



HAL
open science

Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande : application aux architectures SCADA

Soraya Mesli Kesraoui

► **To cite this version:**

Soraya Mesli Kesraoui. Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande : application aux architectures SCADA. Automatique / Robotique. Université de Bretagne Sud, 2017. Français. NNT : 2017LORIS442 . tel-01738049

HAL Id: tel-01738049

<https://theses.hal.science/tel-01738049v1>

Submitted on 20 Mar 2018

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.



THESE / UNIVERSITE BRETAGNE SUD

UFR Sciences et Sciences de l'Ingénieur

sous le sceau de l'Université Bretagne Loire

Pour obtenir le titre de :

DOCTEUR DE L'UNIVERSITE BRETAGNE-SUD

Mention : STIC

Ecole Doctorale SICMA

présentée par

Soraya Mesli Kesraoui

Préparée à l'unité mixte de recherche :

Lab-STICC (UMR 6285), IRISA (UMR 6074),
CRIStAL (UMR 9189)

Établissement de rattachement :

Université Bretagne Loire

Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande : Application aux architectures SCADA

Thèse soutenue le 11 mai 2017

devant la commission d'examen composée de :

M. Yamine Ait Ameur

Professeur, IRIT (UMR 5505), École Nationale Supérieure d'électronique,
d'électrotechnique, d'Informatique, d'Hydraulique et de Télécommunications / Président

M. Christophe Kolski

Professeur, LAMIH (UMR 8201), Université de Valenciennes et du Hainaut-Cambrésis /
Rapporteur

M. Olivier H. Roux

Professeur, IRCCyN (UMR 6597), École Centrale de Nantes / Rapporteur

Mme. Pascale Marangé

Maître de Conférences, CRAN (UMR 7039), Université de Lorraine / Examineur

M. Pascal Berruet

Professeur, Lab-STICC (UMR 6285), Université Bretagne Sud, Lorient / Directeur de thèse

M. Armand Toguyeni

Professeur, CRIStAL (UMR 9189), École centrale de Lille / Co-directeur

M. Flavio Oquendo

Professeur, IRISA (UMR 6074), Université Bretagne Sud, Vannes / Co-directeur

M. Alain Bignon

Docteur, Responsable R&I, Segula Technologies, Lanester / Responsable industriel CIFRE

*"Je dédie cette thèse
à mes chers parents."*

Remerciements

Ho que c'est difficile d'écrire et de trouver les mots justes pour exprimer mes remerciements et ma gratitude envers celles et ceux qui m'ont soutenu durant ces trois années de thèse.

Je remercie tout d'abord Alain BIGNON qui a cru en moi et qui m'a proposé ce sujet de thèse. Je te remercie aussi pour tes conseils, pour ton encadrement et surtout pour ton soutien jusqu'à la dernière minute. Tu resteras toujours "Mon meilleur chef!".

Je remercie également messieurs Pascal BERRUET, Armand TOGUYENI et Flavio OQUENDO d'avoir accepté d'encadrer cette thèse. Travailler sous la direction de trois professeurs avec des contraintes de distance n'est pas toujours évident, mais vous avez su rendre ce travail agréable. Je vous remercie surtout pour l'intérêt que vous avez porté à mes travaux, pour le suivi, les conseils, votre disponibilité et aussi pour la bonne ambiance qui a rythmé nos réunions.

Je souhaite remercier messieurs Christophe KOLSKI et Olivier H. ROUX d'avoir accepté de relire ces travaux. Je remercie également monsieur Yamine AIT AMEUR d'avoir accepté d'examiner et de présider le jury de cette thèse. Je remercie aussi madame Pascale MARANGE d'avoir bien voulu accepter la charge d'examineur.

Je tiens aussi à remercier le personnel des laboratoires lab-STICC et L'IRISA pour leur professionnalisme. Je n'oublie pas de remercier les doctorants du Lab-STICC (Amandine PORCHER, Fanny GUENNOG, Rani KHAN, Thomas TOUBLANC) et de l'IRISA pour les discussions très enrichissantes qu'on a pu avoir au cours de ces années de thèse.

Je remercie également et chaleureusement mes collègues de Segula Technologies, surtout l'équipe du midi : Perine Le SENECHAL, Éric LE BRIS, Sophie PRAT, Laurianne BOULHIC, Olga GOUBALI, Nicolas AUFFRET, Davy RODIER et Fabien SILONE. Merci à Perine, Olga et Laurianne d'avoir relu et corrigé les chapitres de cette thèse. Merci à Eric d'avoir accepté de participer à mes multiples entretiens. Merci à vous tous pour les bons moments qu'on a pu partager. Je remercie aussi Raoul DJOUSSE, Landry RAHAMEFY et Franck NGANKAM d'avoir participé et contribué à mes travaux.

Je tiens spécialement à remercier mes amies depuis toujours : Sonia, Farisa, Dihia, Nassima, Loulou et Kahina. Merci pour votre soutien et votre amitié. Cette thèse m'a aussi permis de rencontrer des personnes formidables comme Olga GOUBALI et Asma BENMESSAOUD GABIS. Je te remercie Olga pour ton soutien et d'avoir toujours cru en moi. Tu as toujours su me communiquer ta joie de vivre et ta bonne humeur dans les moments difficiles. Je te remercie Asma pour les bons moments passés ensemble et pour toutes les discussions imaginables et inimaginables qu'on a pu avoir.

Je remercie spécialement mes parents qui ont toujours été là pour moi et qui ont toujours cru en moi. Je remercie aussi mes frères et sœurs : Ouissem, Sabrina, Toufik et Cherif. Je tiens aussi à remercier mes beaux-parents de m'avoir considéré comme leur fille et de m'avoir toujours soutenue. Je remercie aussi Salim, Djamila et Zazi.

Je n'oublie pas de remercier la personne qui m'a supporté durant ces trois années de thèse. Je te remercie Djamel d'être toujours à mes côtés et d'avoir supporté mes nombreuses crises de stress et d'anxiété. Je suis convaincu que sans toi, cette thèse n'aurait pas eu lieu. Merci !

Je remercie tous ceux qui ont contribué de près ou de loin à cette thèse.

Cette thèse est le fruit d'une collaboration entre l'entreprise SEGULA Technologies, acteur majeur des métiers de l'ingénierie, et trois laboratoires de recherches : Le Lab-STICC, l'IRISA et CRIStAL. Le Lab-STICC est un pôle de référence en recherche sur les systèmes communicants. L'IRISA de Vannes est un pôle de référence en recherche sur les architectures logicielles, l'analyse et le traitement d'images, la fouille de données, l'interaction gestuelle, l'informatique mobile et décisionnelle. CriStAL est un pôle de référence en recherche sur l'ingénierie des données et de modèles, les systèmes embarqués temps réels, le traitement d'image, le génie logiciel et la conception système.



Segula Engineering France
165 rue de la Montagne du Salut
BP 50256 - 56602 Lanester Cedex
Tel : +33 (0)2 97 87 73 07
www.segulatechnologies.com



Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance (Lab-STICC) de Lorient, UMR 6285
Centre de Recherche Christiaan Huygens
Rue de Saint Maudé 56321 Lorient Cedex
Tel : +33 (0)2 97 87 45 60
www.lab-sticc.fr



Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) de Vannes, UMR 6074
Campus de Tohannic
Bâtiment ENSIBS, rue Yves Mainguy
BP 57356017 Vannes cedex
Tel : +33 (0)2 97 01 72 35
www-irisa.univ-ubs.fr



Centre de Recherche en Informatique, Signal et Automatique (CRIStAL) de Lille, UMR 9189
Université Lille 1
Bâtiment M3 extension, Avenue Carl Gauss
59655 Villeneuve d'Ascq Cedex
www.cristal.univ-lille.fr

Sommaire

Sommaire	v
Table des figures	ix
Liste des tableaux	xi
Glossaire	xiii
Contributions scientifiques	1
Introduction générale	1
I Contexte et état de l’art	7
1 Contexte et problématiques	9
1.1 Introduction	10
1.2 L’ingénierie système : contexte académique	10
1.2.1 La spécification des exigences	11
1.2.2 La vérification des exigences	15
1.3 Conception des systèmes de contrôle-commande : contexte industriel	18
1.3.1 Les systèmes de contrôle-commande	18
1.3.2 L’architecture SCADA	19
1.3.3 Approches traditionnelles pour la conception des systèmes de contrôle- commande	20
1.3.4 Génération automatique des programmes de contrôle-commande	23
1.4 Problématiques et verrous scientifiques	29
1.4.1 Problématiques	29
1.4.2 Verrous scientifiques	30
1.5 Conclusion	31
2 État de l’art : <i>systematic mapping</i>	33
2.1 Introduction	34
2.2 État de l’art au travers des revues de la littérature existante	34
2.2.1 Classification des langages formels	34

2.2.2	Comparaison des langages de spécification	36
2.2.3	Utilisation des méthodes formelles	36
2.2.4	Bilan sur les revues de la littérature existantes	37
2.3	Méthode	37
2.4	Étape 1 : Définition du protocole	38
2.4.1	Définition des questions de recherche	38
2.4.2	Stratégie de la recherche	39
2.4.3	Sélection des articles	40
2.4.4	Extraction des données et classification	40
2.4.5	Évaluation de la validité de l'étude	41
2.5	Étape 2 : Conduction	42
2.6	Étape 3 : Rapport de synthèse	44
2.6.1	Résultats démographiques	44
2.6.2	Proposition d'une classification des langages formels	45
2.7	Résultats obtenus	48
2.7.1	QR1 : Quels sont les langages de spécification utilisés pour la vérification formelle ?	48
2.7.2	QR2 : Comment a été réalisée la vérification formelle ?	52
2.7.3	QR3 : Quels sont les objectifs de la vérification formelle ?	54
2.8	Conclusion et Discussions	58
2.8.1	Bilan	58
2.8.2	Propositions	58
 II Approches de vérification formelle : propositions méthodologiques		61
 3 Vérification formelle d'une chaîne de contrôle-commande élémentaire		63
3.1	Introduction	64
3.2	Exemple illustratif : V2VM	64
3.2.1	Comportement de l'utilisateur	65
3.2.2	L'interface de supervision	65
3.2.3	Le programme de commande	66
3.2.4	La partie opérative	66
3.2.5	Les exigences de conception	67
3.3	Contexte et travaux connexes	67
3.3.1	Vérification formelle des programmes de commande	67
3.3.2	Vérification formelle des interfaces de supervision	68
3.3.3	Manques dans les travaux existants et problématique associée	69
3.4	Vérification formelle d'une chaîne de contrôle-commande élémentaire	69
3.4.1	Concepts utilisés	70

3.4.2	Approche générale	72
3.4.3	Formalisation d'une chaîne de contrôle-commande élémentaire	74
3.4.4	Vérification formelle d'une chaîne de contrôle-commande élémentaire	85
3.5	Conclusion	85
4	Vérification formelle des modèles de conception (P&ID)	87
4.1	Introduction	88
4.2	Exemple illustratif : P&ID	88
4.3	Contexte et travaux connexes	89
4.3.1	Vérification des P&IDs	89
4.3.2	Les architectures logicielles	90
4.4	Approche proposée	91
4.4.1	Concepts utilisés	91
4.4.2	Vérification formelle des diagrammes P&ID	93
4.4.3	Un style architectural pour la norme ANSI/ISA-5.1	93
4.4.4	Génération des modèles Alloy à partir des P&ID	98
4.4.5	Formalisation des exigences et vérification formelles des P&ID	99
4.4.6	Aide à l'analyse des résultats	100
4.5	Conclusion	104
III	Approches de vérification formelle : mise en œuvre et applications	107
5	Implémentation	109
5.1	Introduction	110
5.2	Méthodologie mise en œuvre	110
5.2.1	Concepts IDM utilisés	111
5.2.2	Outils et langages utilisés	112
5.3	Flot de vérification des composants standards	115
5.3.1	Opération de modélisation de la tâche utilisateur	116
5.3.2	Opération de transformation de HAMSTERS en AT intermédiaire (AT')	117
5.3.3	Opération de transformation des IHM SCADA en AT'	122
5.3.4	Opération de transformation des programmes LD en AT'	126
5.3.5	Opération de transformation de AT' en AT	134
5.3.6	Opération de modélisation du composant physique en AT	136
5.3.7	Opération de spécification des exigences en CTL	137
5.3.8	Opération de vérification formelle	137
5.3.9	Bilan	138
5.4	Flot de vérification formelle des diagrammes P&ID	138
5.4.1	Opération de formalisation de la norme ANSI/ISA-5.1	139
5.4.2	Opération de construction	140

5.4.3	Opération de spécification des exigences	142
5.4.4	Opération d'épuration	143
5.4.5	Opération de transformation de P&ID en Alloy	144
5.4.6	Opération de vérification	147
5.4.7	Opération de visualisation des erreurs	148
5.4.8	Opération de correction	151
5.4.9	Bilan	151
5.5	Conclusion	152
6	Application à des cas industriels	153
6.1	Introduction	154
6.2	Vérification formelle de V2VM : Étude de cas	154
6.2.1	Opération de modélisation de la tâche utilisateur	154
6.2.2	Opération de transformation de HAMSTERS en AT	155
6.2.3	Opération de transformation des IHM SCADA en AT	156
6.2.4	Opération de transformation des programmes LD en AT	158
6.2.5	Opération de modélisation du composant physique en AT	159
6.2.6	Opération de spécification des exigences en CTL	161
6.2.7	Vérification formelle de V2VM	161
6.2.8	Validité	163
6.2.9	Bilan de la vérification formelle de la V2VM	164
6.3	Vérification formelle du P&ID du système EdS : Etude de cas	164
6.3.1	Opération de construction	164
6.3.2	Opération de spécification des exigences	165
6.3.3	Opération d'épuration et de transformation du P&ID en Alloy	168
6.3.4	Opération de vérification	168
6.3.5	Opération de visualisation des erreurs	169
6.3.6	Validité	169
6.3.7	Bilan de la vérification formelle du P&ID du système EdS	170
6.4	Conclusion	171
7	Conclusion	173
7.1	Introduction	173
7.2	Rappel des contributions	174
7.2.1	Intégration de la vérification formelle dans une démarche de conception	174
7.2.2	Aide à l'obtention des modèles formels	175
7.2.3	Application à des cas d'étude concrets	176
7.3	Perspectives	176
7.3.1	Extensions des flots proposés	176
7.3.2	Aide à la spécification des exigences	177

7.3.3	Vérification formelle du système global	178
Bibliographie		205
Annexes		209
A	Données de la Systematic Mapping	209
A.1	Requêtes de recherche dans les différentes bases	209
A.2	Listes des articles retenus pour la <i>systematic mapping</i>	210
B	Guide d'entretien réalisé avec les experts métiers	215
B.1	Présentation	215
B.2	Analyse des exigences	215
B.3	Analyse de la conception système	216
B.4	Analyse de la vérification	216
B.5	Confrontation avec le P&ID, la commande, et l'interface de supervision	216
C	Modèle complet de V2VM	219
C.1	Tâches utilisateurs	219
C.2	Interface de supervision	220
C.3	Programme de commande	222
C.3.1	Composant physique	223
C.4	Extrait d'un contre-exemple Alloy en XML	223
D	Grammaire VB en xtext	225

