé pour définir les propositions atomiques pour la spécification de propriétés formelles. Les opérateurs supplémentaires fournis par l'extension du langage d'expression sont présentés sur le métamodèle de la Figure C.4. Cette extension fournit différentes métaclasses :
— `IsObject` permet de vérifier si l'objet courant est un objet donné ;
— `IsTypeOf` permet de vérifier si l'objet courant est une instance d'une classe donnée ;
— `IsInState` examine si l'état courant d'un objet actif est un état donné de sa machine à états ;
— `TransitionHasSource` permet de savoir si l'état source de la transition courante étant tirée est un état donné ;
— `TransitionHasTarget` permet de savoir si l'état cible de la transition courante étant tirée est un état donné ;
— `IsTransition` permet de savoir si la transition courante étant tirée est une transition donnée.

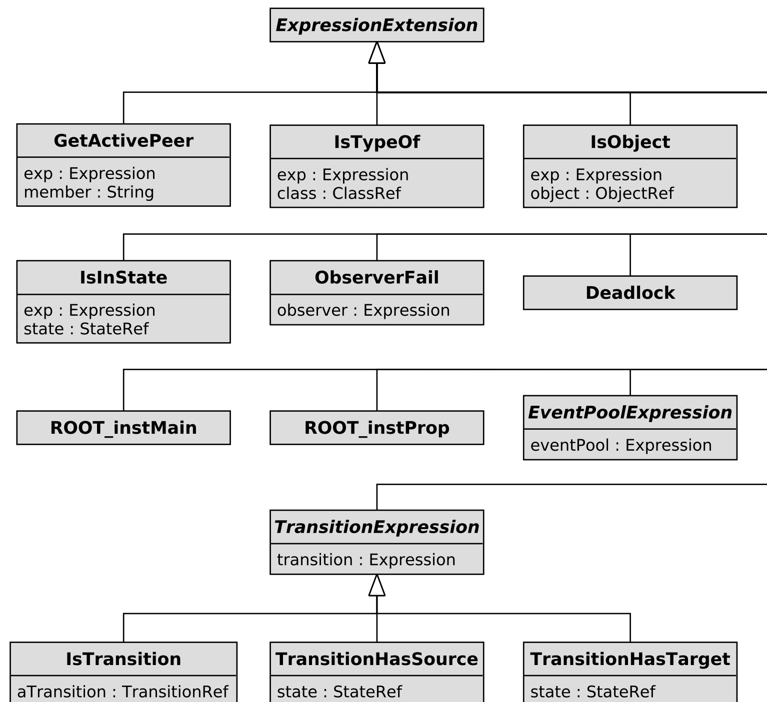



FIGURE C.4 – Métamodèle des opérateurs supplémentaires fournis par l'extension du langage d'expression pour UML.

En ce qui concerne la navigation du modèle, cette extension fournit les opérateurs suivants :

— `ROOT_instMain` (avec pour macro C ROOT_instMain) donne accès à la structure composite *Main* contenant les objets du système et de l'environnement ;
— `ROOT_instProp` (avec pour macro C ROOT_instProp) donne accès à la structure composite *Prop* contenant les PUSMs ;
— `GetActivePeer` permet d'avoir un accès direct aux objets actifs (p. ex. aux objets de l'environnement) connectés à l'autre bout des liens de communication (même s'ils contiennent des ports).

Cette extension du langage d'expression fournit également deux opérateurs permettant de faciliter la vérification formelle :

— `ObserverFail` vérifie si un automate observateur détecte une défaillance dans une configuration donnée (c.-à-d. s'il est dans un état d'acceptation) ;
— `Deadlock` devient vrai lorsque l'exécution du système est bloquée (c.-à-d. lorsqu'il y a un *deadlock*).

La Figure C.5 détaille les opérateurs disponibles pour introspecter le contenu des *event pools* (`EventPoolExp`) :

— `EpIsEmpty` permet de savoir si un *event pool* est vide ;
— `EpIsFull` permet de savoir si un *event pool* est plein ;

— `EpGetLength` permet d'accéder au nombre d'évènements actuellement stockés dans un *event pool* ;
— `EpGetFirst` permet d'accéder au premier évènement stocké dans un *event pool* (c.-à-d. l'évènement le plus vieux) ;
— `EpGetLast` permet d'accéder au dernier évènement stocké dans un *event pool* (c.-à-d. l'évènement le plus récent) ;
— `EpContains` est utilisé pour vérifier si un *event pool* contient une occurrence d'un évènement spécifique ;
— `EpContainsWithPort` est utilisé pour vérifier si un *event pool* contient une occurrence d'un évènement spécifique reçu sur un port donné.

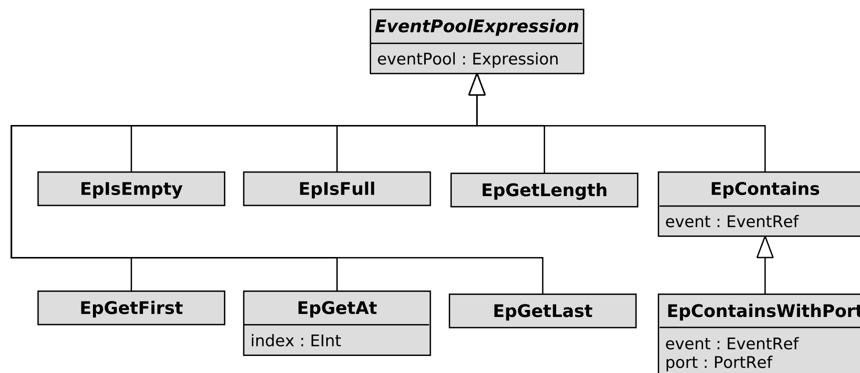



FIGURE C.5 – Métamodèle des opérateurs `EventPoolExp` fournis par l'extension du langage d'expression pour UML.

ANNEXE D

# PRÉSENTATION DES CAS D'ÉTUDES

Cette annexe présente une description détaillée des deux cas d'études utilisés au cours des expérimentations avec l'interpréteur EMI-UML.

## D.1 Contrôleur de passage à niveau

Le premier cas d'étude est un Contrôleur de Passage à Niveau (CPN) dont la description donnée ici est partiellement inspirée de [Bes+18c]. Un passage à niveau est un équipement utilisé à l'intersection des voies ferrées et des routes pour avertir et protéger les usagers de la route du trafic ferroviaire. Pour les besoins expérimentaux de cette thèse, un modèle UML du système logiciel permettant de piloter le passage à niveau a été conçu.

La Figure D.1 présente le diagramme de structure composite de ce modèle. La classe *Main*, classe racine de la structure composite, contient à la fois les objets du système et ceux de l'environnement. Elle est composée de deux capteurs *tcEntrance* et d'un capteur *tcExit* permettant de détecter le passage des trains, d'un contrôleur (*controller*), d'une barrière (*gate*) et de feux de signalisation (*roadSign* et *railwaySign*). L'environnement de ce modèle est représenté par un objet *train* représentant un train passant en boucle sur le passage à niveau. On considère ici que le passage à niveau ne peut être franchi que par un seul train à la fois et que le sens de franchissement est toujours le même.

Le modèle UML du CPN est constitué d'objets actifs dont le comportement est modélisé à

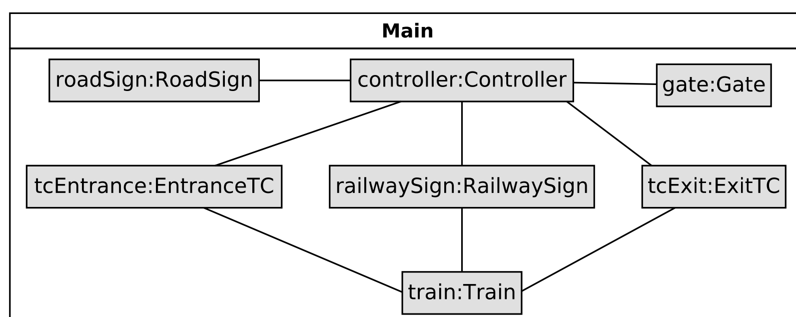



FIGURE D.1 – Diagramme de structure composite du modèle de CPN.