Table des figures

1	Organisation de ce manuscrit	3
1.1	Représentation d'un système de contrôle-commande	18
1.2	L'architecture SCADA	19
1.3	Architecture simpliste d'un système interactif	20
1.4	Cycle opérationnel d'un API	21
1.5	Le projet Anaxagore	27
1.6	Structure d'un composant standard dans Anaxagore	28
1.7	Flot de conception Anaxagore	29
2.1	Classifications existantes des langages formels	35
2.2	Le processus de la revue systématique	38
2.3	Évolution de la sélection des articles pertinents	43
2.4	Provenance des articles	44
2.5	Nombre d'articles par an	44
2.6	Nuage des mots clés	45
2.7	Classification des langages formels	46
2.8	Type des langage de spécification des systèmes	48
2.9	Langages basés sur les modèles	49
2.10	Langages basés sur les propriétés	49
2.11	Langages semi-formels	50
2.12	Types des langages formels	50
2.13	Classes des langages formels	50
2.14	Type des langages de spécification des propriétés	51
2.15	Les <i>Model-Checkers</i>	52
2.16	Les theorem provers	52
2.17	Les <i>Model-Checkers</i> à travers le temps	53
2.18	Les <i>Theorem-Provers</i> à travers le temps	53
2.19	Les propriétés vérifiées par chaque technique formelle	54
2.20	Le domaine de la vérification	55
2.21	Les propriétés vérifiées dans chaque facette du système	56
2.22	Utilisation des méthodes formelles dans l'industrie et dans l'académie	56
2.23	L'objectif général de l'utilisation des méthodes formelles	57
3.1	Les différentes vues de la V2VM	65
3.2	Exemple d'automate temporisé dans UPPAAL	71
3.3	Approche de vérification formelle des composants standards	73
3.4	Extrait de la tâche <i>Ouvrir la vanne</i> dans la notation Hamsters	74
3.5	Automate temporisé correspondant au modèle de tâches de la figure 3.4	76
3.6	Modélisation de l'IHM , Niveau exploitation : Objet de commande	78

3.7	Modélisation de l'IHM , Niveau exploitation : Objet informationnel	78
3.8	Modélisation de l'IHM , Les fonctions prédéfinies : a) Automate temporisé des fonctions, b) Code C des fonctions	79
3.9	Exemple d'un programme LD	80
3.10	Modélisation des temporisateurs ; (a) TON ; (b) TOF ;	80
3.11	Automate initial	81
3.12	Communication Automate principal et temporisateur	82
3.13	Modélisation du programme LD de l'exemple 3.9	82
3.14	Modélisation du comportement d'une vanne	83
4.1	Exemple de diagramme P&ID	88
4.2	Approche générale de vérification des P&ID	93
4.3	Vanne trois voies motorisée (V3VM)	95
4.4	Les connecteurs dans la norme ANSI/ISA-5.1	96
4.5	La vérification de la cohérence du style : (a) L'instance retournée par Alloy ; (b) Le P&ID correspondant	98
4.6	Extrait d'un contre-exemple retourné par l'outil Alloy	100
4.7	Data State Reference Model (DSRM) de Chi [2000]	103
4.8	Outil de vérification et de visualisation des erreurs	105
5.1	Méthodologie mise en œuvre dans le projet de recherche	111
5.2	Outils	113
5.3	Flot de vérification des composants standards	115
5.4	Modélisation de la tâche utilisateur	116
5.5	Extrait de modèle de tâches HAMSTERS	117
5.6	Transformation de HAMSTERS en AT'	117
5.7	Méta-modèle HAMSTERS	118
5.8	Méta-modèle UPPAAL intermédiaire [Gerking 2013]	120
5.9	Transformation des IHM SCADA en AT'	122
5.10	Extrait du Méta-modèle Panorama E2	124
5.11	Formalisation des fonctions prédéfinies dans Panorama E2 en : A) Automate temporisé ; B) Fonctions C	125
5.12	Transformation des codes VB en C	125
5.13	Transformation des programme LD en AT'	126
5.14	Extrait d'un modèle Straton (fichier XML)	127
5.15	Méta-modèle Straton	128
5.16	Méta-modèle Beremiz	129
5.17	Règle ATL de transformation des éléments en parallèles	133
5.18	Transformation de AT' en AT	134
5.19	Méta-modèle XML	135

5.20	Extrait d'un modèle XML	135
5.21	Opération de modélisation du composant physique en AT	136
5.22	Opération de spécification des exigences en CTL	137
5.23	Vérification formelle par UPPAAL	138
5.24	Flot de vérification des diagrammes P&ID	139
5.25	Opération de formalisation de la norme ANSI/ISA	140
5.26	Opération de construction	140
5.27	Exemple d'un P&ID saisi sous Visio à partir d'une bibliothèque	141
5.28	Opération de spécification des exigences	142
5.29	Opération d'épuration	143
5.30	Extrait du modèle synoptique correspondant au P&ID de la figure 5.27	144
5.31	Opération de transformation des diagrammes P&ID en Alloy	144
5.32	Le méta-modèle des diagrammes P&ID [Bignon 2012]	145
5.33	Le métamodèle Alloy [Garis et al. 2012]	146
5.34	Opération de vérification formelle	148
5.35	Opération de visualisation des erreurs sur le P&ID	148
5.36	Flot de visualisation des contre-exemples Alloy sur les diagrammes P&ID	149
5.37	Méta-modèle pour la visualisation des erreurs	150
5.38	Opération de correction	151
6.1	La tâche <i>Ouvrir la vanne</i> en HAMSTERS	155
6.2	Automate temporisé généré à partir du modèle HAMSTERS de la figure 6.1	156
6.3	Implémentation de la V2VM dans Anaxagore	157
6.4	Automate temporisé généré à partir du bouton Ouverture	157
6.5	Programme LD de la V2VM	158
6.6	Automate temporisé généré automatiquement à partir du programme LD de la figure 6.5	159
6.7	Modélisation du composant matériel : (a) Capteur fin de course-fermeture ; (b) Capteur fin de course-ouverture ; (c) La vanne	160
6.8	Simulation du contre-exemple dans Straton	162
6.9	Un extrait du diagramme P&ID du système EdS	165
6.10	Visualisation du contre-exemple	169
C.1	Extrait du contre-exemple en XML	223

Liste des tableaux

2.1	Fiche de relecture pour l'extraction de données	40
2.2	Provenance des articles et sélections	43
2.3	Citations	45
2.4	Les propriétés vérifiées par domaines d'application	55
3.1	Les différentes variables booléennes de la V2VM	66
3.2	Correspondances des opérateurs entre CTL et UPPAAL	72
3.3	Transformation des tâches en signaux	75
3.4	Transformation des opérateurs de tâches en automates temporisés	76
4.1	Visualisation des contre-exemples Alloy sur le P&ID	104
5.1	Tableau récapitulatif des différences entre Straton et Beremiz	130
5.2	Règles de transformations des éléments Ladder en UPPAAL	131
6.1	Composition du modèle complet de la V2VM	162
6.2	Récapitulatif de la vérification formelle du P&ID du système EdS	170
6.3	Temps de génération des modèles UPPAAL pour les composants de la bibliothèque Anaxagore	171
A.1	Requêtes de recherche pour les différentes bases	209
A.2	Articles retenus pour l'étude	210

Glossaire

- ADL** : Architecture Description Languages. 90
- API** : Automate Programmable Industriel. 19–22, 67, 79, 81, 113, 126, 130
- AT** : Automates temporisés. 70, 80, 86, 161
- ATL** : Atlas Transformation Language. 134, 135
- CTL** : Computation Tree Logic. ix, 13, 48, 51, 63, 68, 70, 71, 73, 74, 85, 137, 164
- EdS** : Eau douce Sanitaire. iv, ix, xi, 153, 154, 164–171, 176
- IDE** : Integrated Development Environment. 113
- IDM** : Ingénierie Dirigée par les Modèles. 26, 27, 68, 69, 93, 111, 112, 114, 152, 175
- IHM** : Interface Homme-Machine. iii, iv, vii, viii, 19–25, 27, 29, 30, 64, 65, 67–69, 73, 74, 76, 78, 79, 81, 83, 84, 86, 109, 110, 112, 114–116, 122, 123, 125, 126, 137, 153, 154, 156, 161, 164, 174, 176, 177
- LD** : Ladder Diagram. iii, iv, viii, 22, 28, 68, 73, 74, 79–82, 86, 109, 110, 112, 126–130, 153, 158, 162, 175
- LTL** : Linear Temporal Logic. 48, 51, 68
- P&ID** : Piping and Instrumentation diagram. iii–v, viii, ix, xi, 4, 5, 26–29, 59, 87–94, 96–106, 109, 110, 114, 138–154, 164–172, 175–178, 216, 217
- SCADA** : Supervisory Control And Data Acquisition. 19, 20, 24, 76, 77, 79
- UML** : Unified Modeling Language. 14, 24, 49–51
- XML** : eXtensible Markup Language. 102, 112, 114, 116, 118, 119, 126, 127, 130, 134, 135, 150, 151, 168

Contributions scientifiques

Conférences internationales avec comité de lecture :

- 1. Soraya MESLI-KESRAOUI, Armand TOGUYENI, Alain BIGNON, Flavio OQUENDO, Djamal KESRAOUI, Pascal BERRUET :** Formal and Joint Verification of Control Programs and Supervision Interfaces for Socio-technical Systems Components. In : *13th IFAC Symposium on Analysis, Design, and Evaluation of Human-Machine Systems HMS 2016 Kyoto, Japan, 30 August – 2 September 2016* 49 (2016), Nr. 19, p. 426–431
- 2. Soraya MESLI-KESRAOUI, Alain BIGNON, Djamal KESRAOUI, Armand TOGUYENI, Flavio OQUENDO, Pascal BERRUET :** Vérification Formelle de Chaînes de Contrôle-Commande d'Éléments de Conception Standardisés. In : *MOSIM 2016, 11ème Conférence Francophone de Modélisation, Optimisation et Simulation, Montréal, Québec, Canada, 22 – 24 Août, 2016*.
- 3. Soraya MESLI-KESRAOUI, Djamal KESRAOUI, Flavio OQUENDO, Alain BIGNON, Armand TOGUYENI, Pascal BERRUET :** *Formal Verification of Software-Intensive Systems Architectures Described with Piping and Instrumentation Diagrams*. p. 210–226. In : TEKINERDOGAN, Bedir (Editor) ; ZDUN, Uwe (Editor) ; BABAR, Ali (Editor) : *Software Architecture : 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 – December 2, 2016, Proceedings*. Cham : Springer International Publishing, 2016. – URL http://dx.doi.org/10.1007/978-3-319-48992-6_16. – ISBN 978-3-319-48992-6

Introduction générale

La conception des systèmes complexes fait intervenir des concepteurs provenant d'horizons techniques très variés. Les systèmes de contrôle-commande, en particulier, parce qu'ils nécessitent des connaissances avancées concernant le système piloté, le système pilotant et l'utilisateur, illustrent cet état de fait. La communication entre les concepteurs associés demeure souvent très compliquée notamment du fait de la variété des langages techniques utilisés. Ces problèmes de communication et d'interprétation de spécifications peuvent conduire à des erreurs de conception et au non respect du cahier des charges. Une étude a montré que 70% des erreurs des produits logiciels trouvent leur origine dans les phases de spécification et de conception [Selby et Selby 2007], et que 72% de ces erreurs sont détectées dans les phases de test ou en opération [Pham 2007]. La détection tardive des erreurs entraîne une explosion des délais ainsi que les coûts de re-conception.

Dans le but de maîtriser la conception des systèmes de contrôle-commande, des démarches de conception plus rigoureuses et de plus en plus automatisées sont adoptées. La démarche de conception ascendante, par exemple, se focalise plus sur la réutilisation du code en adoptant une forte standardisation des composants constitutifs du système. La conception consiste ainsi en l'agrégation de plusieurs composants réutilisables. Bien qu'elle offre une forte réutilisation, cette démarche manque souvent de cohérence entre les composants utilisés et de vision globale du système à concevoir. Á contrario, la démarche, dite descendante, se focalise sur une vision plus générale en se basant sur des modèles de haut niveau. L'idée derrière cette démarche consiste à spécifier le système par un modèle abstrait décrivant un point de vue de conception, qui sera raffiné ou transformé en modèles décrivant d'autres points de vue, garantissant ainsi une certaine cohérence entre les modèles de départ et les modèles générés. Cependant, le raffinement et la transformation peuvent engendrer la propagation d'erreurs vers les modèles raffinés ou transformés. De plus, ces modèles manquent de modularité et sont donc difficilement réutilisables. Á l'issue de ces deux démarches, la démarche mixte a émergé. Elle combine les deux premières démarches pour garantir, d'une part, la modularité et la réutilisation du code par l'utilisation d'un ensemble de composants prédéfinis "sur étagère" et, d'autre part, l'intégration d'une cohérence et d'une vision globale offertes par les modèles. En pratique, elle consiste donc en la construction d'un modèle structurel en s'appuyant sur des composants standards du domaine. Ensuite les modèles de contrôle-commande sont générés par intégration des modèles de contrôle-commande de chaque composant apparaissant dans le modèle structurel du système.

Cependant, la démarche mixte hérite des deux démarches, ascendante et descendante, de leurs avantages et aussi de leurs inconvénients. Les composants prédéfinis doivent être libres de tout défaut afin de garantir la qualité des systèmes conçus à partir de ces composants. Les modèles quant à eux, doivent intégrer les différentes exigences et recommandations du cahier des charges. Ils doivent être complets, cohérents et corrects afin de garantir la qualité des systèmes obtenus après raffinement de ces modèles.

Contexte

Des études menées précédemment par SEGULA Technologies en collaboration avec le Lab-STICC ont abouti à la définition d'une démarche de conception conjointe de programme de commande et d'interface de supervision [Bignon et al. 2010, Bignon 2012, Bignon et al. 2013, Goubali et al. 2014]. Cette démarche (mixte) s'appuie sur deux piliers fondamentaux, le premier est la standardisation des composants de conception et le deuxième l'utilisation de modèles métier maîtrisés par les experts pour la spécification de leur système. Les modèles métier sont obtenus par une approche ascendante d'agrégation des éléments de conception. Les programmes de commande et les interfaces de supervision sont obtenus par une approche descendante de raffinages successifs des modèles métiers mettant en œuvre les concepts de l'ingénierie dirigée par les modèles. Bien qu'assurant déjà une bonne cohérence, la démarche proposée ne permet pas la vérification au plus tôt des programmes générés. Nos travaux, s'inscrivant dans la continuité de ces résultats, étudient les apports de différentes techniques de vérification formelle au sein de la démarche précédemment développée.

Nos travaux, réalisés au cours d'une thèse CIFRE (Convention Industrielle de Formation à la Recherche en Entreprise) entre l'entreprise Segula Technologies et les laboratoires de recherche : LabSTICC, IRISA et CRISAL, se placent dans le contexte de la mise en œuvre de boucles de pilotage pour les systèmes de contrôle-commande. Pour s'inscrire dans la démarche "first time right", il est nécessaire de proposer des méthodes de conception intégrant des aspects de vérification et de rebouclage rapide inhérent à tout processus de conception.

Problématique

Garantir la qualité d'un système revient à vérifier son respect vis-à-vis du cahier des charges, son aptitude à assurer les fonctionnalités demandées et aussi son bon fonctionnement. Les techniques de vérification formelle ont démontré leur efficacité dans la détection des erreurs et la garantie de la qualité des systèmes. Ces techniques basées essentiellement sur des notations mathématiques offrent un moyen robuste pour la détection et la correction des erreurs de conception. Elles sont basées sur une description en langages formels du système, sur-lequel un ensemble de propriétés est vérifié.

Cependant, plusieurs obstacles limitent l'utilisation de ces techniques dans le monde industriel [Bjørner et Havelund 2014]. Le manque d'automatisme, cas du *Theorem-Proving* par exemple, l'explosion combinatoire des états, dans le cas du *Model-Checking*, et la difficulté de la manipulation des notations formelles par des concepteurs métier, restent les plus récurrentes.

Objectifs de la thèse

L'objectif général de nos travaux de thèse consiste à introduire des boucles de vérification formelles dans les processus de conception des systèmes de contrôle-commande complexes,

afin de garantir la qualité des systèmes produits par la détection et l'élimination des erreurs dans les phases de conception.

Nous ambitionnons de répondre à trois objectifs émergeant de notre contexte de travail. Le premier consiste à garantir la qualité des composants prédéfinis utilisés dans la conception des systèmes de contrôle-commande afin de limiter la propagation des erreurs vers le système complet. Le deuxième objectif vise à garantir la qualité des modèles métiers utilisés pour le raffinement. Ces modèles doivent être complets et cohérents. Ils intègrent aussi bien des exigences fonctionnelles que des exigences non-fonctionnelles. Par ailleurs, les notations formelles étant peu accessibles à notre public de concepteurs visé, le troisième objectif de nos travaux porte sur les moyens de faciliter l'obtention des modèles formels dans un contexte industriel.

Structuration du manuscrit

Nous avons proposé, à travers ce travail de recherche, des contributions méthodologiques et aussi techniques pour garantir la qualité des systèmes de contrôle-commande, en utilisant les méthodes formelles. La suite du manuscrit est structurée en trois parties regroupant sept chapitres.

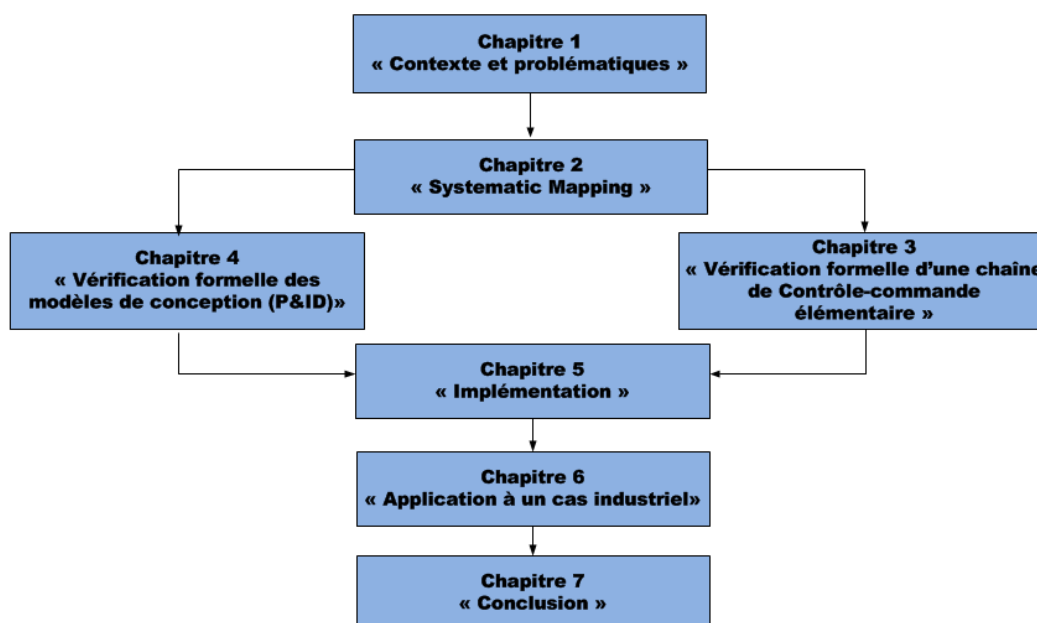


FIGURE 1 – Organisation de ce manuscrit

La première partie, composée des chapitres 1 et 2, présente le contexte de ce travail de thèse et un état de l'art scientifique.

Le chapitre 1 présente le contexte industriel et académique des travaux présentés dans ce manuscrit. Il présente aussi un aperçu des méthodes et démarches de conception des systèmes

de contrôle-commande. Il se termine par une présentation détaillée de la problématique et les verrous scientifiques de nos travaux de thèse.

Le chapitre 2 dresse un état de l'art, relatif à nos travaux, issu d'une revue systématique de la littérature. Nous présentons, dans ce chapitre, le protocole suivi pour l'élaboration de la revue systématique et son application pour l'extraction des articles pertinents. Nous présentons, ensuite, les résultats concernant l'usage des méthodes formelles dans le domaine industriel et académique que nous discutons, à la fin du chapitre.

La deuxième partie présente les différentes contributions méthodologiques. Elle est constituée des chapitres : 3, 4.

Le chapitre 3 présente, en détail, notre approche de vérification d'une chaîne de contrôle-commande élémentaire utilisée dans une démarche de conception ascendante. En effet, nous présentons, à travers un exemple de composant standard, la formalisation des différentes vues de ce composant en automates temporisés. Ensuite, nous listons les différents types de propriétés qu'un composant standard doit vérifier. Enfin, nous écrivons ces propriétés en CTL et nous les vérifions par *Model-Checking* sous UPPAAL.

Le chapitre 4 présente notre approche de vérification formelle des diagrammes P&ID (approche descendante). Dans ce chapitre, nous proposons et nous définissons un style architectural, en Alloy, pour la norme ANSI/ISA-5.1 qui régit la construction de diagrammes P&ID. Nous présentons aussi la formalisation et la génération automatique des modèles Alloy à partir des diagrammes P&ID. Sur ces modèles combinés avec le style architectural, nous vérifions, par *Model-Checking*, la complétude, la cohérence, la compatibilité de ces diagrammes à la norme ANSI/ISA-5.1 [ISA 1992] et aussi leur conformité au cahier des charges i.e. leur satisfaction des exigences fonctionnelles et non-fonctionnelles. Dans le cas où l'outil d'analyse retourne un contre-exemple (non satisfaction d'une exigence), nous proposons aussi une visualisation de celui-ci sur les diagrammes P&ID afin d'assister les concepteurs dans la compréhension et la correction d'erreurs.

La troisième partie de ce manuscrit, aborde nos contributions de mise en œuvre et d'application. Elle est composée des chapitres 5 et 6.

Dans le chapitre 5, nous présentons l'automatisation de nos approches sous la forme de deux flots de vérification semi-automatisés. Le premier flot supporte notre approche de vérification formelle des composants standards. Il porte sur la génération automatique par transformation de modèles, des automates temporisés à partir des différentes vues des composants standards. Le deuxième est dédié à la vérification des diagrammes P&ID et consiste en la génération des modèles Alloy à partir des diagrammes P&ID. Ces deux flots ainsi que les différentes transformations de modèles permettant la génération automatique de modèles formels sont présentés dans ce chapitre.

Dans le chapitre 6, nous appliquons nos deux approches de vérification formelle à des cas industriels concrets. L'approche de vérification des composants standards a été appliquée sur la

vanne deux voies motorisée (V2VM). Par ailleurs, nous appliquons notre deuxième approche sur le P&ID du système de stockage, de production et de distribution d'eau douce embarquée dans un navire (EdS).

Enfin, le chapitre 7 conclut ce manuscrit par un rappel des différentes contributions présentées et les perspectives pour des recherches futures liées à cette thèse.

Première partie

Contexte et état de l'art



Contexte et problématiques

Résumé : Dans ce chapitre, nous présentons et situons le contexte académique et industriel de cette thèse. Le contexte académique porte sur l'ingénierie système, la spécification des exigences, les langages de spécification, la vérification formelle et les méthodes formelles. Le contexte industriel porte sur la conception des systèmes de contrôle-commande complexes. Tout d'abord, nous présentons l'architecture générale d'un système de contrôle-commande, puis nous nous focalisons sur l'architecture SCADA. Ensuite, nous présentons les différentes approches traditionnelles pour la conception des systèmes de contrôle-commande et celles basées sur la génération automatique. Enfin, nous abordons et présentons les problématiques liées à cette thèse et les différents verrous scientifiques qui en découlent.

Sommaire

1.1	Introduction	8
1.2	L'ingénierie système : contexte académique	8
1.2.1	La spécification des exigences	9
1.2.2	La vérification des exigences	13
1.3	Conception des systèmes de contrôle-commande : contexte industriel	16
1.3.1	Les systèmes de contrôle-commande	16
1.3.2	L'architecture SCADA	17
1.3.3	Approches traditionnelles pour la conception des systèmes de contrôle-commande	18
1.3.4	Génération automatique des programmes de contrôle-commande	21
1.4	Problématiques et verrous scientifiques	27
1.4.1	Problématiques	27
1.4.2	Verrous scientifiques	28
1.5	Conclusion	29

1.1 Introduction

Le besoin d'amélioration imposé aux industriels, ainsi que le besoin de rendre les systèmes plus performants, plus sûrs, plus ergonomiques, offrant toujours plus de fonctionnalités aux utilisateurs, ne font qu'augmenter la complexité des systèmes. Cette complexité se traduit par l'augmentation importante de la taille des composants et des fonctionnalités. Elle a pour conséquences de nécessiter des équipes de développement de taille importante et souvent composées d'intervenants appartenant à divers domaines de l'ingénierie.

Une étude menée dans le domaine du génie logiciel, impliquant 14 systèmes complexes et comportant 3418 erreurs, montre que 70% des erreurs sont introduites lors des phases d'analyse (spécification et conception) [Selby et Selby 2007]. Ces résultats sont similaires à ceux de Sourisse et Boudillon [1997], qui trouvent que le pourcentage des erreurs introduites dans les phases d'analyse approche les 79%. Ces erreurs sont détectées généralement tardivement dans le cycle de vie d'un système et leurs corrections font exploser les coûts. D'autres études montrent que 72% des erreurs sont détectées à la livraison [Pham 2007] et que le coût relatif aux corrections augmente considérablement au fil du temps [Boehm et al. 1981]. En effet, plus l'erreur est détectée tard dans le cycle de vie, plus le coût relatif à sa correction est élevé. Les investigations montrent aussi que 95% des erreurs pourraient être détectées et éliminées dans la phase où elles sont introduites, contre 47% dans les phases ultérieures [Selby et Selby 2007]. L'adoption de méthodes qui offrent des outils et techniques pour gérer la complexité d'un système et contrôler sa conception et sa réalisation permettent de diminuer considérablement l'introduction des erreurs dans chaque phase du cycle de développement.

Les grands concepteurs de système complexes, comme la NASA, se sont rendus compte, dès les années 60, de la nécessité de définir des démarches rigoureuses et outillées afin d'assister la conception des systèmes complexes. Ces démarches sont connues sous le nom de *l'Ingénierie système* (IS).

1.2 L'ingénierie système : contexte académique

L'ingénierie système est une démarche dédiée à la réalisation des systèmes complexes. Son but est de cadrer le processus du développement d'un système. Elle propose des méthodes et outils pour maîtriser le développement des systèmes durant toutes les phases du cycle de vie, à savoir : la définition des besoins des clients, la conception, la réalisation, l'intégration et la vérification. Elle permet, ainsi, de gérer la complexité des systèmes, en diminuant le risque d'introduction des erreurs. Plusieurs normes, telles-que ANSI/EIA-632 [EIA/ANSI-632 1999], IEEE1220 [IEEE-1220 1999], ISO 15288 [ISO-15288 2002] visent à encadrer la conception des systèmes complexes en adoptant l'IS. Nous retenons la définition suivante pour la notion d'IS :

Définition 1.1 [Fiorèse et Meinadier 2012] *L'IS est une démarche méthodologique coopérative et interdisciplinaire, fondée sur la science et l'expérience, qui englobe l'ensemble des activités adéquates pour concevoir, développer, faire évoluer et vérifier un ensemble de produits, processus et compétences humaines apportant une solution économique et performante aux besoins des parties prenantes et acceptable par tous. Cet ensemble est intégré en un système, dans un contexte de recherche d'équilibre et d'optimisation sur tout son cycle de vie.*

L'IS propose un processus de développement fondé sur plusieurs étapes. Une des étapes les plus importantes dans le cycle de développement d'un système, selon l'IS, est l'ingénierie des exigences. Une exigence est définie par :

Définition 1.2 [Fiorèse et Meinadier 2012] *Une exigence prescrit une propriété dont l'obtention est jugée nécessaire. Son énoncé peut être une fonction, une aptitude, une caractéristique ou une limitation à laquelle doit satisfaire un système, un produit ou un processus.*

Le processus de l'ingénierie des exigences permet d'extraire les besoins des parties prenantes, de les analyser, de les spécifier et enfin, de les vérifier [Fiorèse et Meinadier 2012]. Nous nous intéressons, dans cette thèse, aux deux dernières étapes, à savoir : la spécification des exigences et la vérification.

1.2.1 La spécification des exigences

La spécification des exigences, dans un cycle de développement, consiste à documenter différents types d'exigences d'une façon cohérente, claire et précise [Alagar et Periyasamy 2011], dans le but de rendre ces exigences facilement compréhensibles par les parties prenantes [Wieggers et Beatty 2013]. Cet ensemble d'exigences est utilisé comme un référentiel qui suit le système durant tout son cycle de développement [Fiorèse et Meinadier 2012]. Dans l'étape de vérification, par exemple, ce référentiel est utilisé afin de s'assurer que le système respecte bien ses exigences. Le processus de spécification est défini comme :

Définition 1.3 IEEE-1233 [1998] *Une spécification d'exigences de système (SES) est généralement considérée comme un document dans lequel sont énoncées les exigences fixées par le client, à l'intention de la communauté technique chargée de la conception et de la réalisation d'un système. Le recueil d'exigences qui forme la spécification et sa représentation sert de passerelle entre les deux parties ; il doit donc pouvoir être compris par chacune d'elles. Lors de la création d'un système, l'une des tâches les plus délicates est d'ailleurs de communiquer ces exigences, en un seul document, à tous les sous-ensembles de ces deux groupes. Cette opération requiert généralement l'utilisation de divers formalismes et langages.*

La spécification doit inclure les activités suivantes :

- L'élicitation des exigences, où toutes les exigences du système sont récoltées, en consultant les parties prenantes et les différentes documentations des systèmes (état de l'art technique, normes, contraintes, etc.).

- L'analyse des exigences et la négociation, où toutes les exigences sont analysées en détail et acceptées par les parties prenantes.
- La validation des exigences, où la cohérence et la complétude des exigences sont vérifiées.

Le processus de spécification produit les documents qui expriment les exigences fonctionnelles, non-fonctionnelles et également les contraintes que le système doit satisfaire. L'ensemble des exigences doit être complet et cohérent. En effet, la complétude signifie que le référentiel d'exigences doit contenir toutes les exigences nécessaires au développement i.e. qu'il ne doit pas omettre ou contenir des exigences inutiles ou superflues. Le référentiel des exigences doit aussi être cohérent, ce qui signifie qu'il ne doit pas contenir des exigences contradictoires. Sur le plan élémentaire, les exigences doivent présenter les critères de qualité suivants [Fiorèse et Meinadier 2012] :

- Unicité : l'exigence porte sur un seul sujet.
- Précision : l'exigence est bien écrite et claire.
- Non-ambiguïté : l'exigence doit avoir une seule interprétation.
- Pure prescription de résultat : l'exigence doit porter sur ce qui est attendu (le quoi ?), non sur la solution (le comment ?).
- Vérifiabilité : pour chaque exigence, une méthode de vérification est définie.
- Faisabilité : l'exigence peut être réalisée dans le contexte de l'état de l'art technologique et technique.
- Réalisme : l'exigence doit être satisfaite dans le contexte des contraintes et ainsi être réalisable.

Les exigences sont écrites généralement selon un dictionnaire (langage) compris par tous les intervenants. Ces langages sont connus sous le nom de "*langages de spécification*". Nous présentons, dans la section suivante, un aperçu des différents langages de spécification.

1.2.1.1 Langages de spécification

Dans la littérature, plusieurs langages sont proposés pour la spécification des exigences. Les langages de spécification sont caractérisés par une syntaxe et une sémantique. Pour le langage naturel, comme le français par exemple, la syntaxe définit l'alphabet du langage et les règles grammaticales pour construire des phrases. La sémantique définit quant à elle, le sens des phrases construites. Les langages de spécification peuvent être classés, selon le caractère mathématique de leur sémantique, en trois classes : formels, semi-formels, ou naturels.

1.2.1.1.1 Les langages formels

Les langages formels sont caractérisés par une syntaxe et une sémantique formelle [Bjørner et Havelund 2014]. Ces langages, basés généralement sur des notations mathématiques

précises, permettent la vérification automatique des exigences spécifiées. Mais, la manipulation des notations mathématiques requiert un savoir-faire et une forte expertise. C'est pour cela que l'utilisation des langages formels est modérée en industrie. Ces langages formels sont de deux types : langages orientés modèle et langages orientés propriété [Wing 1990].