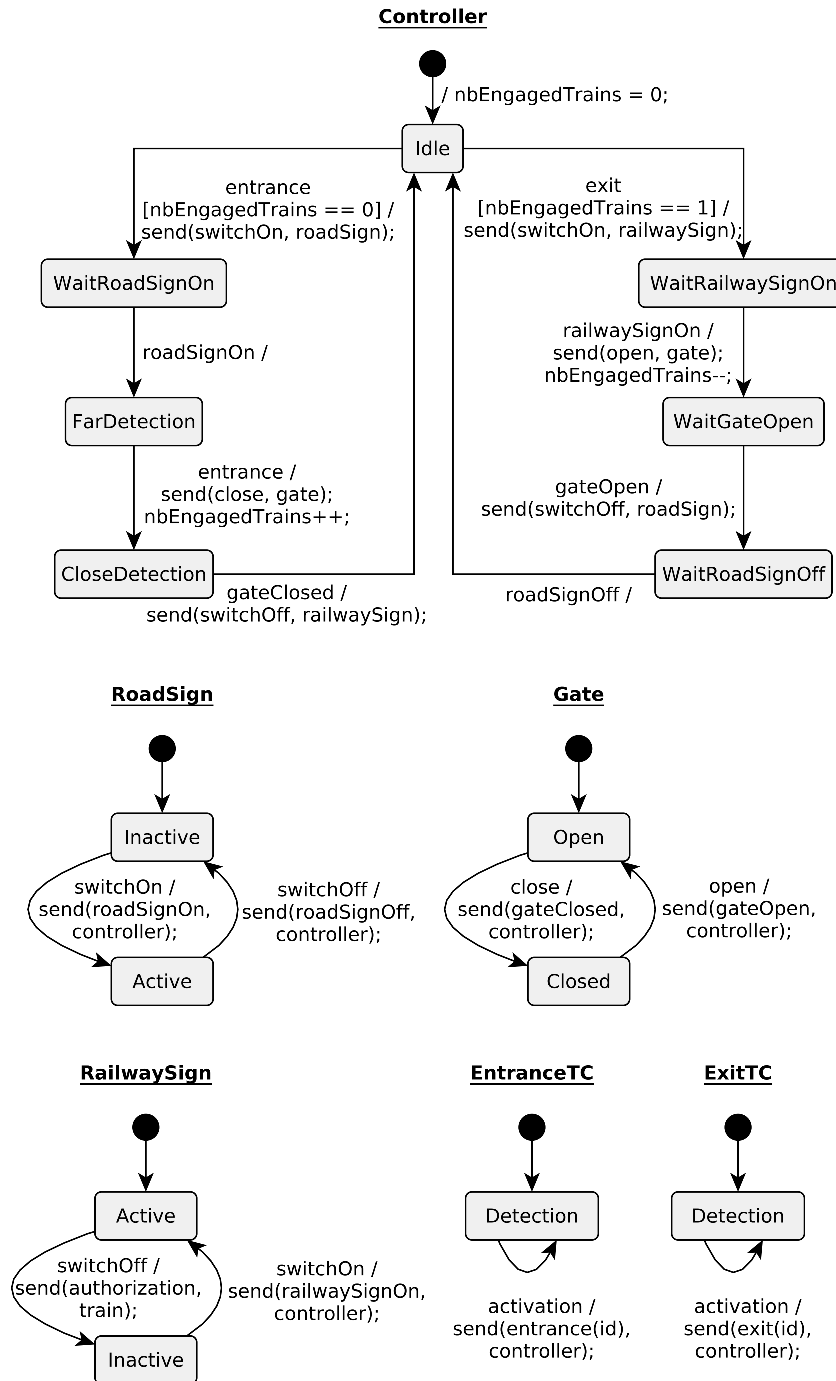



FIGURE D.2 – Machine à états du système du modèle de CPN.

l'aide des machines à états représentées en Figure D.2. Le contrôleur (*controller*) coordonne le mouvement de la barrière *gate* ainsi que l'activation des feux *roadSign* et *railwaySign* à partir des signaux reçus par les capteurs *tcEntrance* et *tcExit*. Les deux capteurs *tcEntrance* situés à l'entrée du passage à niveau avertissent le contrôleur de l'arrivée d'un train en envoyant l'évènement *entrance* au contrôleur. De la même façon, le capteur *tcExit* notifie le contrôleur avec l'évènement *exit* lorsqu'un train quitte le passage à niveau. Pour traiter ces évènements, la machine à états du contrôleur possède deux boucles. La première permet de fermer la barrière *gate* et d'allumer le feu *roadSign* avant le passage d'un train. Le système donne ensuite l'autorisation au train de passer en éteignant le feu *railwaySign*. La seconde boucle est responsable de l'ouverture de la barrière *gate* et de l'extinction du feu *roadSign* lorsque plus aucun train n'est engagé sur le passage à niveau.

La Figure D.3 présente la machine à états de l'objet *train* qui réalise une abstraction de l'environnement du CPN. Cette machine à états représente le passage d'un train activant les différents capteurs à tour de rôle. Elle modélise aussi le fait que le conducteur du train doit attendre l'extinction du feu de signalisation *railwaySign* avant de pouvoir s'engager sur la section de voie ferrée comportant le passage à niveau.

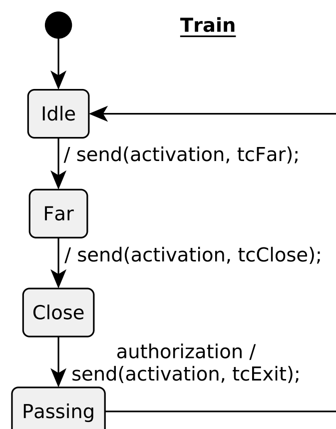



FIGURE D.3 – Machine à états de l'environnement du CPN.

## D.2 Interface du régulateur de vitesse

Le second cas d'étude correspond à l'Interface utilisateur d'un Régulateur de Vitesse (IRV) dont la description donnée ici est partiellement inspirée de [Bes+19d]. Un régulateur de vitesse (ou en anglais *Cruise Control System* (CCS)) contrôle automatiquement la vitesse du véhicule en ajustant son accélération pour maintenir une vitesse constante choisie par le conducteur. Pour ce cas d'étude, nous focalisons nos efforts de conception et de vérification sur l'interface