Les langages orientés modèle permettent la spécification des différentes vues du système directement dans un modèle unique [Wing 1990], caractérisé par des types de données concrètes [Bjørner et Havelund 2014]. Parmi ces langages, on peut citer : les machines à état, les automates temporisés [Alur et Dill 1990], les réseaux de Petri [Petri 1966, Reisig 2013], Z¹ [Spivey et Abrial 1992], Alloy [Jackson 2002], VDM² [Jones 1986], Raise [George et al. 1992], la méthode B³ [Abrial et Abrial 2005], B-événementielle [Abrial 2010], ASM [Gurevich et al. 1995] et TLA+⁴ [Lamport 2002].

Les langages orientés propriété spécifient le système en un ensemble de propriétés souhaitées, dont le but est de définir un modèle qui satisfait l'ensemble des propriétés spécifiées. Ils sont caractérisés par des types de données abstraites [Bjørner et Havelund 2014]. Ces langages peuvent être axiomatiques ou algébriques [Wing 1990].

Dans les langages axiomatiques, les objets sont définis par des types. Les opérations, sur ces objets, sont écrites sous forme d'assertions en logique du premier ordre [Alagar et Periyasamy 2011]. Parmi eux, on peut citer : HOL⁵ [Gordon et Melham 1993], PVS⁶ [Owre et al. 1999], Coq [Bertot et Castéran 2013] et les logiques temporelles [Manna et Pnueli 2012] comme CTL⁷ [Clarke et Emerson 1981].

Dans les langages algébriques, les objets et les processus sont définis en algèbre [Alagar et Periyasamy 2011]. Un objet est introduit comme un ensemble de définitions. Les opérations sur les objets sont définies par des équations. Les langages algébriques les plus connus sont : OBJ3 [Goguen et al. 1987], CafeOBJ [Diaconescu et Futatsugi 1998] et Maude [Clavel et al. 2002].

1.2.1.1.2 Les langages semi-formels

Les langages semi-formels sont caractérisés par une syntaxe formelle bien précise. Cependant, leur sémantique présente des ambiguïtés d'interprétation dues à leur caractère *polysémique*. Un langage est dit polysémique s'il contient un symbole qui représente plus qu'un seul objet [Héon et al. 2010]. Par conséquent, les langages semi-formels sont plus expressifs que

-
1. Z : Zermelo
 2. VDM : Vienna Development Method
 3. B : Bourbaki
 4. TLA : Temporal Logic of Actions
 5. HOL : High Order Logic
 6. PVS : Prototype Verification System
 7. CTL : Computation Tree Logic

les langages formels. De plus, ils reposent souvent sur des notations graphiques ou textuelles qui les rendent d'un côté faciles à utiliser, mais d'un autre côté, difficiles à analyser. L'analyse de ces langages nécessite leurs transformations en langages formels. Parmi les langages semi-formels, on peut citer : UML⁸ [OMG 2015b], SysML⁹ [OMG 2015a], OCL¹⁰ [OMG 2014], SADT¹¹ [Dickover et al. 1977], etc.

UML est le langage semi-formel le plus utilisé pour la spécification des exigences. Il offre plusieurs diagrammes permettant de modéliser, à l'aide d'une syntaxe graphique, les différentes vues statiques ou dynamiques d'un système. Il est caractérisé par une syntaxe précise définie par un méta modèle et des contraintes OCL. Par contre, sa sémantique n'est pas formellement définie, malgré les efforts de l'OMG dans le projet F-UML [OMG 2016]. Bien qu'étant le langage le plus utilisé sur le plan académique et industriel, UML manque de précision dans sa sémantique. Ce qui le rend difficilement analysable par une machine.

Afin de vérifier automatiquement les langages semi-formels, plusieurs travaux proposent de les transformer en langages formels. Par exemple, différents diagrammes UML sont transformés en langages formels [Dong et al. 2001, Truong et Souquières 2004, Kim et Carrington 2000, Ng et Butler 2003].

1.2.1.1.3 Les langages naturels

Les langages naturels, comme le français ou l'anglais, ont un pouvoir d'expression plus élevé que les langages semi-formels ou formels. Ils ont l'avantage aussi d'être compréhensibles par les différents intervenants du projet, une qualité très appréciée des industriels, qui continuent à les utiliser pour la spécification des exigences [Kudo et al. 2015]. En revanche, ces langages sont confus, ambigus et peuvent avoir plusieurs interprétations, ce qui les rend difficilement analysables par une machine et, par conséquent, ils peuvent être une source d'erreurs. Des erreurs, souvent introduites lors de la conception, résultent de mésinterprétations entre les différents concepteurs.

La norme SBVR¹² [OMG 2015c], du groupe OMG, propose un dictionnaire contrôlé pour la spécification des exigences. À la différence d'un langage purement naturel, ces langages contrôlés permettent aux exigences d'être automatiquement analysables.

Des travaux récents, comme ceux de Bajwa [2014], proposent de générer des notations formelles à partir des exigences spécifiées en Anglais. Dans leurs travaux, le langage naturel est parsé et normalisé selon la norme SBVR [OMG 2015c]. Une fois les spécifications normalisées, elles sont ensuite transformées en langages semi-formels comme OCL [Bajwa et al. 2010] ou formels comme Alloy [Bajwa et al. 2012]. D'autres travaux proposent d'utiliser les ontologies afin de transformer les contraintes, écrites en langage naturel, en un langage for-

8. UML : Unified Modeling Language

9. SysML : Systems Modeling Language

10. OCL : Object Constraint Language

11. SADT : Structured Analysis and Design Technique

12. SBVR : Semantic Business Vocabulary and Rules

mel [Sadoun et al. 2013].

Les exigences définies, lors de l'étape de spécification, doivent être vérifiées durant tout le cycle de développement d'un système. Nous présentons dans la section suivante la définition de la vérification ainsi que les différentes techniques de vérification.

1.2.2 La vérification des exigences

Si le rôle de la spécification est la réécriture des besoins des clients d'une façon claire et précise. La vérification a pour rôle de vérifier que les exigences, résultantes de la spécification, sont respectées durant tout le cycle de développement. Son objectif est de détecter les erreurs introduites dans chaque étape de développement. La vérification répond à la question : "Faisons-nous le produit correctement ?". Selon la norme ISO 9000, la vérification consiste à :

Définition 1.4 [ISO 2000] *La confirmation par des preuves tangibles que les exigences spécifiées ont été satisfaites.*

A partir de cette définition, on peut constater que la vérification utilise des preuves tangibles, c.à.d. des méthodes, pour déterminer si les exigences sont satisfaites ou non. Nous présentons, dans la section suivante, un aperçu de la littérature des différentes méthodes de vérification.

1.2.2.1 Les méthodes de vérification

Une méthode de vérification consiste à s'assurer que le système satisfait une exigence (propriété). Les méthodes de vérification les plus utilisées, dans la littérature, sont : le test, la simulation et les méthodes formelles.

1.2.2.1.1 Le test

Le test permet de détecter les erreurs d'un système par rapport à un ensemble de scénarios définis antérieurement [Beizer 2003]. De ce fait, cette méthode est loin d'être exhaustive. Le test est pratiqué généralement après l'intégration du système, une fois que ce dernier est opérationnel. La correction des erreurs détectées à ce stade de développement (intégration), reste la plus coûteuse [Boehm et al. 1981]. Néanmoins, cette méthode de vérification reste la plus utilisée dans l'industrie [Bjørner et Havelund 2014], car elle peut être réalisée sur des systèmes de taille importante. Elle permet aussi de vérifier le système réel et pas seulement une abstraction de celui-ci, comme dans le cas de la simulation ou des méthodes formelles.

1.2.2.1.2 La simulation

La simulation consiste à reproduire le comportement d'un système afin de vérifier son bon fonctionnement. Le comportement simulé est capturé sous forme de modèles abstraits qui

sont, ensuite, exécutés sur des plateformes dédiées. La simulation consiste, alors, à faire des expérimentations sur des modèles abstraits [März et al. 2010]. En effet, l'analyste peut jouer des scénarios sur le simulateur afin de vérifier les performances du système.

Comme le test, la simulation peut être utilisée pour vérifier des systèmes de taille réelle, ce qui justifie son importante utilisation dans l'industrie. Mais, contrairement au test, la simulation peut être pratiquée dans des étapes en amont à l'implémentation et à l'intégration. Cela permet de détecter les erreurs avant que le système ne soit implémenté et ainsi réduire les coûts de correction. En revanche, la simulation souffre de plusieurs inconvénients comme le degré de réalisme ou granularité du simulateur. En effet, elle ne couvre pas la totalité du comportement du système, ce qui limite le nombre de scénarios vérifiables. De ce fait, la simulation n'est pas exhaustive.

1.2.2.1.3 Les méthodes de vérification formelle

Les méthodes formelles, telle que définies par Bjørner et Havelund [2014], sont des techniques basées sur les mathématiques. Elles permettent de vérifier formellement des spécifications écrites en langages formels. Elles assurent que le système est correct en vérifiant un ensemble de propriétés. Elles donnent, ainsi, une cartographie des propriétés satisfaites et non satisfaites du système développé. Le caractère mathématique de ces méthodes, les rend plus précises et exhaustives comparées aux tests [Bennion et Habli 2014] et à la simulation. Ainsi, ces techniques sont de plus en plus utilisées pour la vérification des systèmes actuels. Mais leur adoption dans l'industrie souffre de plusieurs obstacles [Bjørner et Havelund 2014], malgré les résultats encourageants de certaines études [Bennion et Habli 2014]. Les méthodes formelles sont de deux types : le *Model-Checking* et le *Theorem-Proving*.

Le *Model-Checking*. [Schnoebelen et al. 1999, Baier et al. 2008] est une méthode formelle purement automatique, généralement supportée par un outil informatique "le *Model-Checker*". Elle consiste à vérifier automatiquement les propriétés souhaitées sur une spécification du système. Le système est généralement modélisé par des machines à états. Les propriétés à vérifier sont écrites dans une logique temporelle. Puis, sur la base des spécifications du système et des propriétés, le *Model-Checker* explore tout l'espace d'états. Il vérifie dans chaque état la satisfaction de la propriété désirée. Si la propriété est violée, le *Model-Checker* retourne un contre-exemple illustrant la trace de l'erreur, i.e, la succession d'états qui violent la propriété [Schnoebelen et al. 1999]. Cependant, pour les systèmes de taille importante, cette exploration exhaustive d'états peut dépasser la capacité des machines (ordinateurs), ce qui fait que la vérification échoue. Ce problème est connu sous le nom de l'*explosion de l'espace des états*.

Le *Model-Checking* est très utile pour la vérification des systèmes critiques complexes [Bennion et Habli 2014, Mesli-Kesraoui et al. 2016b;a]. En revanche, le problème de l'explosion de l'espace des états reste à ce jour, un vrai obstacle pour son adoption dans l'industrie. Afin d'évi-

ter cette explosion de l'espace des états, plusieurs techniques sont proposées dans la littérature comme : les BDD, les SAT, l'abstraction et la symétrie [Xin-feng et al. 2009].

UPPAAL [Larsen et al. 1997]¹³ est un exemple de *Model-Checker*, utilisé pour la vérification formelle des systèmes temporisés. Le *Model-Checker* SPIN [Holzmann 2004]¹⁴ est utilisé pour la vérification des systèmes concurrents. Roméo [Gardey et al. 2005]¹⁵ est un *Model-Checker* pour la vérification des systèmes temps réel modélisés par des réseaux de Petri.

Theorem-Proving. Les langages axiomatiques sont généralement vérifiés par le *Theorem-Proving* [Alagar et Periyasamy 2011]. Le *Theorem-Proving* est une méthode formelle basée sur les logiques mathématiques. Ces logiques sont utilisées pour spécifier le système et ses propriétés attendues (exigences). Ensuite, certaines règles d'inférence et axiomes sont appliqués afin de déduire et de prouver les propriétés à partir de la spécification du système. Ce procédé ne souffre pas du problème de l'explosion de l'espace des états, mais, selon l'expressivité de la logique utilisée, le *Theorem-Proving* peut être entièrement automatique ou nécessiter une intervention humaine [Almeida et al. 2011]. Cette technique requiert de l'expérience et de la connaissance formelle. Si la logique utilisée est moins expressive, comme la logique du premier ordre par exemple, la construction de la preuve est entièrement automatique. C'est le cas par exemple du *theorem prover* ACL2 [Kaufmann et Moore 1996]. Par contre, si la logique est plus expressive, comme la logique d'ordre supérieur, l'automatisation de la preuve est partielle. Comme il n'existe pas de procédures de décision capables de prouver certaines propriétés dans cette logique [Almeida et al. 2011], l'assistant de preuve nécessite l'intervention de l'analyste pour conduire la preuve.

Isabelle/HOL [Nipkow et al. 2002] est un assistant de preuve pour la logique d'ordre supérieur. PVS [Owre et al. 1999] est un langage de spécification formel et un démonstrateur de théorèmes. Why3 [Bobot et al. 2011] est une plateforme de vérification de programmes basée sur le langage de spécification WhyML. L'idée derrière Why3 est de "*fournir autant que possible d'automatisation*" [Bobot et al. 2011]. En effet, le langage WhyML, basé sur la logique du premier ordre, est transformé aux formats d'entrées de différents démonstrateurs de théorèmes (PVS, Coq, Z3 ...). Les résultats montrent que why3 donne une meilleure performance de vérification que d'autres démonstrateurs de théorèmes, concernant le nombre d'obligations de preuves chargées [Mentré et al. 2012].

Après avoir présenté le contexte théorique de nos travaux de thèse, nous présentons, dans les sections suivantes, le contexte industriel et l'état de l'art technique du projet. Nous évoquons, ensuite, la problématique liée à nos travaux de thèse et les verrous scientifiques qui en ressortent.

13. <http://www.uppaal.org/>

14. <http://spinroot.com/spin/whatispin.html>

15. <https://romeo.rts-software.org/>

1.3 Conception des systèmes de contrôle-commande : contexte industriel

Cette thèse s'inscrit dans un cadre industriel portant sur la conception des systèmes de contrôle-commande. Nous décrivons dans les sections suivantes les différentes parties constituant un système de contrôle-commande. Ensuite, nous abordons les différentes approches utilisées dans la littérature pour la conception de ces systèmes.

1.3.1 Les systèmes de contrôle-commande

Un système de contrôle-commande permet le pilotage d'un procédé industriel physique au travers des fonctions de commande, de surveillance et de supervision (figure 1.1). La commande a un rôle opérationnel qui consiste à faire exécuter un ensemble d'opérations pour agir sur le procédé physique [Niel et Craye 2002]. La surveillance a un rôle informationnel qui porte sur le recueil des signaux provenant du procédé et de la commande, reconstituer l'état du système, dresser des historiques et le traitement de défaillance. La supervision a un rôle décisionnel qui permet d'optimiser, en présence ou non de défaillances, le fonctionnement du système.

L'interaction du système de contrôle-commande avec le procédé physique est réalisée, d'une part, par des observations, à travers des capteurs et, d'autre part, par des actions réalisées par l'intermédiaire d'actionneurs [Cottet et Grolleau 2005] (figure 1.1). Les capteurs permettent de transformer les grandeurs physiques du procédé matériel en signaux électriques interprétables par le système de contrôle-commande. Les actionneurs (moteurs, transformateurs...) permettent de transformer les commandes électriques du système de contrôle-commande, en ordres qui permettent d'agir sur le procédé physique en changeant son état.