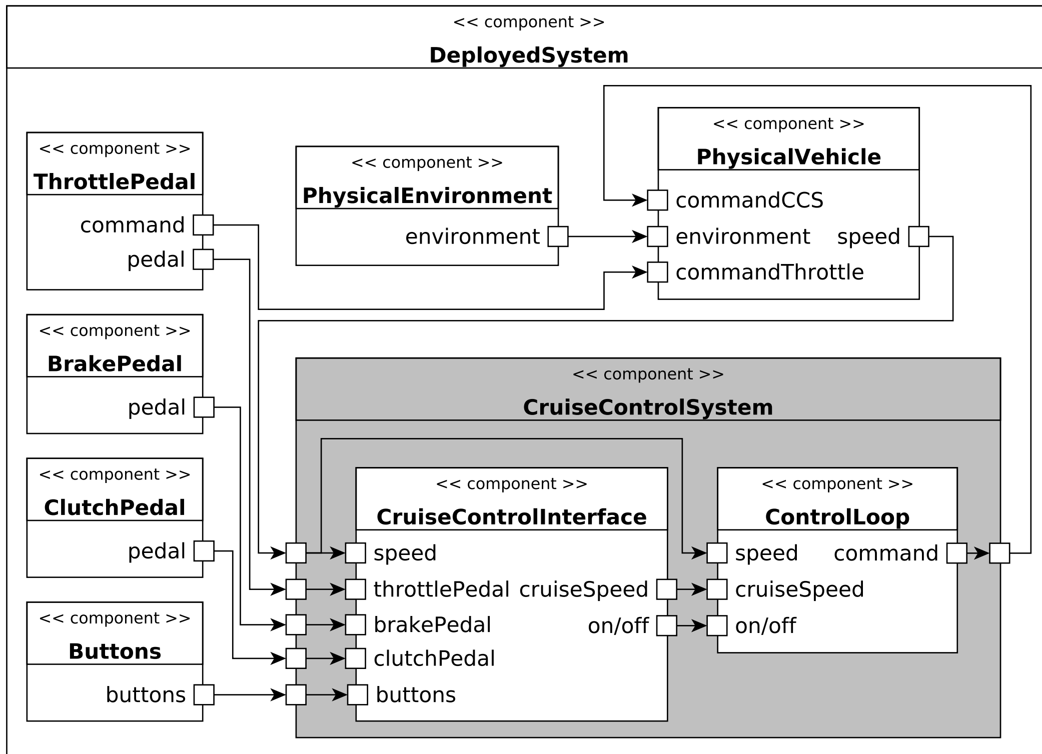



FIGURE D.4 – Diagramme de composants d’un régulateur de vitesse.

utilisateur, appelée IRV (ou en anglais *Cruise Control Interface* (CCI)), car ce sous-système contient une grande partie de la logique de contrôle du CCS. Ce cas d’étude a été conçu dans le cadre de cette thèse en se basant partiellement sur les travaux [DLT14 ; LDD14] et les expériences antérieures de certains membres de l’équipe sur ce type de système.

La Figure D.4 présente un diagramme de composants d’un CCS montrant ses interactions avec le conducteur et le véhicule. Ce diagramme de composants a été conçu avec l’hypothèse que le CCS opère indépendamment de tout autre système (p. ex. l’*Electronic Stability Program* (ESP)). Comme illustré sur ce diagramme, le CCS interagit avec le procédé physique à contrôler, appelé ici *PhysicalVehicle*. Ce composant prend en entrée la commande venant du *CruiseControlSystem* et la commande venant de la pédale d’accélération (*ThrottlePedal*) pour piloter automatiquement ou manuellement le moteur du véhicule. La dernière entrée provient de l’environnement (*PhysicalEnvironment*) qui peut appliquer certaines forces sur le véhicule (p. ex. le profil de la route, la friction de l’air) et perturber sa conduite. Le véhicule est aussi équipé d’un capteur permettant de mesurer la vitesse courante. Les valeurs capturées sont fournies comme entrées au *CruiseControlSystem*.

Le *CruiseControlSystem* est constitué de deux composants : l’interface utilisateur (*CruiseControlInterface*) et la boucle de régulation (*ControlLoop*). L’interface utilisateur (CCI) est

responsable de la gestion de toutes les entrées reçues par le CCS : les données des trois pédales du véhicule permettant de savoir si chaque pédale est appuyée ou relâchée, et les données provenant des boutons sur lesquels l'utilisateur peut appuyer pour contrôler le CCS. Avec ces données, le CCI pilote la boucle de régulation (*ControlLoop*) en charge d'ajuster la vitesse du véhicule à la vitesse de croisière calculée par le CCI. La boucle de régulation est ici considérée comme une boite noire qui exécute un algorithme de régulation pour calculer la commande devant être appliquée sur le moteur du véhicule (*PhysicalEngine*).

Dans ce cas d'étude, nous nous focalisons sur la conception et la vérification du sous-système du CCI. Par conséquent, tous les composants externes au CCI sont considérés comme faisant partie de l'environnement de ce sous-système. La Figure D.4 a ainsi permis d'acquérir une meilleure compréhension de cet environnement pour en réaliser une abstraction pertinente pour la phase de vérification.

Pour appliquer l'approche EMI sur ce cas d'étude, nous avons modéliser le CCI sous la forme d'un modèle UML. Le diagramme de structure composite de ce modèle est illustré sur la Figure D.5. La classe *Main* est la classe composite racine du modèle. Elle contient la part *cci*, qui correspond au système à l'étude, et la part *env* qui modélise son environnement. Ces deux éléments communiquent en échangeant des signaux à travers des ports. Le modèle sépare ici de façon distincte le système et l'environnement sous forme d'un modèle UML modulaire comme présenté dans notre contribution en [Bes+20].

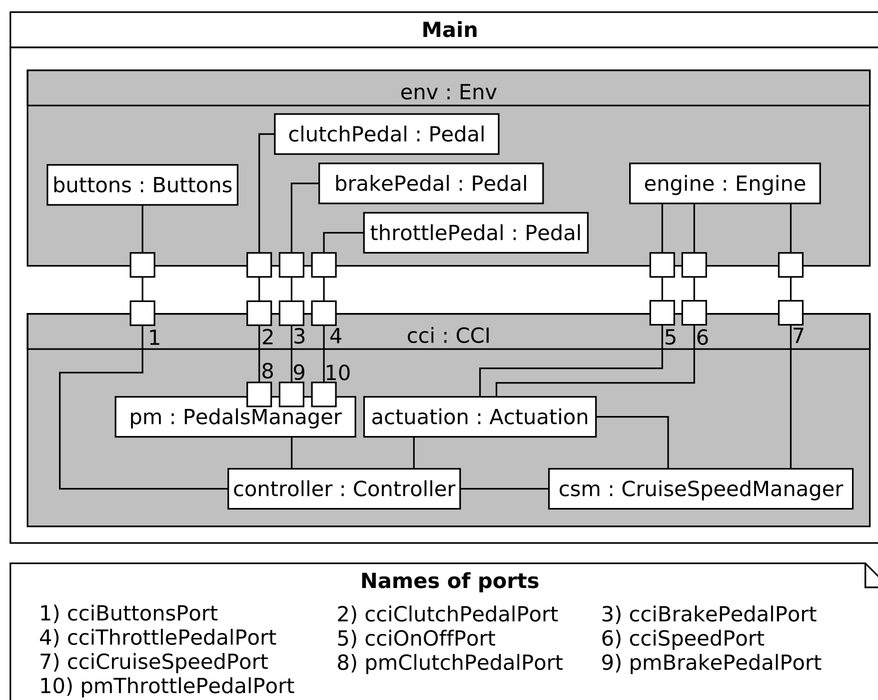
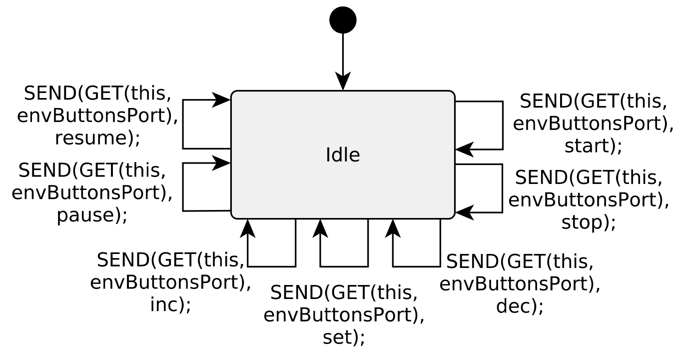



FIGURE D.5 – Diagramme de structure composite du modèle d'IRV.

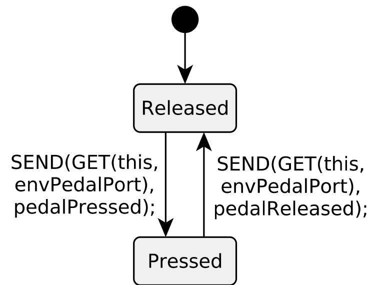
**Abstraction de l'environnement.** La part *env* contient un objet *buttons* (Figure D.6a) qui modélise les différents boutons (c.-à-d. *start*, *stop*, *inc*, *dec*, *set*, *pause*, *resume*) qui peuvent être manipulés, ainsi que les trois pédales (c.-à-d. la pédale d'embrayage *clutchPedal*, la pédale de frein *breakPedal* et la pédale d'accélération *throttlePedal*) (Figure D.6b) qui peuvent être appuyées ou relâchées par le conducteur. En accord avec la Figure D.4, le véhicule (*PhysicalVehicle*) et la boucle de régulation (*ControlLoop*) font aussi partie de l'environnement. Dans notre modèle UML, ils ont été abstraits dans l'objet *engine* (Figure D.6c). Dans un véhicule réel, le CCS va essayer d'ajuster la vitesse du véhicule à la vitesse de croisière (ou en anglais *cruise speed*) donnée par le CCI, mais à cause de contraintes physiques (p. ex. le profil de la route, la friction de l'air), il n'est pas toujours possible pour le CCS de maintenir le véhicule à cette vitesse. Pour prendre cela en compte, le moteur (*engine*) ne réalise aucune corrélation entre la vitesse de croisière donnée en entrée et la vitesse courante qu'il retourne. Ainsi, la vitesse courante peut aller de façon non-déterministe de 0 à 200 km/h en un seul pas. Cette abstraction permet de considérer un sur-ensemble de tous les cas possibles pour les activités de vérification. Les machines à états des différents objets de l'environnement sont illustrées sur la Figure D.6. Il s'agit principalement de machines à états pâquerettes permettant de considérer tous les entrelacements d'évènements possibles. On notera que pour considérer que le moteur est plus réactif que les boutons et les pédales, il faut ajouter la garde suivante sur toutes les transitions de ces deux objets : "`EP_IS_EMPTY(GET(GET(ROOT_instMain, env), engine))`". Cette hypothèse ainsi que l'hypothèse de réactivité ont été prises en compte pour nos expérimentations.

**Système à l'étude.** La part *cci* décrit le système que nous voulons vérifier. Ce système vise à envoyer de nouvelles consignes (c.-à-d. la valeur courante de la vitesse de croisière) au moteur (*engine*) en accord avec les actions de l'utilisateur et la vitesse courante du véhicule. Le comportement de ces objets actifs contenus dans le *cci* est défini par des machines à états illustrées en Figure D.7. Ces machines à états utilisent le langage d'action d'EMI-UML décrit en annexe C. Le *controller* (le contrôleur en Figure D.7a) reçoit les évènements de *buttons* et de *pm* (le manager de pédales en Figure D.7b), qui est connecté aux trois pédales (*clutchPedal*, *breakPedal*, et *throttlePedal*) à travers des ports. À partir de ces évènements, le contrôleur détermine le statut du CCS et délègue la génération des évènements de sortie aux objets *actuation* et *csm*. L'*actuation* (Figure D.7c) envoie des signaux *On* et *Off* pour respectivement activer la boucle de régulation lorsque le CCS est engagé (c.-à-d. le CCS est activé et agit sur le moteur) et la désactiver lorsque le CCS est éteint ou désengagé (c.-à-d. le CCS est activé mais n'agit pas sur le moteur). Le *csm* (le manager de la vitesse de croisière en Figure D.7d) calcule la vitesse de croisière en accord avec les évènements des boutons filtrés par le contrôleur et envoie de nouvelles consignes chaque fois que l'*actuation* le demande. Sur toutes ces

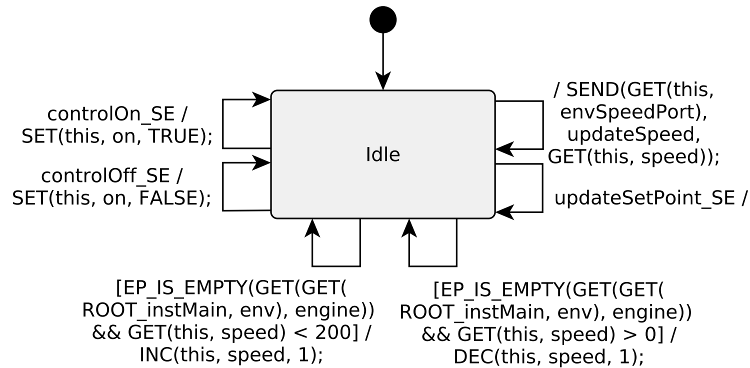
machines à états, des transitions supplémentaires (d'un état vers lui-même) peuvent être nécessaires pour ignorer explicitement certains évènements conformément à la stratégie utilisée par l'interpréteur pour traiter les évènements. Pour finir, on notera que la transition en gras sur la Figure D.7a correspond à la transition à l'origine du bogue identifié en section 10.4.5.2 et dont la correction est proposée en Figure 10.11.




(a) Machine à états de la classe *Buttons*.




(b) Machine à états de la classe *Pedal*.

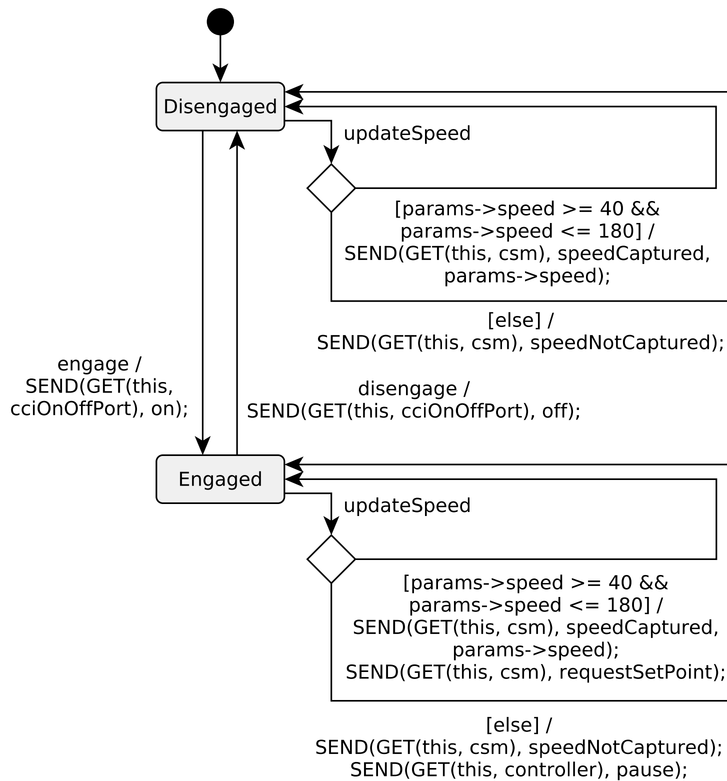



(c) Machine à états de la classe *Engine*.

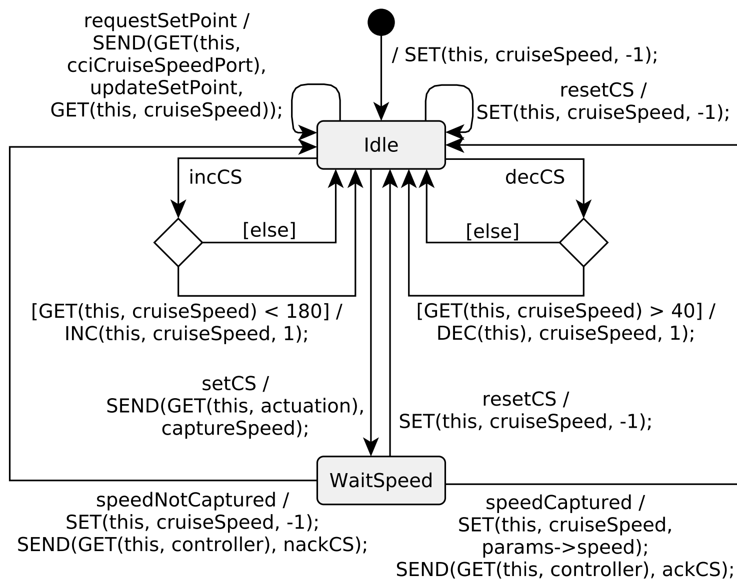
FIGURE D.6 – Machines à états de l'environnement de l'IRV.