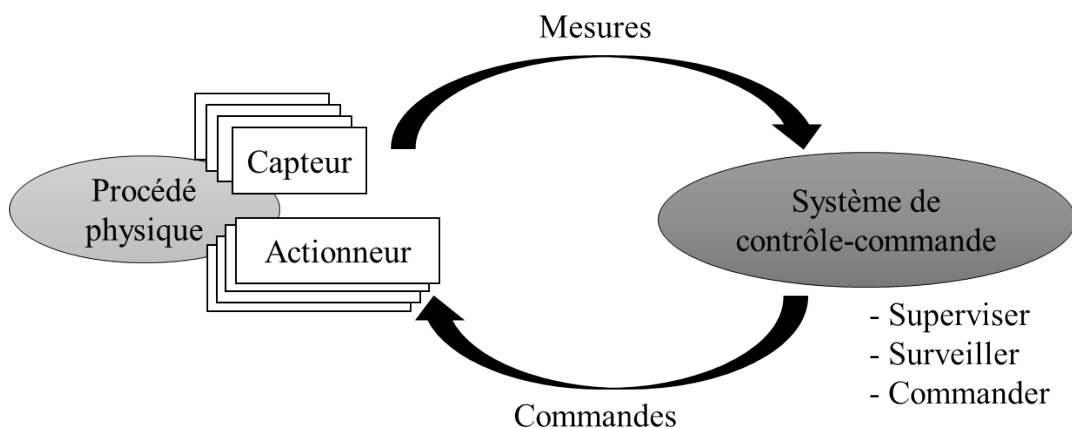


FIGURE 1.1 – Représentation d'un système de contrôle-commande

Dans l'industrie, il existe plusieurs types de systèmes de contrôle-commande dont les systèmes de type SCADA, sur lesquels nous allons nous focaliser.

1.3.2 L'architecture SCADA

Les systèmes SCADA¹⁶ ou télésurveillance et acquisition des données sont des systèmes de contrôle-commande qui se focalisent plutôt sur la supervision [Daneels et Salter 1999]. Un système SCADA est caractérisé par une architecture matérielle et une architecture logicielle.

L'architecture matérielle d'un SCADA (figure 1.2) est généralement composée des stations clients, des serveurs de données, des Automates Programmables Industriels (API) et du système physique contrôlé. Les stations clients gèrent principalement l'interaction homme-machine. Les serveurs de données ont pour rôle la gestion et l'acquisition des données à partir des automates programmables. À partir des données des capteurs, les automates programmables agissent sur le procédé physique à travers les actionneurs.

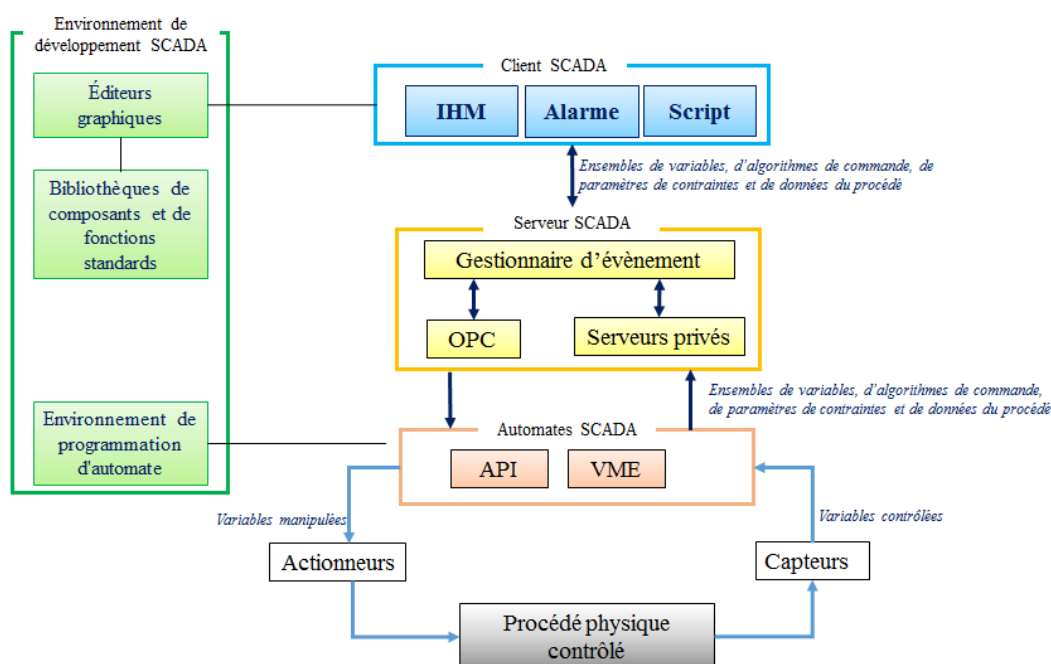


FIGURE 1.2 – L'architecture SCADA

L'architecture logicielle d'un système SCADA est composée des logiciels embarqués dans les différents composants de l'architecture physique. Le client SCADA est constitué d'une Interface Homme-Machine (IHM), généralement appelée "IHM de supervision" qui affiche toutes les informations liées aux états du système contrôlé et qui permet aux opérateurs de

16. SCADA : Supervisory Control And Data Acquisition

suivre l'état du système et de le contrôler à distance. Le client contient aussi un gestionnaire d'alarmes et de scripts (figure 1.2).

Le serveur SCADA est le coordinateur entre l'IHM et les programmes de commande. Il est constitué d'un gestionnaire d'événements et intègre plusieurs protocoles de communications tels que OPC, etc. Le gestionnaire d'événements se charge du transfert de données entre le client SCADA et les API. En effet, le gestionnaire d'événements assure, d'une part, l'acquisition des données provenant des API et, d'autre part, il permet de transformer les consignes des opérateurs sur l'IHM de supervision, en données interprétables qui sont ensuite envoyées aux programmes de commande.

Les API englobent des programmes de commande écrits dans la norme IEC 61131-3 [IEC 2013] et s'exécutent d'une manière cyclique. Dans chaque cycle, les données provenant des serveurs de données et des capteurs du système physique sont capturées. À partir de ces données, le programme de commande calcule les sorties qu'il envoie, d'une part, aux actionneurs pour manipuler le système physique et, d'autre part, aux serveurs de données pour afficher l'état du système à l'utilisateur sur l'IHM de supervision.

1.3.3 Approches traditionnelles pour la conception des systèmes de contrôle-commande

La conception des systèmes SCADA porte sur la conception de l'architecture matérielle et également logicielle. La conception de l'architecture matérielle consiste à définir les composants constitutifs du système physique, les serveurs de données, les API et les outils SCADA à utiliser pour l'exploitation (exécution) des IHM de supervision. La conception de l'architecture logicielle porte sur deux axes : la conception des IHM de supervision et la conception des programmes de commande. Ces deux axes sont détaillés ci-après.

1.3.3.1 Conception des IHM de supervision

Les interfaces de supervision industrielles sont des systèmes interactifs, c.à.d. que l'utilisateur peut interagir avec le système à travers une IHM. La conception des systèmes interactifs repose sur la séparation entre la partie graphique et l'application [Aït-Ameur et al. 2005]. L'architecture d'un système interactif est généralement composée de trois modules : présentation, dialogue et application (figure 1.3).



FIGURE 1.3 – Architecture simplifiée d'un système interactif

La présentation est la partie graphique présentée à l'utilisateur. Elle est composée de plusieurs widgets (boutons, listes déroulantes, etc.) qui permettent à l'utilisateur d'interagir avec

l'interface. Le dialogue ou le contrôleur de dialogue est le principal élément dans la conception des systèmes interactifs. Il joue le rôle de coordinateur entre la partie présentation et l'application. En effet, il permet d'invoquer les fonctions nécessaires pour accomplir une tâche demandée par l'utilisateur. Il permet aussi d'afficher l'état du système à l'utilisateur à travers des objets d'affichage. L'application regroupe toutes les données et fonctions qui permettent au système d'évoluer et d'accomplir les tâches demandées par l'utilisateur. Dans les systèmes SCADA, l'application est constituée de serveurs de données, de programmes de commande et du procédé physique.

Dans la littérature, plusieurs approches ont été proposées pour, d'une part, intégrer l'utilisateur dans le processus de conception et, d'autre part, cadrer le processus de conception des systèmes interactifs [Kolski et al. 2001]. Parmi ces approches : le modèle en étoile, le modèle de Long et le modèle Nabla.

Le modèle en étoile [Hartson et Hix 1989] propose un cycle de développement des systèmes interactifs sous forme d'étoile où l'évaluation est au centre du modèle. L'idée derrière ce modèle est de permettre à l'utilisateur d'évaluer toutes les étapes de développement. Bien que cette approche place l'utilisateur au centre de la conception, le cycle de vie proposé est en inadéquation avec les pratiques séquentielles de développement utilisées dans l'industrie.

Le modèle de Long [Long et Denley 1990] utilise le même principe en ajoutant des étapes d'évaluation et des itérations à chaque étape du cycle de développement classique (séquentiel).

Le modèle Nabla, proposé par Kolski [1995], intègre aussi l'utilisateur dans le cycle de développement, tout en différenciant la conception de l'interface de celle des modules d'aides. Pour concevoir un système interactif adapté, d'une part, aux besoins informationnels des utilisateurs et, d'autre part, aux besoins en mode de coopération utilisateur-module d'aide, le cycle de vie du modèle Nabla comporte un double cycle en V.

1.3.3.2 Conception des programmes de commande

Les API présentent un fonctionnement opérationnel cyclique [De Smet et Rossi 2002] sur trois étapes (figure 1.4).

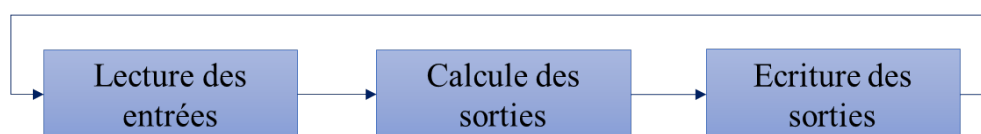


FIGURE 1.4 – Cycle opérationnel d'un API

Dans la première étape, les données provenant de l'environnement (IHM de supervision et les capteurs du système physique) sont lues et stockées dans des variables internes. Dans la deuxième étape, le programme de commande est exécuté, c.à.d. que des données de sorties sont calculées à partir des données d'entrée selon les instructions du programme de commande. À

la troisième et dernière étape, ces sorties sont écrites, i.e. envoyées à l'environnement composé des actionneurs et de l'IHM de supervision dans le cas des systèmes SCADA.

Pour la réalisation des programmes de commande exécutables par des APIs, la norme IEC 61131-3 [IEC 2013] propose cinq langages normalisés : LD¹⁷, IL¹⁸, ST¹⁹, SFC²⁰, FBD²¹. Ainsi, la conception des programmes de commande consiste à produire des programmes dans un de ces cinq langages.

Pour la conception de la commande, plusieurs approches traditionnelles ont été proposées dans la littérature. Ces approches, détaillées dans [Lee 2006, Bévan 2013, Coupât 2014], sont soit basées sur la théorie du contrôle par supervision, soit sur la synthèse algébrique ou bien basées sur les contraintes logiques (filtres).

La théorie de supervision (Supervisory Control Theory) a été initiée principalement par Wonham et Ramadge [1988]. Elle repose sur la définition d'un superviseur général qui impose au système physique de respecter un comportement désiré [Wonham et Ramadge 1988]. Le système physique est considéré comme un générateur des événements *contrôlables* et *non-contrôlables*. Le rôle du superviseur est d'inhiber ou d'autoriser les événements contrôlables pour respecter les spécifications du cahier des charges. Les événements non-contrôlables sont toujours autorisés. Cette méthode purement théorique, repose sur la réalisation d'un superviseur. Or, les programmes de commande des API cherchent à définir un **contrôleur** qui permet d'agir sur le système contrôlé.

La commande supervisée est une extension très intéressante de la théorie de la supervision qui permet de séparer la commande de la supervision [Charbonnier 1996]. La commande couplée au système physique forme un système étendu qui génère des événements. Ces derniers sont autorisés ou inhiber par le superviseur.

Ces approches offrent la possibilité de construire une commande sûre qui intègre les spécifications du cahier des charges. Le comportement du superviseur est modélisé généralement par des outils formels comme les automates à états finis, les réseaux de Petri [Toguyeni et al. 2007], etc. Par conséquent, ces approches souffrent du problèmes de l'explosion combinatoire car le nombre des états du superviseur peut évoluer d'une manière exponentielle par rapport au nombre de composants [Coupât 2014].

La synthèse algébrique [Hietter 2009] repose sur l'utilisation de l'algèbre de Boole et des fonctions booléennes pour la conception de la commande d'un système critique. En effet, le concepteur a la charge de définir des fragments de la spécification sous forme de relations entre des fonctions booléennes. La cohérence de ces fragments est vérifiée automatiquement. Ensuite, un système de résolution d'équations permet la génération d'un ensemble de solutions (lois de commande). Le concepteur peut alors choisir la solution qui satisfait ses critères.

17. LD : Ladder Diagram

18. IL : Instruction List

19. ST : Structured Text

20. SFC : Sequential Function Chart

21. FBD : Function Block Diagram

Cette méthode présente des avantages et également des inconvénients. En effet, elle permet l'obtention automatique des lois de commande, ce qui permet à terme leur génération automatique. Toutefois, les formalismes (algèbre de Boole...) manipulés par cette méthode sont très différents des standards de développement et donc difficilement interprétables par les experts métier [Coupat 2014].

La commande par contraintes logiques est basée sur l'utilisation d'un filtre entre la partie commande et la partie opérative [Alanche et al. 1986, Toguyeni 1992, Marangé 2008]. Ce filtre intègre les contraintes de sécurité que le système physique doit respecter. Le filtre est vérifié hors ligne (avant l'exécution du code) et il est utilisé pour une vérification en ligne de la commande [Marangé 2008]. En effet, le filtre permet d'empêcher la partie opérative (système physique) d'atteindre une situation de défaut, qui viole des contraintes de sécurité, lors de l'exécution du programme de commande. Le filtre assure que le système physique n'atteint jamais une situation dangereuse indépendamment du programme de commande. De ce fait, l'approche par filtre permet de garantir la sécurité. De plus, elle peut s'appliquer à des programmes d'API déjà existants. En revanche, la description de toutes les contraintes de sécurité dans le filtre peut être une tâche fastidieuse.

1.3.3.3 Bilan des approches traditionnelles

Bien que les approches traditionnelles reposent sur des concepts théoriques solides, leur application dans le monde industriel reste très limitée, car d'une part, ces approches changent le cycle de développement séquentiel très répandu en industrie [Bignon 2012]. D'autre part, l'utilisation de ces approches traditionnelles dans la conception des systèmes de contrôle-commande complexes, entraîne des erreurs importantes et, par conséquent, des coûts élevés dus à la re-conception [Goubali 2017]. En complément de ces techniques de conception traditionnelles et afin de réduire les temps de conception et les erreurs, il existe d'autres approches qui exploitent les apports d'une modélisation basée sur les modèles pour générer automatiquement des programmes de commandes et des IHM de supervision.

1.3.4 Génération automatique des programmes de contrôle-commande

Contrairement aux approches de conception traditionnelles, les approches basées sur les modèles permettent de générer automatiquement les programmes de commandes et/ou les IHM de supervision à partir d'un modèle abstrait du système. Dans la littérature la génération automatique a été traitée selon trois grandes approches : l'approche ascendante, l'approche descendante et l'approche mixte.

1.3.4.1 L'approche ascendante

Dans l'approche ascendante, la conception du système est réalisée par agrégation des différents composants constitutifs du système. Cette approche inspirée de la notion d'objet en

informatique, favorise la réutilisation et la structuration du code.

Les travaux de [Mouchard \[2002\]](#) et [Lallican \[2007\]](#) s'inscrivent dans la démarche ascendante. En effet, les programmes de commande élémentaires sont générés automatiquement à partir d'un modèle d'objet du système créé à partir d'une bibliothèque de composants prédéfinis. Chaque composant encapsule un ensemble de vues (vue de commande, vue de contrainte, vue de partie opérative,...). Ces travaux visent seulement la génération des programmes de commande élémentaires et les modèles du système physique (partie opérative), les interfaces de supervisions ne sont pas générées.

Les outils SCADA offrent souvent une bibliothèque de composants et de fonctions prédéfinis (figure 1.2), que les développeurs utilisent par instanciation (copies) pour construire le système global. SIMATIC Automation Designer [[Falkman et al. 2011](#)] est un exemple d'outil SCADA permettant la génération automatique des programmes de commande et des interfaces de supervision à partir d'une bibliothèque de composants prédéfinis dans cet outil.

L'approche ascendante offre des avantages importants pour la réutilisation des composants standards dans plusieurs projets, ce qui réduit considérablement les temps de conception. En revanche, le code du système global, résultant d'une agrégation de codes des composants constitutifs, est souvent volumineux. Des routines d'optimisation sont généralement appliquées pour optimiser les codes générés.

1.3.4.2 L'approche descendante

Contrairement à l'approche ascendante, l'approche descendante part d'une description abstraite du système complet afin d'atteindre, après plusieurs raffinages, une description plus détaillée. Plusieurs auteurs proposent l'utilisation de cette approche pour la génération automatique des programmes de commande et des IHM de supervision.

Au niveau des programmes de commande, les travaux de [de Lamotte \[2006\]](#) proposent une solution pour la modélisation, l'analyse et la génération automatique des programmes de commande pour des systèmes reconfigurables. Pour ce faire, l'auteur propose un langage de haut niveau (abstrait) permettant la modélisation de l'architecture physique et logique du système. À partir de ce modèle, les programmes de commande (SFC) sont alors générés automatiquement par transformation de modèles. [Sacha \[2010\]](#) propose une approche pour la génération automatique des programmes de commande à partir d'une spécification abstraite en UML. [Wang et al. \[2009\]](#) utilisent le langage formel des automates temporisés pour modéliser le comportement d'un programme de commande. À partir des automates temporisés, les codes de programmes de commande sont générés en langage Ladder. La méthode SCADE²² [[Technologies, Dormoy 2008](#)] permet la génération automatique du code C d'une application à partir d'un langage graphique et formel (SCADE). Cette approche offre aussi des outils de vérification formelle et de simulation pour la vérification des modèles SCADE avant leur implémentation.

Au niveau des IHM, l'outil Ergo-Conceptor [[Moussa et Kolski 1991](#)] adopte l'approche

22. SCADE : Safety-Critical Application Development Environment

descendante pour la génération automatique des IHM de supervision pour les systèmes industriels complexes. L'approche proposée par Ergo-Conceptor se compose de trois modules. Dans le premier module, le système à contrôler est décrit selon les besoins de l'utilisateur humain. Cette description est utilisée dans le deuxième module, pour générer les spécifications de l'IHM. À partir de ces spécifications, les vues graphiques des IHM sont générées semi-automatiquement [Moussa et Kolski 1991; 1992].

L'approche descendante repose sur la description du système en sa globalité contrairement à l'approche ascendante (composant). Néanmoins, cette approche souffre de plusieurs inconvénients. En effet, les modèles utilisés dans cette approche manquent de modularité et par conséquent sont difficilement réutilisables. En plus, si des erreurs de conception sont commises sur les modèles de départ, le raffinage successif peut propager ces erreurs sur les modèles concrets.

Il est clair que les approches (descendante et ascendante) sont complémentaires. Nous présentons, dans la section suivante, un ensemble de travaux de recherche qui combinent ces deux approches (approche mixte) pour la génération automatique des programmes de commande et des interfaces de supervision.

1.3.4.3 L'approche mixte

Ce type d'approche repose sur la combinaison des approches ascendantes et descendantes. En effet, une bibliothèque de composants est utilisée en complément d'un modèle abstrait qui modélise le système. Ce dernier est construit par une approche ascendante à partir des composants prédéfinis dans la bibliothèque. Il est ensuite raffiné successivement jusqu'à l'obtention des modèles plus détaillés ou des codes exécutables.

Bévan [2013] propose la génération automatique des programmes de commande et des modèles de la partie opérative pour des systèmes transitoires. Cette génération automatique est réalisée à partir d'un modèle abstrait regroupant un ensemble de composants prédéfinis (sur étagère).

Dans le même contexte, le projet Anaxagore [Bignon 2012] pour la conception des systèmes sociotechniques reconfigurables repose sur une approche mixte. Nous présentons ce projet en détail dans les sections suivantes.

1.3.4.3.1 Le projet Anaxagore

Le projet Anaxagore est né de l'expérience personnelle de Bignon [2012] dans la conception des systèmes sociotechniques. Ces derniers sont des systèmes complexes caractérisés par une forte interaction avec l'humain. Ils comprennent des parties techniques et des opérateurs humains qui utilisent le système en situations réelles [Luzeaux et Ruault 2013]. La conception des systèmes sociotechniques fait intervenir de plus en plus des concepteurs de domaines techniques très variés. Cette diversité de concepteurs entraîne plusieurs problèmes de communication et d'interprétation des spécifications [Bignon 2012]. Ces problèmes sont source d'erreurs

qui sont détectées généralement dans les phases de tests. La résolution de ces problèmes, dans le projet Anaxagore, a été abordée selon deux axes.

D'une part, la standardisation de briques de conception est considérée comme une solution efficace pour fiabiliser la conception de produits matériels ou logiciels. C'est par ailleurs une attente forte des industriels pour réduire leurs coûts et délais de conception [Nguyen et al. 1997]. Ces techniques de standardisation, utilisées initialement en génie logiciel, facilitent la réutilisation et la génération automatique du code. Elles reposent sur différents paradigmes comme l'objet et le modèle. L'objet est créé une seule fois et peut être réutilisé plusieurs fois à travers des instances (copies). Les composants prédéfinis, ou "sur étagère", que les concepteurs utilisent pour concevoir le système global en les connectant les uns aux autres, incarnent un exemple réel de l'adoption de cette notion d'objet dans l'industrie.

Le modèle est au cœur du processus de conception dans l'ingénierie dirigée par les modèles (IDM). L'IDM, proposée par l'OMG [OMG 2003], est une forme d'ingénierie générative, à travers laquelle, le système est modélisé sous forme de modèle. Ce dernier doit être conforme à un méta-modèle (modèle abstrait). À partir de ce modèle, des modèles plus spécifiques (détaillés) peuvent être générés automatiquement par des transformations de modèles. Un cas courant de l'utilisation des concepts de l'IDM dans la conception du logiciel, est la génération du code à partir de modèles abstraits de l'architecture [Cavalcante et al. 2014, Bignon et al. 2010, Goubali et al. 2014].

D'autre part, une enquête menée auprès des concepteurs des systèmes sociotechniques a permis de montrer que ces derniers se basent sur un schéma technique abstrait décrivant l'architecture du système [Bignon et al. 2010]. Ce schéma suit le système durant l'ensemble de son cycle de développement. En effet, chaque concepteur se base sur le modèle abstrait de l'architecture pour concevoir la facette du système qui correspond à son métier. Dans les systèmes de gestion de fluide, par exemple, l'architecture physique du système est modélisée par un schéma de tuyauterie et instrumentation ou P&ID²³.

Afin de réduire les erreurs de conception et les coûts de re-conception, en favorisant la réutilisabilité, le projet Anaxagore propose un flot de conception pour les systèmes sociotechniques reconfigurables [Bignon 2012, Bignon et al. 2013; 2010, Goubali et al. 2014]. Dans le cadre de ce projet, une démarche de conception conjointe de programmes de commande et d'interfaces de supervision pour ces systèmes, a été proposée. La démarche est fondée sur une importante standardisation des composants constitutifs du système à concevoir. En effet, la démarche s'appuie sur une bibliothèque de composants de conception standardisés pour la spécification du système et la génération des applicatifs. Chaque composant se compose de vues représentant chacune une facette de la conception. Cette démarche utilise également les techniques de l'IDM pour créer des passerelles entre les différents corps de métiers.

Pour les systèmes de gestion de fluide, comme le montre la figure 1.5, Anaxagore permet de générer une application de contrôle-commande complète pour un système spécifié grâce à

23. P&ID : Piping & Instrumentation Diagram

un modèle métier normalisé (le P&ID) [Bignon et al. 2013, Goubali et al. 2014].

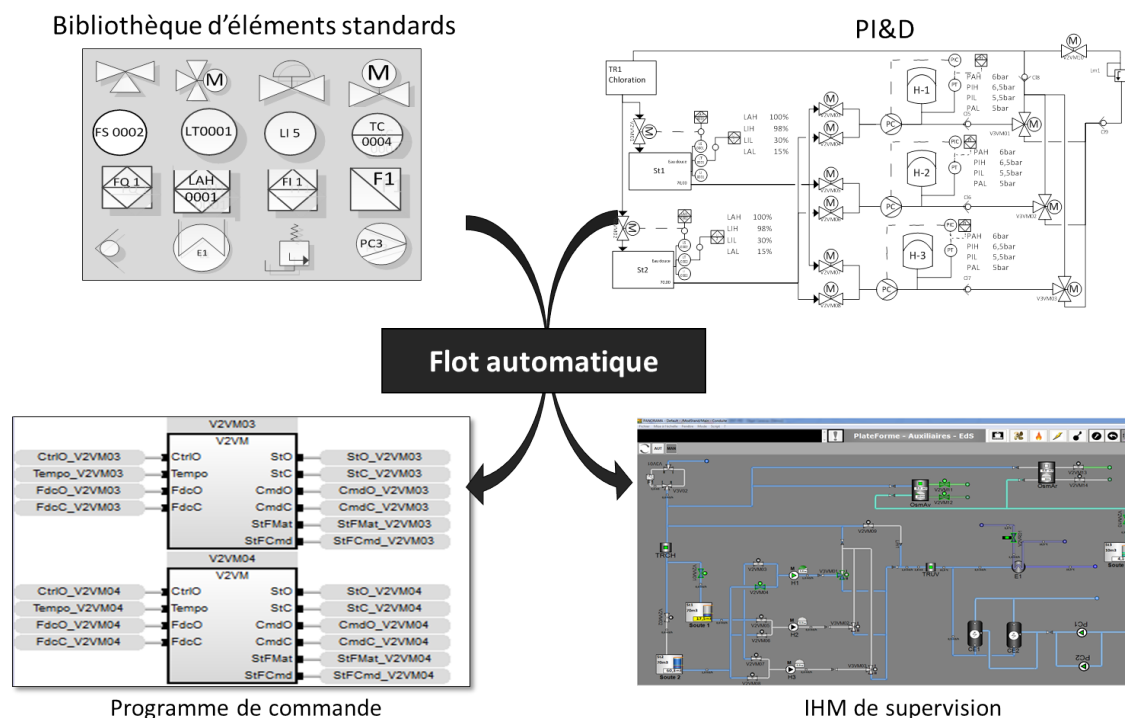


FIGURE 1.5 – Le projet Anaxagore

L'utilisation d'une bibliothèque de composants standards et les techniques de l'IDM permettent de réduire les efforts et les temps de conception tout en augmentant la réutilisabilité. En effet, les composants standards, développés une fois et stockés dans une bibliothèque, peuvent être réutilisés, sous forme d'instances, autant de fois que nécessaire dans le système. La génération automatique des applicatifs à partir du P&ID, implémentée par des transformations de modèles, permet de garantir la cohérence entre les applicatifs et le modèle P&ID et ainsi, de réduire les erreurs. Dans les sections suivantes, nous présentons la bibliothèque de composants standards, le modèle métier (P&ID) et le flot de conception d'Anaxagore.

La bibliothèque de composants standards. La bibliothèque de composants standards contient plusieurs composants. Chaque composant se compose de vues représentant chacune une facette de la conception. Comme le montre la figure 1.6, les trois vues sont : une vue de supervision, une vue de commande et une vue de synoptique.

La vue de supervision d'un composant correspond au composant logiciel permettant de le représenter sur l'interface homme-machine (IHM) de supervision. Cette IHM permet aux opérateurs des machines de contrôler les composants physiques à distance. Les vues de supervision

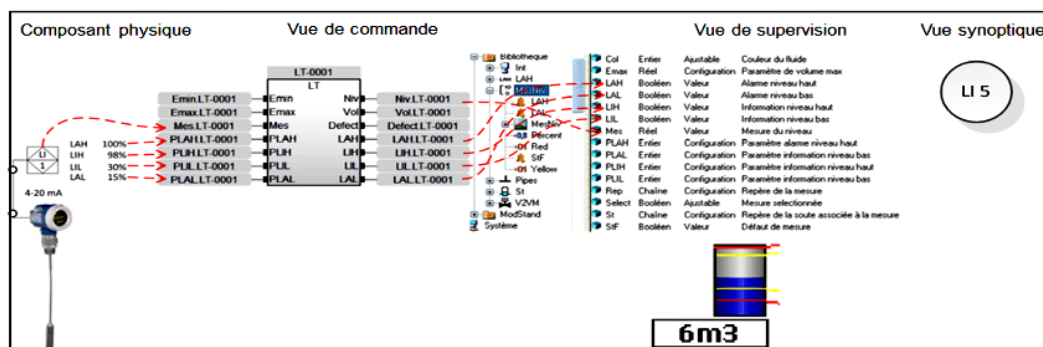


FIGURE 1.6 – Structure d'un composant standard dans Anaxagore

des différents composants ont été implémentées avec le logiciel Panorama E2.

Écrite en langage LD, selon la norme IEC 61131-3 [IEC 2013], la vue de commande contient les modèles qui permettent de contrôler et commander chaque composant à distance. Les instructions des opérateurs dans la salle machine sont capturées par l'interface de supervision à travers les différents widgets de l'interface graphique. Ces instructions sont ensuite transmises au programme de commande qui, après lecture de l'état des différents capteurs liés au composant, envoie les ordres de commandes permettant d'activer les actionneurs afin de changer l'état du composant physique (par exemple, ouvrir un composant fermé, ou fermer un composant ouvert).

La vue synoptique décrit le symbole graphique du composant sur le P&ID. Les vues synoptiques de chaque composant ont été créées par l'outil Visio de Microsoft.

Le modèle métier : P&ID. Le P&ID, illustré sur la figure 1.5, est une description graphique détaillée montrant l'ensemble de la tuyauterie, les équipements et une grande partie de l'instrumentation associée à un processus donné [McAviney et Mulley 2004]. Le P&ID est un diagramme normalisé qui schématise les composants et les connexions d'un procédé industriel. Chaque composant est représenté par un symbole défini dans la norme ANSI/ISA-5.1-1984 [ISA 1992]. Ces composants sont reliés par des connecteurs matériels, comme les tuyaux et les câbles, ou bien des liens logiciels [Bignon et al. 2013]. Les données échangées entre le système physique et le programme de commande (les instrumentations) sont aussi représentées dans ce diagramme.

Le flot de conception Anaxagore. Le flot de conception Anaxagore est basé sur l'approche mixte [Bignon 2012, Bignon et al. 2010] pour la génération conjointe de programmes de commande et d'interfaces de supervision. Il se compose de plusieurs opérations (figure 1.7).

L'opération de construction est une tâche manuelle réalisée par le concepteur en suivant une démarche ascendante [Bignon 2012]. Le concepteur se base sur la bibliothèque de composants

standards et les besoins définis dans le cahier des charges pour définir l'architecture physique du système à travers un diagramme P&ID (synoptique).

Les autres opérations, telles que, l'opération d'inventaire, d'assemblage et d'insertion adoptent l'approche descendante. Ainsi, la nomenclature, le programme de commande et l'interface de supervision sont raffinés à partir du modèle abstrait : le P&ID.

L'opération d'inventaire permet de générer automatiquement, par une transformation de modèle, la nomenclature. Cette dernière liste tous les composants et les connecteurs du système. L'interface de supervision est générée automatiquement, par l'opération d'insertion, à partir de la nomenclature, de la bibliothèque et d'un modèle standard d'IHM. Ce dernier est une interface graphique développée initialement sous une plateforme dédiée. Lors de l'opération d'insertion, ce modèle standard d'IHM est enrichi par les différentes vues de supervision des composants standards figurants sur le P&ID. L'opération d'assemblage permet de générer, à partir d'un modèle de programme de commande, de la nomenclature et de la bibliothèque, un programme de contrôle-commande élémentaire.