(a) Machine à états de la classe *Controller*.

(b) Machine à états de la classe *PedalsManager*.




(c) Machine à états de la classe *Actuation*.




(d) Machine à états de la classe *CruiseSpeedManager*.

FIGURE D.7 – Machines à états du modèle d'IRV.

ANNEXE E

# LISTE DES PROPOSITIONS ATOMIQUES

---

Cette section décrit toutes les propositions atomiques utilisées dans ce manuscrit de thèse pour la vérification de modèles avec l'interpréteur EMI-UML. Pour chaque proposition atomique, nous donnons sa signification en langage naturel (`[LN]`), son intuition en pseudo-code (`[PC]`) avec une notation pointée et le prédicat exprimé dans le langage d'observation (`[LO]`) d'EMI-UML.

## E.1 Liste des propositions atomiques pour le modèle du contrôleur de passage à niveau.

`[LN] trainIsPassing` : vérifie si l'état courant de la machine à états du *train* est dans l'état *Passing*.

`[PC] trainIsPassing = "ROOT_instMain.train.IsInState(STATE_Train_Passing)"`

`[LO] trainIsPassing = "IS_IN_STATE(GET(ROOT_instMain, train), STATE_Train_Passing)"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] gateIsOpen` : vérifie si l'état courant de la machine à états de la barrière *gate* est dans l'état *Open*.

`[PC] gateIsOpen = "ROOT_instMain.gate.IsInState(STATE_Gate_Open)"`

`[LO] gateIsOpen = "IS_IN_STATE(GET(ROOT_instMain, gate), STATE_Gate_Open)"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] gateIsClosed` : vérifie si l'état courant de la machine à états de la barrière *gate* est dans l'état *Closed*.

`[PC] gateIsClosed = "ROOT_instMain.gate.IsInState(STATE_Gate_Closed)"`

`[LO] gateIsClosed = "IS_IN_STATE(GET(ROOT_instMain, gate), STATE_Gate_Closed)"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] roadSignIsOff` : vérifie si l'état courant de la machine à états du feu de signalisation *roadSign* est dans l'état *Inactive*.

`[PC] roadSignIsOff = "ROOT_instMain.roadSign.IsInState(STATE_RoadSign_Inactive)"`

[LO] roadSignIsOff = "IS_IN_STATE(GET(ROOT_instMain, roadSign), STATE_RoadSign_Inactive)"

---

[LN] roadSignIsOn : vérifie si l'état courant de la machine à états du feu de signalisation *roadSign* est dans l'état *Active*.

[PC] roadSignIsOn = "ROOT_instMain.roadSign.IsInState(STATE_RoadSign_Active)"

[LO] roadSignIsOn = "IS_IN_STATE(GET(ROOT_instMain, roadSign), STATE_RoadSign_Active)"

## E.2 Liste des propositions atomiques pour le modèle d'interface du régulateur de vitesse.

[LN] ccsDisengaged : vérifie si l'état courant de la machine à états de *actuation* est dans l'état *Disengaged*.

[PC] ccsDisengaged = "ROOT_instMain.cci.actuation.IsInState(STATE_Actuation_Disengaged)"

[LO] ccsDisengaged = "IS_IN_STATE(GET(GET(ROOT_instMain, cci), actuation), STATE_Actuation_Disengaged)"

---

[LN] ccsEngaged : vérifie si l'état courant de la machine à états de *actuation* est dans l'état *Engaged*.

[PC] ccsEngaged = "ROOT_instMain.cci.actuation.IsInState(STATE_Actuation_Engaged)"

[LO] ccsEngaged = "IS_IN_STATE(GET(GET(ROOT_instMain, cci), actuation), STATE_Actuation_Engaged)"

---

[LN] evStop : vérifie si l'*event pool* de *csm* contient une occurrence du signal *stop*.

[PC] evStop = "ROOT_instMain.cci.csm.EpContains(SIGNAL_stop)"

[LO] evStop = "EP_CONTAINS(GET(GET(ROOT_instMain, cci), csm), SIGNAL_stop)"

---

[LN] evSet : vérifie si l'*event pool* de *csm* contient une occurrence du signal *set*.

[PC] evSet = "ROOT_instMain.cci.csm.EpContains(SIGNAL_set)"

[LO] evSet = "EP_CONTAINS(GET(GET(ROOT_instMain, cci), csm), SIGNAL_set)"

---

[LN] canResume : vérifie si l'attribut *canResume* de *pm* est vrai.

`[PC] canResume = "ROOT_instMain.cci.pm.canResume == TRUE"`

`[LO] canResume = "GET(GET(GET(ROOT_instMain, cci), pm),`
`canResume) == TRUE"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evThrottleReleased` : vérifie si l'*event pool* de *pm* contient une occurrence du signal *pedalReleased* reçue sur le port *pmThrottlePedalPort*.

`[PC] evThrottleReleased = "ROOT_instMain.cci.pm.EpContainsWithPort(`
`SIGNAL_pedalReleased, PORT_PedalsManagerPedalPort_pmThrottlePedalPort)"`

`[LO] evThrottleReleased = "EP_CONTAINS_WITH_PORT(GET(GET(ROOT_instMain, cci),pm),`
`SIGNAL_pedalReleased, PORT_PedalsManagerPedalPort_pmThrottlePedalPort)"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evResume` : vérifie si l'*event pool* de *controller* contient une occurrence du signal *resume*.

`[PC] evResume = "ROOT_instMain.cci.controller.EpContains(SIGNAL_resume)"`

`[LO] evResume = "EP_CONTAINS(GET(GET(ROOT_instMain, cci), controller),`
`SIGNAL_resume)"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evOff` : vérifie si le premier évènement dans l'*event pool* de l'objet relié à *actuation* via le port *cciOnOffPort* est une occurrence du signal *off*.

`[PC] evOff = "ROOT_instMain.cci.actuation.GetActivePeer(cciOnOffPort).`
`EpGetFirst() == SIGNAL_Off"`

`[LO] evOff = "EP_GET_FIRST(GET_ACTIVE_PEER(GET(GET(ROOT_instMain, cci),`
`actuation), cciOnOffPort)) == SIGNAL_Off"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evOn` : vérifie si le premier évènement dans l'*event pool* de l'objet relié à *actuation* via le port *cciOnOffPort* est une occurrence du signal *on*.

`[PC] evOn = "ROOT_instMain.cci.actuation.GetActivePeer(cciOnOffPort).`
`EpGetFirst() == SIGNAL_On"`

`[LO] evOn = "EP_GET_FIRST(GET_ACTIVE_PEER(GET(GET(ROOT_instMain, cci),`
`actuation), cciOnOffPort)) == SIGNAL_On"`

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evUpdateSetPoint` : vérifie si le premier évènement dans l'*event pool* de l'objet relié à *actuation* via le port *cciOnOffPort* est une occurrence du signal *updateSetPoint*.