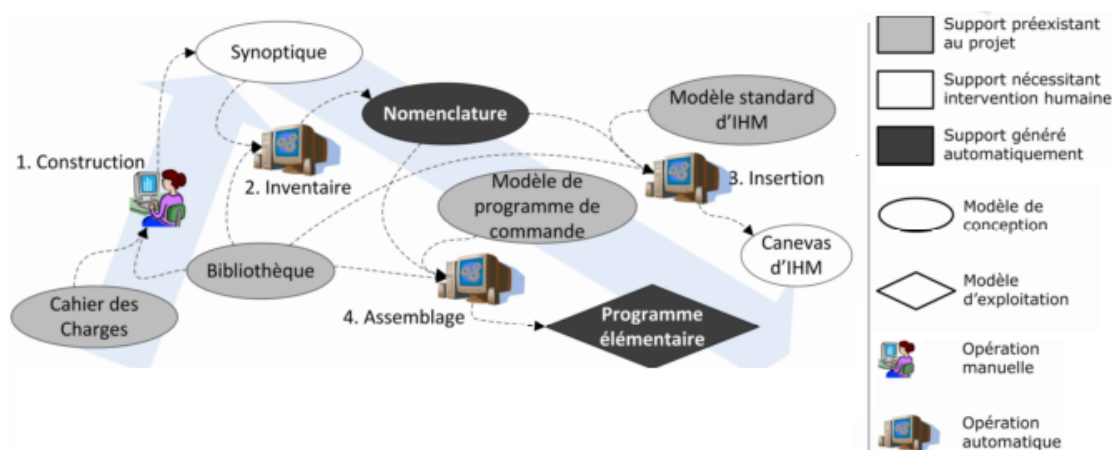


FIGURE 1.7 – Flot de conception Anaxagore

1.4 Problématiques et verrous scientifiques

1.4.1 Problématiques

La génération automatique des interfaces de supervision et des programmes de commande par une approche mixte (ascendante et descendante) repose sur une bibliothèque de composants standards et un modèle abstrait du système. Cette génération automatique assure une cohérence entre le modèle abstrait et les modèles raffinés (générés automatiquement) et réduit le temps de conception. Néanmoins, une génération automatique efficace n'a d'intérêt que si les modèles abstraits et les composants standards ont les qualités requises.

Souvent les composants standards utilisés dans une approche ascendante sont validés par des tests empiriques. Une vérification formelle et exhaustive du bon fonctionnement de ces composants est donc nécessaire pour garantir que le système global (interface de supervision et programme de commande) généré à partir de ces composants est correct.

Les pratiques actuelles dans l'industrie pour l'élaboration des modèles d'architectures (dans une approche ascendante) sont informelles et manuelles. Ces modèles doivent être vérifiés afin de s'assurer de leur cohérence, leur complétude et de leur respect du cahier des charges. Cette vérification est souvent réalisée manuellement par les experts, ce qui conduit à des erreurs détectées tardivement dans la phase de tests, lorsque le système est en grande partie implémenté. Les erreurs introduites dans les modèles d'architectures peuvent être propagées aux codes des interfaces de supervision et des programmes de commande générés automatiquement à partir de ces modèles. Une vérification formelle et exhaustive des modèles d'architectures permettrait de vérifier mathématiquement, que l'architecture du système est cohérente et complète. La vérification au niveau architectural, permettrait de réduire sensiblement les coûts et les erreurs [Medvidovic et Taylor 2000]. Elle permettrait, aussi, de garantir la qualité des codes générés automatiquement [Clarke et Wing 1996].

Une étude, réalisée sur plusieurs projets industriels, montre que la qualité des systèmes est améliorée par l'utilisation des méthodes formelles. En effet, les méthodes formelles permettent la détection des erreurs, l'amélioration de la conception, l'augmentation de la confiance dans l'exactitude, l'identification précoce des défauts et offrent une meilleure compréhension [Woodcock et al. 2009]. Ces études montrent, aussi, que les intervenants en général, étaient satisfaits des techniques formelles utilisées dans leurs projets [Woodcock et al. 2009]. Pour cela, les travaux, réalisés dans le cadre de cette thèse, consistent à introduire dans les premières phases d'une approche mixte de génération automatique des programmes de commande et des IHM de supervision, un ensemble de techniques de vérification par méthodes formelles, que ce soit au niveau des exigences ou des bonnes propriétés des modèles de conception. Ces techniques de vérification permettront de proposer plus rapidement et à moindre coût des systèmes plus sûrs et répondant aux exigences des utilisateurs avec une meilleur qualité.

1.4.2 Verrous scientifiques

Afin d'introduire dans une approche de conception mixte les étapes de la vérification formelle, plusieurs verrous scientifiques sont à lever. Dans cette section, nous énumérons ces verrous.

Verrou 1 : Que vérifier ? La définition des propriétés à vérifier est une étape indispensable pour l'intégration de la vérification formelle. En effet, le type de propriétés à vérifier conditionne le choix des modèles formels [Traonouez 2009], la méthode de vérification et l'outil de vérification. Par conséquent, il est nécessaire de définir dans un premier temps les propriétés auxquelles le modèle d'architecture et aussi les composants de la bibliothèque doivent

satisfaire. Donc, le verrou porte ici sur la définition des exigences à vérifier sur les modèles d'architecture et aussi les composants de la bibliothèque standard.

Verrou 2 : Comment vérifier ? Le deuxième verrou de nos travaux consiste à déterminer la méthode de vérification formelle la plus adéquate afin de vérifier les propriétés identifiées (verrou 1).

Les techniques de vérification formelle sont basées sur la spécification du système et les exigences à vérifier dans un langage formel. Ce verrou porte sur :

- La définition du langage formel pour modéliser formellement le modèle d'architecture et les composants standards,
- La définition du langage formel pour la formalisation des propriétés à vérifier,
- La définition de la méthode de vérification la plus adéquate pour vérifier les différentes propriétés sur les modèles formels.

Le choix de la méthode formelle à utiliser doit tenir compte des avantages et des inconvénients de chaque méthode. En effet, le *Model-Checking* est très apprécié dans un contexte industriel car la vérification est automatique. Mais, il souffre du problème de l'explosion combinatoire du nombre des états. Le *Theorem-Proving* ne souffre pas d'explosion combinatoire, mais, cette technique n'est pas complètement automatique. Il conviendra donc de choisir la méthode formelle qui permet d'intégrer la vérification formelle, tout en contournant ses inconvénients.

Verrou 3 : Formalisation. Une fois les modèles, les exigences à vérifier, les langages formels et la méthode formelle définis, il sera question de procéder à la formalisation. Le modèle abstrait et les composants de la bibliothèque doivent être modélisés dans les langages formels définis en réponse au verrou 2. Le verrou ici, porte sur le degré de granularité de la modélisation. En effet, la modélisation doit être assez concrète afin de vérifier les propriétés à un degré de granularité proche du système réel. En même temps, la modélisation doit être assez abstraite pour pouvoir appliquer les méthodes formelles [Bowen et Hinchey 1995; 2006].

Verrou 4 : Automatisation de la démarche. Le caractère mathématique des méthodes formelles et la difficulté de leur prise en main limitent leur utilisation dans le monde industriel. En effet, des études ont montré que même les utilisateurs expérimentés dans les méthodes formelles trouvent que les spécifications formelles sont faciles à lire et à comprendre mais difficiles à écrire [Snook et Butler 2001]. Donc, il est très utile d'assister les concepteurs pour conduire la vérification formelle. Ainsi, les ingénieurs métiers bénéficieront du pouvoir de ces méthodes dans la détection exhaustive des erreurs sans avoir besoin de connaissances en vérification formelle. Le verrou porte sur l'obtention automatique des modèles formels, définis dans le verrou 3, à partir des modèles d'architecture et des composants standards.

1.5 Conclusion

Ce chapitre a introduit et posé les contextes dans lesquels s'inscrit cette thèse pour en préciser les problématiques. Les verrous de cette thèse portent essentiellement sur l'intégration de la

vérification formelle dans un projet industriel complexe. La vérification formelle permet d'apporter une preuve mathématique solide qu'un système satisfait un ensemble de propriétés. Pour intégrer la vérification formelle dans un projet, plusieurs critères sont à prendre en compte. Ces critères portent essentiellement sur le domaine d'application, la facette du système étudié, le type des exigences à vérifier et la méthode formelle la plus adéquate. Nous essayons dans le chapitre 2 de définir, à travers une étude de la littérature, les pratiques actuelles de la communauté scientifique dans le domaine de la vérification formelle.

2

État de l'art : *systematic mapping*

Résumé : Dans ce chapitre, nous présentons l'état de l'art scientifique de la spécification et de la vérification formelle. Cette étude bibliographique a été conduite en utilisant de nouvelles méthodes telles que la *systematic mapping*. Nous présentons le protocole suivi pour conduire cette revue ainsi que les résultats obtenus.

Sommaire

2.1	Introduction	32
2.2	État de l'art au travers des revues de la littérature existante	32
2.2.1	Classification des langages formels	32
2.2.2	Comparaison des langages de spécification	34
2.2.3	Utilisation des méthodes formelles	34
2.2.4	Bilan sur les revues de la littérature existantes	35
2.3	Méthode	35
2.4	Étape 1 : Définition du protocole	36
2.4.1	Définition des questions de recherche	36
2.4.2	Stratégie de la recherche	37
2.4.3	Sélection des articles	38
2.4.4	Extraction des données et classification	38
2.4.5	Évaluation de la validité de l'étude	39
2.5	Étape 2 : Conduction	40
2.6	Étape 3 : Rapport de synthèse	42
2.6.1	Résultats démographiques	42
2.6.2	Proposition d'une classification des langages formels	43
2.7	Résultats Obtenus	46
2.7.1	QR1 : Quels sont les langages de spécification utilisés pour la vérification formelle ?	46
2.7.2	QR2 : Comment a été réalisée la vérification formelle ?	50
2.7.3	QR3 : Quels sont les objectifs de la vérification formelle ?	51
2.8	Conclusion et Discussions	56
2.8.1	Bilan	56
2.8.2	Propositions	56

2.1 Introduction

A travers cette thèse, nous souhaitons intégrer la vérification formelle dans un flot de conception automatisé. Cependant, pour utiliser les méthodes formelles, un ensemble de défis doit être levé. Ces derniers sont formulés en questions et présentés comme suit :

1. Quelles sont les exigences qui peuvent être vérifiées formellement ?
2. Quel langage choisir pour la spécification de ces exigences ?
3. Quel langage choisir pour la spécification du système ?
4. Quelle méthode formelle est la mieux adaptée pour la vérification ?
5. Quel outil choisir pour mettre en œuvre la méthode ?

Afin de trouver des réponses à ces questions, nous avons parcouru différentes revues de la littérature que nous présentons dans la section suivante. Il est vrai que l'étude de ces revues, nous a permis d'acquérir les concepts et les terminologies du domaine de la vérification formelle. Cependant, nous n'avons pas trouvé de réponses à toutes nos questions. Nous avons alors décidé de compléter cet état de l'art, par une *systematic mapping*.

Dans les sections suivantes, nous présentons un aperçu de ces différentes revues que nous avons classées en trois catégories. Puis, nous discutons de leurs limites.

2.2 État de l'art au travers des revues de la littérature existante

Dans la littérature, plusieurs études et revues ont été réalisées sur la spécification et/ou la vérification formelle. Ces études traitent : soit la classification des différents langages et/ou méthodes formelles ; soit la comparaison de ces langages et/ou méthodes ; ou bien, le retour d'expérience de l'utilisation de ces méthodes.

2.2.1 Classification des langages formels

Plusieurs classifications ont été proposées afin de catégoriser les langages formels selon plusieurs critères. Nous les présentons, ci-après, par ordre chronologique (figure 2.1).

1. [Wing \[1990\]](#) a proposé une classification des langages de spécification formels selon la structure du système spécifié. Les langages ont été ainsi classés en langages basés sur les propriétés et langages basés sur les modèles.
2. [Barroca et McDermid \[1992\]](#) proposent de classer les langages formels en cinq catégories :
 - Langages basés sur les modèles comme Z et VDM,
 - Langages algébriques comme OBJ [[Futatsugi et al. 1985](#)] et PLUSS [[Gaudel 1992](#)],
 - L'algèbre de processus comme CSP [[Hoare et al. 1985](#)] et CCS [[Milner 1989](#)],
 - Langages basés sur les logiques comme les LTL et CTL,

- Langages basés sur les réseaux comme les réseaux de Petri.
3. [Lamsweerde \[2000\]](#), quant à lui, trouve que la classification basée sur les modèles/propriétés est confuse, car dans les systèmes réels, les deux classes (basée sur les modèles et basée sur les propriétés) sont utilisées conjointement. Il propose, alors, une nouvelle classification, cette fois-ci, basée sur l'expressivité du langage. Ainsi, les langages formels sont classés en :
- Langages basés sur l'historique comme les logiques temporelles,
 - Langages basés sur les états comme Z et B,
 - Langages basés sur les transitions comme les automates,
 - Langages fonctionnels qui peuvent être algébriques, comme OBJ [\[Futatsugi et al. 1985\]](#) et ASL [\[Astesiano et Wirsing 1987\]](#), ou bien d'ordre supérieur comme HOL [\[Gordon et Melham 1993\]](#) et PVS,
 - Langages opérationnels comme les réseaux de Petri et l'algèbre de processus.
4. [Almeida et al. \[2011\]](#) classent les langages formels en : langages basés sur les modèles, langages algébriques et langages déclaratifs. Les langages basés sur les modèles sont divisés en :
- Langages basés sur les états abstraits comme ASM et B,
 - Langages basés sur les ensembles et les catégories comme Alloy, Z et VDM,
 - Automates,
 - Langages pour les systèmes temps réel, comme Lustre [\[Halbwachs et al. 1991\]](#) et les automates temporisés.
5. [Alagar et Periyasamy \[2011\]](#) ont repris la classification de [Wing \[1990\]](#) et ont ajouté les logiques à la classe des langages basés sur les propriétés (figure 2.1).

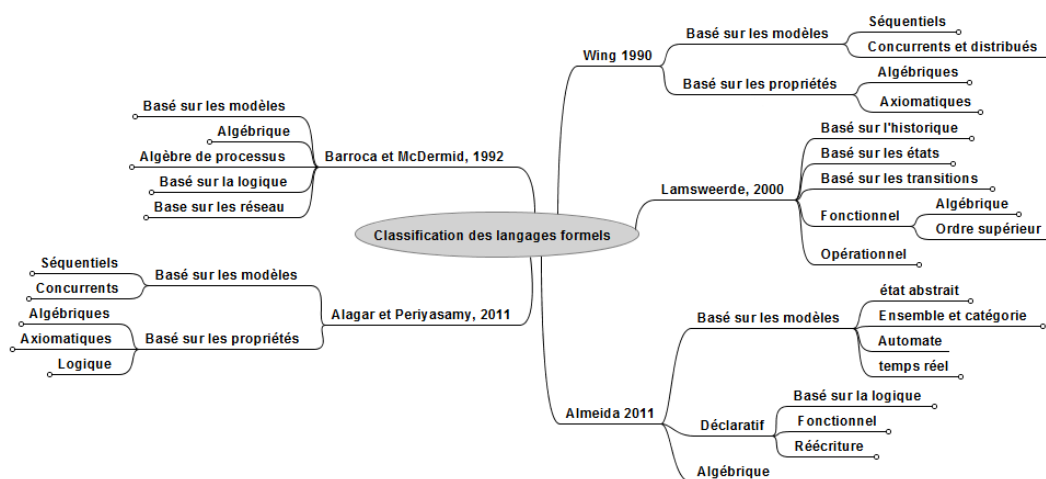


FIGURE 2.1 – Classifications existantes des langages formels

Plusieurs langages ont émergé ces dernières années. Ces langages sont développés pour satisfaire de nouveaux besoins dans la modélisation. En effet, les langages stochastiques (basés sur les probabilités), par exemple, ne figurent dans aucune de ces classifications. Nous déduisons, donc, que les classifications citées ci-dessus sont incomplètes. Une nouvelle classification des langages formels permettra de compléter les classifications existantes.