`[PC] evUpdateSetPoint = "ROOT_instMain.cci.actuation.GetActivePeer(cciOnOffPort).`
`EpGetFirst() == SIGNAL_updateSetPoint"`

```
[LO] evUpdateSetPoint = "EP_GET_FIRST(GET_ACTIVE_PEER(GET(GET(ROOT_instMain, cci),
actuation), cciOnOffPort)) == SIGNAL_updateSetPoint"
```

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] intervalCS` : vérifie si la vitesse de croisière *cruiseSpeed* est dans son intervalle de fonctionnement [40, 180] km/h.

```
[PC] intervalCS = "ROOT_instMain.cci.csm.cruiseSpeed >= 40
&& ROOT_instMain.cci.csm.cruiseSpeed <= 180"

[LO] intervalCS = "GET(GET(GET(ROOT_instMain, cci), csm), cruiseSpeed) >= 40
&& GET(GET(GET(ROOT_instMain, cci), csm), cruiseSpeed) <= 180"
```

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] unknownCS` : vérifie si la vitesse de croisière *cruiseSpeed* est égale à -1.

```
[PC] unknownCS = "ROOT_instMain.cci.csm.cruiseSpeed == -1"

[LO] unknownCS = "GET(GET(GET(ROOT_instMain, cci), csm), cruiseSpeed) == -1"
```

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] failureObserved` : vérifie si l'automate *Observateur6* (*observer6* en anglais dans le modèle) atteint un état d'acceptation.

```
[PC] failureObserved = "observer6.ObserverFail()"

[LO] failureObserved = "OBSERVER_FAIL(observer6)"
```

— — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — — —

`[LN] evStopController` : vérifie si l'*event pool* du *controller* contient le signal *stop*.

```
[PC] evStopController = "ROOT_instMain.cci.controller.EpContains(SIGNAL_stop)"

[LO] evStopController = "EP_CONTAINS(GET(GET(ROOT_instMain, cci), controller),
SIGNAL_stop)"
```

ANNEXE F

# PAGES WEB ARCHIVÉES

Les liens suivants pointent vers des versions spécifiques des pages web référencées dans ce document telles qu'elles étaient au moment de sa création. Il reste possible de naviguer ensuite vers d'autres versions plus anciennes ou plus récentes stockées sur https://web.archive.org/.

| Page web | Lien de la page web archivée |
|---|---|
| ALE | https://web.archive.org/web/20200821113722/http://gemoc.org/ale-lang/ |
| ALF | https://web.archive.org/web/20200821121218/https://www.omg.org/spec/ALF/1.1/PDF |
| Alt-Ergo | https://web.archive.org/web/20200821114201/http://alt-ergo.lri.fr/ |
| Atelier B | https://web.archive.org/web/20200821114607/https://www.atelierb.eu/ |
| B Toolkit | https://web.archive.org/web/20200825114227/https://github.com/edwardcrichton/BToolkit |
| ClearSy | https://web.archive.org/web/20200821114707/https://www.clearsy.com/ |
| CompCert | https://web.archive.org/web/20200821161537/http://compcert.inria.fr/ |
| Coq | https://web.archive.org/web/20200821093704/https://coq.inria.fr/ |
| DAP | https://web.archive.org/web/20200821120633/https://microsoft.github.io/debug-adapter-protocol/ |
| FMI | https://web.archive.org/web/20200821120721/https://fmi-standard.org/ |
| Frama-C | https://web.archive.org/web/20200723115638/http://frama-c.com/ |
| fUML | https://web.archive.org/web/20200702010249/https://www.omg.org/spec/FUML/1.4/PDF |
| fUML réf. impl. | https://web.archive.org/web/20200821113256/http://modeldriven.github.io/fUML-Reference-Implementation/ |
| GCC | https://web.archive.org/web/20200821121807/https://gcc.gnu.org/ |
| GDB | https://web.archive.org/web/20200821120357/https://www.gnu.org/software/gdb/ |
| Gemoc Studio | https://web.archive.org/web/20200821113543/http://gemoc.org/studio.html |
| HaLVM | https://web.archive.org/web/20200821143432/https://github.com/GaloisInc/HaLVM |
| Isabelle | https://web.archive.org/web/20200821114301/https://isabelle.in.tum.de/ |
| JDB | https://web.archive.org/web/https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html |
| LLVM | https://web.archive.org/web/20200821121850/https://llvm.org/ |
| LSP | https://web.archive.org/web/20200821120554/https://microsoft.github.io/language-server-protocol/ |

| Page web | Lien de la page web archivée |
| --- | --- |
| MARTE | https://web.archive.org/web/20200821115724/https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf |
| MDETools'19 | https://web.archive.org/web/20200821143948/https://mdetools.github.io/mdetools19/challengeproblem.html |
| MISRA | https://web.archive.org/web/20200821131921/https://www.misra.org.uk/ |
| Modèle de défibrillateur | https://web.archive.org/web/20200821143825/https://github.com/Pyponou/defibrillator |
| Moka | https://web.archive.org/web/https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution |
| MPS | https://web.archive.org/web/20200821121521/https://www.jetbrains.com/mps/ |
| OBP2 | https://web.archive.org/web/20200821092138/http://www.obpcdl.org/ |
| OBP2 onglet EMI | https://web.archive.org/web/20200821092234/http://www.obpcdl.org/bare-metal-uml/ |
| Patrons de Dwyer | https://web.archive.org/web/20200821143133/https://matthewbdwyer.github.io/psp/patterns/ltl.html |
| PlantUML | https://web.archive.org/web/20200821144318/https://plantuml.com/ |
| PSCS | https://web.archive.org/web/20200702011357/https://www.omg.org/spec/PSCS/1.2/PDF |
| PSSM | https://web.archive.org/web/20200821120938/https://www.omg.org/spec/PSSM/1.0/PDF |
| PVS | https://web.archive.org/web/20200821114406/https://pvs.csl.sri.com/ |
| QVT | https://web.archive.org/web/20200702010155/https://www.omg.org/spec/QVT/1.3/PDF |
| Regehr blog | https://web.archive.org/web/20200821121402/https://blog.regehr.org/archives/213 |
| *Sanitizers* | https://web.archive.org/web/20200821143557/https://clang.llvm.org/docs/MemorySanitizer.html |
| Sirius | https://web.archive.org/web/20200821092907/http://www.eclipse.org/sirius/ |
| Spin | https://web.archive.org/web/20200821114956/http://spinroot.com/spin/whatispin.html |
| Spoofax | https://web.archive.org/web/20200821121612/http://www.metaborg.org/en/latest/index.html |
| UML | https://web.archive.org/web/20200821120011/https://www.omg.org/spec/UML/2.5.1/PDF |
| UML Designer | https://web.archive.org/web/20200821132102/http://www.umldesigner.org/ |
| Valgrind | https://web.archive.org/web/20200821143707/https://valgrind.org/ |
| Why3 | https://web.archive.org/web/20200821114459/http://why3.lri.fr/ |
| XMI | https://web.archive.org/web/20200702005741/https://www.omg.org/spec/XMI/2.5.1/PDF |
| Xtend | https://web.archive.org/web/20200814083634/https://www.eclipse.org/xtend/ |
| xtUML | https://web.archive.org/web/20200821121720/https://xtuml.org/ |
| Z3 | https://web.archive.org/web/20200821143317/https://github.com/Z3Prover/z3 |

TABLE F.1 – Liens des pages web archivées.

# ACRONYMES

**$A_{cc}$** *Acceptance*.

**AADL** *Architecture Analysis and Design Language*.

**ALE** *Action Language for Ecore*.

**ALF** *Action Language for Foundational UML*.

**ANTLR** *ANother Tool for Language Recognition*.

**APC** *Atomic Propositions Collector*.

**APE** *Atomic Proposition Evaluator*.

**API** *Application Programming Interface*.

**AST** *Abstract Syntax Tree*.

**ATL** *ATLAS Transformation Language*.

**BDD** *Binary Decision Diagram*.

**BPMN** *Business Process Model And Notation*.

**CCI** *Cruise Control Interface*.

**CCS** *Cruise Control System*.

**CCSL** *Clock Constraint Specification Language*.

**CIM** *Computational Independent Model*.

**CSP** *Communicating Sequential Processes*.

**CTL** *Computational Tree Logic*.

**DAP** *Debug Adapter Protocol*.

**DBM** *Difference Bounds Matrix*.

**DFS** *Depth First Search*.

**DSL** *Domain Specific Language*.

**DSML** *Domain Specific Modeling Language*.

**DTD** *Document Type Definition*.

**EBNF** *Extended Backus-Naur Form*.

**EDI** Environnement de Développement Intégré.

**EEU** *Environment Execution Unit*.

**EHA** *Extended Hierarchical Automata*.

**EMF** *Eclipse Modeling Framework*.

**EMI** *Embedded Model Interpreter*.

**ESM** *Executable State Machine*.

**ESP** *Electronic Stability Program*.

**ETL** *Epsilon Transformation Language*.

**EVA** *Evolved Value Analysis*.

**FIFO** *First In First Out*.

**FMI** *Functional Mockup Interface*.

**fUML** *foundational UML*.

**GPL** *General Purpose Language*.

**GPML** *General Purpose Modeling Language*.

**IDM** Ingénierie Dirigée par les Modèles.

**IPC** *Inter-Process Communication*.

**JVM** *Java Virtual Machine*.

**LSP** *Langage Server Protocol*.

**LTL** Logique Temporelle Linéaire.

**MDA** *Model Driven Architecture*.

**MDE** *Model Driven Engineering*.

**MoCCML** *Model of Concurrency and Communication Modeling Language*.

**MOF** *Meta Object Facility*.

**OCL** *Object Constraint Language*.

**OMG** *Object Management Group*.

**OS** *Operating System*.

**P** *Projection*.

**PC** *Personal Computer*.

**PDM** *Plateform Description Model*.

**PIM** *Plateform Independent Model*.

**PSC** *Property Sequence Chart*.

**PSCS** *Precise Semantics of UML Composite Structures*.

**PSM** *Plateform Specific Model*.

**PSOT** *Precise Semantics of Time*.

**PSSM** *Precise Semantics of UML State Machines*.

**PUSM** *Property UML State Machine*.

**QVT** *Query/View/Transformation*.

**RTD** *RunTime Data*.

**SDL** *Specification and Description Language*.

**SEU** *System Execution Unit*.

**SoC** *System on Chip*.

**SOS** *Structural Operational Semantics*.

**SPEM** *Software & Systems Process Engineering Metamodel*.

**STR** *Semantic Transition Relation*.

**SVG** *Scalable Vector Graphics*.

**SysML** *Systems Modeling Language*.

**TCP** *Transmission Control Protocol*.

**TGG** *Triple Graph Grammars*.

**TGV** Train à Grande Vitesse.

**TOPCASED** *Toolkit in Open Source for Critical Applications & Systems Development*.

**TR** *Transition Relation*.

**UML** *Unified Modeling Language*.

**V&V** Vérification et Validation.

**VDM** *Vienna Development Method*.

**VDM-SL** *VDM Specification Language*.

**VM** *Virtual Machine*.

**WCET** *Worst Case Execution Time*.

**xDSML** DSML eXécutable.

**XMI** *XML Metadata Interchange*.

**XML** *eXtensible Markup Language*.

# Table des figures


1 Aperçu des problèmes, des objectifs et des contributions scientifiques. . . . . . 5
2 Plan de lecture de ce manuscrit. . . . . . . . . . . . . . . . . . . . . . . . . . . 8

1.1 Définition d'un langage basé soit sur sa syntaxe concrète (en vert) soit sur sa syntaxe abstraite (en bleu). (Figure adaptée de [Com08]) . . . . . . . . . . . . 17
1.2 Sémantique opérationnelle de la fonction $add$. . . . . . . . . . . . . . . . . . . . 19
1.3 Les relations "conforme à" et "implémente" . . . . . . . . . . . . . . . . . . . . . 22
1.4 Approche d'exécution basée sur une sémantique translationnelle. . . . . . . . . . 24
1.5 Approche d'exécution basée sur une sémantique opérationnelle. . . . . . . . . . 26

2.1 Schéma de principe du *model-checking*. . . . . . . . . . . . . . . . . . . . . . . 37
2.2 Schéma de principe du *model-checking* avec automates de Büchi. . . . . . . . 38
2.3 Schéma de l'approche par raffinement. . . . . . . . . . . . . . . . . . . . . . . . 42
2.4 Schéma de l'approche utilisant une transformation vers un langage d'analyse. . 44
2.5 Schéma de l'approche d'analyse dédiée au langage. . . . . . . . . . . . . . . . 47
2.6 Schéma de l'approche d'analyse par API. . . . . . . . . . . . . . . . . . . . . . 49

3.1 Schéma de A#1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.2 Schéma de A#2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
3.3 Schéma de A#3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.4 Schéma de A#4. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.5 Schéma de A#5. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.6 Schéma de A#6. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.7 Schéma de l'approche X. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4.1 Schéma conceptuel de l'architecture candidate pour l'approche EMI. . . . . . . 73
4.2 Mise en œuvre de l'architecture candidate pour l'analyse de modèles. . . . . . 76
4.3 Mise en œuvre de l'architecture candidate pour l'exécution embarquée de modèles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

5.1 Présentation des actions de débogage disponibles. . . . . . . . . . . . . . . . . 86
5.2 Architecture utilisée pour la vérification formelle. . . . . . . . . . . . . . . . . . 89

5.3 Architecture utilisée pour le *model-checking*. . . . . . . . . . . . . . . . . . . . . 91
5.4 Architecture utilisée pour le *monitoring*. . . . . . . . . . . . . . . . . . . . . . . . . 93
5.5 Architecture utilisée pour le *model-checking* avec des propriétés LTL. . . . . . . 98

6.1 Architecture pour l'ordonnancement de l'exécution réelle. . . . . . . . . . . . . 107
6.2 Architecture de *model-checking* avec filtrage. . . . . . . . . . . . . . . . . . . . . 109
6.3 Architecture de *model-checking* avec ordonnancement. . . . . . . . . . . . . . 111
6.4 Architecture de *model-checking* avec l'ordonnanceur dans la boucle de vérification et un environnement découplé. . . . . . . . . . . . . . . . . . . . . . . . . 113

7.1 Positionnement du langage d'observation par rapport au langage d'action d'un interpréteur EMI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
7.2 Architecture logicielle pour l'analyse de modèles avec un interpréteur EMI. . . . 124

8.1 Méthodologie de conception d'un interpréteur EMI. . . . . . . . . . . . . . . . . 128
8.2 Métamodélisation des données d'exécution d'un langage. . . . . . . . . . . . . 130
8.3 Architecture de modélisation utilisée pour construire un modèle objet. . . . . . 132
8.4 Exemple simplifié de métamodèle$_{ext}$ RTD pour le langage UML. . . . . . . . . . 133

9.1 Exemple d'une machine à états de PUSM représentant un automate de Büchi. 144
9.2 Exemple des machines à états de PUSMs représentant des automates observateurs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
9.3 Composant d'exécution de modèles avec composition synchrone. . . . . . . . 149
9.4 Architecture utilisée pour le *model-checking* avec des PUSMs. . . . . . . . . . 151
9.5 Architecture utilisée pour le *monitoring* avec des PUSMs. . . . . . . . . . . . . 153

10.1 Schéma d'un passage à niveau. . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
10.2 Interface graphique de simulation de OBP2. . . . . . . . . . . . . . . . . . . . . 164
10.3 Diagramme MSC montrant la fermeture du passage à niveau. . . . . . . . . . . 165
10.4 Interface graphique de débogage de OBP2. . . . . . . . . . . . . . . . . . . . . 166
10.5 Espace d'état du contrôleur de passage à niveau avec *FifoEventPool*. . . . . . 167
10.6 Contre-exemple obtenu pour la propriété $E4_{CPN}$ avec *FifoEventPool*. . . . . . . 168
10.7 *Timing diagram* d'une exécution du contrôleur de passage à niveau avec un ordonnancement à priorité fixe et l'hypothèse de réactivité pour l'implémentation *FifoEventPool*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171
10.8 Machines à états des PUSMs (représentant des automates de Büchi) pour le modèle d'IRV. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
10.9 Combinaison de plusieurs PUSMs en un pour vérifier $E1_{IRV}$, $E2_{IRV}$ et $E3_{IRV}$ simultanément. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173

10.10 Machines à états des PUSMs (représentant des automates observateurs) pour le modèle d'IRV. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
10.11 Extrait de la machine à états de l'objet *controller* de l'IRV. . . . . . . . . . . . 175
10.12 Exemple de dessin pour le programme LOGO ci-dessus. . . . . . . . . . . . . . 182
10.13 Capture d'écran de l'interface AnimUML montrant un diagramme de séquence interactif pour le modèle du contrôleur de passage à niveau. . . . . . . . . . . . 183
10.14 Capture d'écran d'une partie de l'interface de Gemoc Studio montrant une session de débogage pour le modèle du contrôleur de passage à niveau. . . . . . 184

A.1 Les deux relations de base en IDM. . . . . . . . . . . . . . . . . . . . . . . . . . 231
A.2 L'architecture en 4 couches du MDA. . . . . . . . . . . . . . . . . . . . . . . . . 232
A.3 Principe d'une transformation de modèles en IDM. . . . . . . . . . . . . . . . . 234
A.4 Vue d'ensemble des concepts d'UML. . . . . . . . . . . . . . . . . . . . . . . . . 236

C.1 Métamodèle des références pour UML. . . . . . . . . . . . . . . . . . . . . . . . 253
C.2 Métamodèle des opérateurs supplémentaires fournis par le langage d'expression pour UML. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 253
C.3 Métamodèle des opérateurs supplémentaires fournis par le langage d'effet pour UML. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 254
C.4 Métamodèle des opérateurs supplémentaires fournis par l'extension du langage d'expression pour UML. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 255
C.5 Métamodèle des opérateurs `EventPoolExp` fournis par l'extension du langage d'expression pour UML. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 256

D.1 Diagramme de structure composite du modèle de CPN. . . . . . . . . . . . . . 257
D.2 Machine à états du système du modèle de CPN. . . . . . . . . . . . . . . . . . . 258
D.3 Machine à états de l'environnement du CPN. . . . . . . . . . . . . . . . . . . . . 259
D.4 Diagramme de composants d'un régulateur de vitesse. . . . . . . . . . . . . . . 260
D.5 Diagramme de structure composite du modèle d'IRV. . . . . . . . . . . . . . . . 261
D.6 Machines à états de l'environnement de l'IRV. . . . . . . . . . . . . . . . . . . . 263
D.7 Machines à états du modèle d'IRV. . . . . . . . . . . . . . . . . . . . . . . . . . 265

# Liste des tableaux

1.1 Synthèse des différentes approches pour l'exécution de modèles. . . . . . . . . 30

2.1 Synthèse des approches d'analyse de modèles. . . . . . . . . . . . . . . . . . . 51

3.1 Positionnement des approches par rapport aux techniques d'analyse et d'exécution de modèles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

3.2 Récapitulatif des problèmes de chaque approche (✗ signifie que le problème est présent et ✓ qu'il est absent). . . . . . . . . . . . . . . . . . . . . . . . . . . 64

4.1 Présentation des différents modes d'exécution. . . . . . . . . . . . . . . . . . . 78

10.1 Résultats du *model-checking* avec différentes implémentations de l'event pool (avec C : le nombre de configurations, A : le nombre de transitions, D : le nombre de *deadlocks*, et les résultats de vérification des propriétés). . . . . . . . . . . 170

10.2 Comparaison des performances d'exécution. . . . . . . . . . . . . . . . . . . . 179

F.1 Liens des pages web archivées. . . . . . . . . . . . . . . . . . . . . . . . . . . 271

# Liste des listings


1.1 Fonction $add$ en assembleur Thumb obtenue grâce à une sémantique translationnelle. . . . . 20
1.2 Sémantique axiomatique de la fonction $add$ avec le formalisme ACSL. . . . . 21
4.1 Interface STR. . . . . 75
5.1 Interface P. . . . . 83
5.2 Interface APE . . . . . 84
5.3 Définition des types utilisés en débogage multivers. . . . . 85
5.4 Définition du débogage multivers interactif. . . . . 86
5.5 Interface APC. . . . . 89
5.6 Interface $A_{cc}$. . . . . 89
5.7 Interface TR. . . . . 91
5.8 Conversion de l'interface STR en TR. . . . . 92
5.9 Opérateur de composition synchrone. . . . . 94
6.1 Définition de la politique de filtrage. . . . . 102
6.2 Opérateur de filtrage. . . . . 103
6.3 Définition de la politique d'ordonnancement. . . . . 104
6.4 Opérateur d'ordonnancement. . . . . 105
6.5 Opérateur de composition asynchrone. . . . . 105
6.6 Pseudo-code de la boucle d'exécution principale. . . . . 107
6.7 Architecture pour l'exécution réelle. . . . . 108
6.8 Architecture de *model-checking* avec filtrage. . . . . 110
6.9 Architecture de *model-checking* avec ordonnancement. . . . . 111
6.10 Architecture de *model-checking* avec découplage de l'environnement. . . . . 113
7.1 Définition du type `EMI` et de la monade `EMIState`. . . . . 118
7.2 Fonctions de l'API de EMI. . . . . 118
7.3 Conversion de l'interface EMI en STR. . . . . 118
9.1 Exemple de code tUML généré à partir d'une propriété LTL. . . . . 147
9.2 Opérateur ajoutant des transitions implicites. . . . . 149
10.1 Exemple de programme LOGO. . . . . 181
B.1 Code Lean des définitions formelles de ce manuscrit. . . . . 242

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA BRETAGNE

**Titre : EMI : Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable**

**Mots clés :** Interprétation, Vérification formelle, Ingénierie dirigée par les modèles, Systèmes embarqués.

**Résumé :** La complexité croissante des systèmes embarqués les expose à davantage de bogues logiciels, d'erreurs de conception et de failles de sécurité. Les besoins en vérification et en validation sont donc de plus en plus importants. Pour exécuter et analyser des modèles de ces systèmes, des transformations sont généralement nécessaires pour obtenir *(i)* le code exécutable pouvant être déployé sur une cible embarquée et *(ii)* des modèles d'analyse permettant d'appliquer des techniques de vérification formelle (p. ex. de *model-checking*). Cependant, ces transformations typiquement non-prouvées sont à l'origine de fossés sémantiques et nécessitent d'établir une relation d'équivalence entre le code exécutable et les modèles d'analyse afin de garantir que ce qui est exécuté est bien ce qui a été vérifié. Pour unifier les activités d'analyse et l'exécution embarquée de modèles, l'approche EMI repose sur un interpréteur de modèles pilotable permettant d'utiliser un unique couple (modèle + sémantique) pour toutes les activités de développement logiciel. Pour évaluer cette approche, un interpréteur de modèles UML a été conçu et appliqué à différents cas d'études de systèmes embarqués afin de mettre en œuvre diverses activités d'analyse (p. ex. simulation, animation, débogage, *model-checking*, *monitoring*).

**Title: EMI: An approach to unify analysis and embedded execution with a controllable model interpreter**

**Keywords:** Interpretation, Formal verification, Model-driven engineering, Embedded systems.

**Abstract:** The increasing complexity of embedded systems renders them more vulnerable to software bugs, design errors and security flaws. Therefore, there is a growing need for verification and validation activities. To execute and analyze models of these systems, transformations are usually required to obtain *(i)* the executable code that can be deployed on embedded targets as well as *(ii)* analysis models used to apply formal verification techniques (e.g., model-checking). However, these typically unproven transformations are responsible for semantic gaps and require to establish an equivalence relation between executable code and analysis models to ensure that what is executed is what has been verified. To unify analysis activities and embedded execution of these models, the EMI approach relies on a controllable model interpreter that uses a unique pair (model + semantics) for all software development activities. To evaluate this approach, a UML model interpreter has been designed and applied on different case studies of embedded systems to perform multiple analysis activities (e.g., simulation, animation, debugging, model-checking, monitoring).