2.2.2 Comparaison des langages de spécification

Dans cette section, nous présentons les études où les auteurs proposent des comparaisons et des évaluations entre deux ou plusieurs langages de spécification.

Kossak et Mashkoor [2016] ont développé un système de comparaison, constitué de cinq critères, pour évaluer la pertinence de six langages formels : Alloy, ASM, B, Event-B, TLA+, Z et VDM, dans les développements industriels. Ces langages sont évalués selon : 1) leur capacité de modélisation ; 2) la phase du développement supportée ; 3) leur capacité technique (outil, traçabilité...) ; 4) leur utilisabilité et 5) leur applicabilité industrielle.

Newcombe [2014] a comparé et étudié l'applicabilité des deux langages formels TLA+ et Alloy dans la vérification des systèmes de taille réelle. De leur étude, il en ressort que TLA+ répond à des exigences industrielles et peut être utilisée dans la vérification formelle de grands projets. Tse et Pong [1991] proposent une étude basée sur plusieurs critères, afin de comparer les langages de spécifications suivants : HOS¹ [Hamilton et Zeldin 1976], SAMM² [Lamb et al. 1978], EDDA [Trattnig et Kerner 1980], SADT, PSL³ [Teichroew et al. 1980] et RSL⁴ [Alford 1978].

Les études présentées ci-dessus proposent un ensemble de critères pour l'évaluation et la comparaison des langages de spécification formels. Les résultats de ces études offrent un support et un guide pour les futurs utilisateurs des méthodes formelles. En revanche, les comparaisons ont été réalisées sur des ensembles réduits de langages formels. Ces évaluations ne peuvent pas être généralisées aux autres langages formels. Une étude plus large, sur un ensemble plus large de langages formels, permettrait de compléter et de généraliser ces résultats.

2.2.3 Utilisation des méthodes formelles

Clarke et Wing [1996] ont étudié les différentes techniques de spécification et de vérification formelles à travers plusieurs projets. Ils ont discuté les apports et les différents obstacles pour leur adoption dans l'industrie. Woodcock et al. [2009] ont mené une étude sur 62 projets industriels où les méthodes formelles ont été utilisées. Cette étude a été réalisée à travers des questionnaires avec les différents intervenants de ces projets. Les résultats de cette étude montrent que les méthodes formelles augmentent la qualité des produits et réduisent les temps

-
1. HOS : Higher Order Software
 2. SAMM : Systematic Activity Modelling Method
 3. PSL : Problem Statement Language
 4. RSL : Requirements Statement Language

et les coûts de développement. Bjørner et Havelund [2014] ont reporté leurs expériences dans l'utilisation des méthodes formelles. Ils présentent plusieurs méthodes formelles et discutent les obstacles qui limitent l'adoption de ces méthodes dans l'industrie. Parmi ces obstacles, on note la difficulté de la prise en main des méthodes formelles et l'immaturation des outils formels. On arrive au même constat résultant des expérimentations menées par Bennion et Habli [2014]. Bowen et Hinchey [1995] proposent un ensemble de dix recommandations pour que les méthodes formelles soient bien appliquées dans le monde industriel. Ces recommandations ont été mises à jour dix ans plus tard, par les mêmes auteurs [Bowen et Hinchey 2006]. Parmi ces recommandations, nous citons par exemple : le choix du langage formel approprié, le niveau d'abstraction de la modélisation, la documentation et la réutilisation.

Ces études permettent d'avoir un aperçu pertinent de l'utilisation réelle des méthodes à travers des projets industriels. Elles permettent aussi de faire ressortir les obstacles limitant l'utilisation des méthodes formelles dans l'industrie. L'utilisation de méthodes formelles dans le monde académique peut aussi révéler des pratiques, des obstacles, et par conséquent, d'autres recommandations d'utilisation.

2.2.4 Bilan sur les revues de la littérature existantes

Les études citées dans la section 2.2 se limitent à présenter et/ou comparer plusieurs méthodes de vérification ou langages de spécification. Nous pensons qu'une étude approfondie, portant sur les fréquences réelles d'utilisation des différents langages de spécification et de méthodes formelles réalisées dans différents domaines, permettrait de faire ressortir les différentes pratiques du monde académique et industriel en vérification formelle. Nous pensons que l'utilisation des techniques de la *systematic mapping* permet de cadrer l'étude, de limiter le biais des chercheurs et d'augmenter la qualité des résultats. À notre connaissance, aucune revue systématique portant sur l'utilisation de la vérification formelle n'a été réalisée. Pour tout cela, nous adoptons la méthode de la revue systématique pour conduire cette étude.

Le but de notre étude est de définir clairement les liens entre la spécification et les vérifications formelles des logiciels. L'objectif est d'avoir une cartographie des langages de spécification, des méthodes formelles et des exigences vérifiées. Cette cartographie permettrait d'avoir une vision générale et synthétique de l'état de l'art dans le domaine des méthodes formelles et leurs utilisations.

2.3 Méthode

La *systematic mapping*, proposée par *Evidence-Based Software Engineering* (EBSE), a pour but de recenser, d'étudier et de conduire de manière systématique et contrôlée, l'état de l'art dans un domaine scientifique particulier [Keele 2007, Brereton et al. 2007, Kitchenham et al. 2009]. Le processus de la *systematic mapping* se déroule en trois grandes étapes (voir figure 2.2).

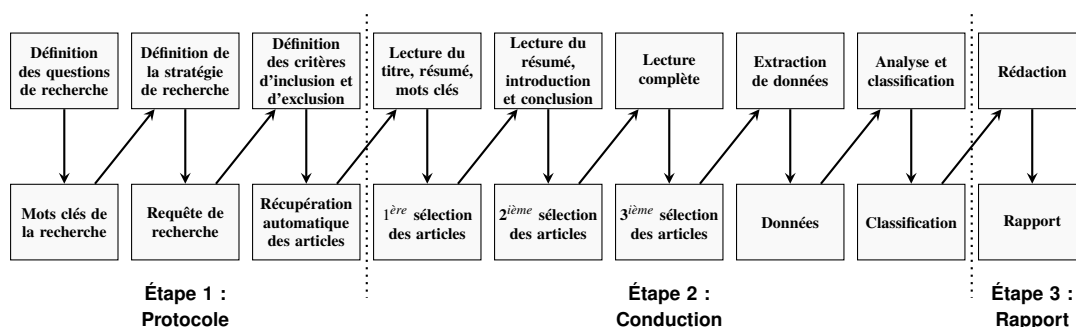


FIGURE 2.2 – Le processus de la revue systématique

La revue systématique est initiée par une première étape qui se caractérise par la définition d'un protocole [Keele 2007]. Ce dernier a pour rôle de déterminer le plan et la méthodologie à suivre pour l'élaboration de l'étude. Le protocole comprend la définition des questions de recherche, les modalités de recherche et d'extraction de données, ainsi que les critères d'inclusion et d'exclusion. Cette première étape est suivie d'une étape dite "conduction", durant laquelle, le protocole est appliqué. À la fin de cette étape, un ensemble d'articles pertinents est sélectionné ; il constituera par la suite la base pour l'extraction de données. La revue se termine par une dernière étape, où les données sont synthétisées à travers les techniques statistiques et rapportées sous forme d'un rapport.

Les objectifs de notre revue systématique ciblent trois grands axes : Les méthodes formelles, les langages de spécifications et le type de propriétés à vérifier. Nous présentons dans les sections suivantes : le protocole, la conduction et les résultats de notre étude.

2.4 Étape 1 : Définition du protocole

L'objectif d'une revue systématique est d'étudier la littérature d'une manière planifiée et non biaisée [Keele 2007] suivant un protocole de recherche bien défini. Le protocole est le plan suivi par tous les participants durant toute l'étude. Il permet de garantir la qualité et l'homogénéité des données. Il est initié par la définition des objectifs de la revue sous forme de questions de recherche auxquelles la revue systématique doit répondre. Les stratégies de recherche sont ensuite définies et portent généralement sur le choix des bases d'articles, les critères d'inclusion et d'exclusion [Keele 2007] et les règles de sélections et de gestion de conflit entre les chercheurs.

2.4.1 Définition des questions de recherche

Nous avons défini trois questions principales pour mener notre recherche. Ces questions sont ensuite affinées en sous questions comme suit :

QR1 : Quels sont les langages de spécification utilisés pour la vérification formelle ?

- QR1.1 : Quels langages ont été utilisés pour la modélisation du système ?
- QR1.2 : Quels langages ont été utilisés pour la spécification des exigences ?

QR2 : Comment a été réalisée la vérification formelle ?

- QR2.1 : Quelles sont les méthodes formelles utilisées ?
- QR2.2 : Ces techniques sont-elles supportées par un outil ?

QR3 : : Quels sont les objectifs de la vérification formelle ?

- QR3.1 : Quelles sont les exigences vérifiées par ces méthodes ?
- QR3.2 : Dans quel domaine la vérification formelle a été réalisée ?
- QR3.3 : Quelles facettes du système ont été vérifiées ?
- QR3.4 : Dans quels contextes (académique, industriel) la vérification a été réalisée ?
- QR3.5 : Quel est le but général de la vérification ?

2.4.2 Stratégie de la recherche

Nous avons effectué une recherche automatique des articles en interrogeant quatre bases d'articles : IEEE Xplore⁵, ACM Digital Library⁶, ScienceDirect⁷, SpringerLink⁸. Pour construire la requête de recherche, nous avons identifié, à partir des questions de recherche, les principaux mots clés (figure 2.2). Ces mots clés sont : *requirements*, *specification*, *verification*. La requête de recherche est construite en combinant ces mots clés et leurs synonymes (différentes variantes utilisées dans la littérature) par des connecteurs logiques (AND/OR). Ensuite, afin de limiter l'étude aux travaux dans le domaine du développement logiciel, des restrictions ont été ajoutées à la requête. Ces restrictions excluent tous les travaux portant sur : le développement matériel, les circuits électroniques, le web, les services, les agents et les tests. À la fin, nous avons obtenu la requête générique suivante :

```
(("requirements specification" OR "requirements modeling" OR
"requirements modelling" OR "requirements languages" OR "specification
languages") AND ("formal verification") NOT (network OR test OR testing
OR hardware OR web OR service OR circuit OR agent))
```

La requête a été adaptée à chacune des bases comme le montre le tableau A.1 dans l'annexe A. Nous avons récupéré tous les articles publiés entre 1990 et décembre 2014 (date de la recherche des articles).

5. IEEE Xplore : <http://ieeexplore.ieee.org/Xplore/home.jsp>

6. ACM Digital Library : <http://dl.acm.org/>

7. ScienceDirect : <http://www.sciencedirect.com/>

8. SpringerLink : <http://link.springer.com/>

2.4.3 Sélection des articles

Pour pouvoir sélectionner les articles, un ensemble de critère d'inclusion et d'exclusion a été élaboré.

Critères d'inclusion. Tout article publié entre 1990 et fin 2014, qui traite la vérification formelle est accepté.

Critères d'exclusion. L'exclusion est appliquée sur chaque article qui :

- présente un état de l'art ou une classification : des exigences, des méthodes de vérification formelle, ou bien, des techniques de spécification ;
- présente une opinion, un avis, un poster, un keynote, tutoriel, sommaire d'une conférence, ou a moins de 4 pages ;
- présente un outil ;
- est écrit dans une autre langue que l'anglais, ou bien, s'il est indisponible ;
- est dans le domaine des réseaux ou du matériel ;
- est incomplet, c.à.d. dans le cas où plusieurs versions d'un article existent, la version la plus complète est choisie.

2.4.4 Extraction des données et classification

Pour extraire les données, nous avons défini une fiche de lecture qui fixe les données à extraire de chaque article (voir tableau 2.1).

ID	Catégorie	Terme	QR
T1	Générale	Auteurs	Documentation
T2		Année	Documentation
T3		Titre	Documentation
T4		Venue	Documentation
T5		Citation	Documentation
T6	Spécification (QR1)	Modélisation du système	QR1.1
T7		Type du langage en T5	QR1.1
T8		Transformation de langage	QR1.1
T9		Type du langage en T7	QR1.1
T10		Formalisation des propriétés	QR1.2
T11		Type du langage en T9	QR1.2
T12	Vérification (QR2)	Méthode	QR2.1
T13		Outil	QR2.2
T14	Objectifs (QR3)	Propriétés vérifiées	QR3.1
T15		Domaine	QR3.2
T16		Vue du système	QR3.3
T17		Contexte	QR3.4
T18		But général	QR3.5

Tableau 2.1 – Fiche de relecture pour l'extraction de données

Nous avons utilisé l'outil StArt⁹ (State of the Art through Systematic Review) [Start] pour l'extraction de données.

La catégorie "Générale" regroupe les données récoltées pour des fins de documentation. Ces données concernent : la liste des auteurs (T1), l'année de publication (T2), le titre de l'article (T3), sa source (T4) et le nombre de citation (T5). Le reste des données à extraire se divise en trois axes : la spécification des exigences, la vérification formelle et les objectifs de la vérification formelle.

Pour la spécification (catégorie "Spécification" dans le tableau 2.1), on extrait le langage de spécification utilisé pour la modélisation du système (T6) et son type (T7) qui peut être : formel, semi-formel ou textuel . Si le langage est semi-formel ou textuel, une transformation de celui-ci vers un langage formel est nécessaire. Dans (T8), on recense le langage formel résultant de cette transformation et son type (T9). Le langage utilisé pour la modélisation des propriétés à vérifier est aussi extrait (T10), ainsi que son type (T11).

Sur l'axe de la vérification formelle (catégorie "vérification" dans le tableau 2.1), on cherche à déterminer la méthode de vérification formelle utilisée (T12) et l'outil de la vérification formelle (T13).

Sur le dernier axe, les données liées aux différents objectifs de la vérification formelle sont extraites (catégorie "Objectifs" dans le tableau 2.1). On détermine les propriétés vérifiées (T14), le domaine d'application (T15), la facette du système concernée par la vérification formelle (T16) et le contexte (T17) qui peut être industriel ou académique. On détermine aussi le but général de la vérification (T18), qui peut être : soit la génération de code sûr à partir d'un modèle formel vérifié ; ou bien, la vérification d'un code déjà existant ; comme il peut être juste un objectif de modélisation.

2.4.5 Évaluation de la validité de l'étude

Nous avons présenté dans les sections précédentes les stratégies prises pour conduire la *systematic mapping*. Nous discuterons, dans cette section, des limites de cette étude. Dans une *systematic mapping* plusieurs menaces peuvent affecter sa validité [Petersen et Gencel 2013, Petersen et al. 2015]. Ces menaces sont présentées comme suit :

La validité descriptive porte sur l'objectivité et la précision des observations [Petersen et al. 2015]. L'objectivité ici concerne l'extraction de données. Cette menace est sous contrôle [Petersen et al. 2015] par l'utilisation de fiches d'extraction de données, qui sont définies dans le protocole (avant que la revue ne soit réalisée).

La validité théorique est déterminée par la capacité à extraire ce que l'on veut capturer [Petersen et al. 2015]. Les stratégies de recherche définies peuvent menacer la validité théorique de notre étude. En effet, nous avons décidé d'exclure tous les articles publiés avant 1990 ou écrits dans d'autres langues que l'anglais, les rapports de recherche et les thèses. En appliquant ces critères, il est possible d'avoir omis des articles pertinents. Nous avons aussi opté

9. Start : http://lapes.dc.ufscar.br/tools/start_tool

pour une recherche automatique des articles en utilisant quatre bases. Il est possible d'avoir omis des articles pertinents qui ne sont pas indexés par les bases interrogées. Néanmoins, la recherche automatique des articles reste l'approche la plus utilisée dans les *systematic mapping* [Petersen et al. 2015]. La première étape de sélection est basée sur le titre et les mots clés de chaque article. Comme il n'existe pas de guide pour le choix des mots clés, il est possible d'avoir éliminé des articles à base de mots clés qui ne sont pas significatifs. Une autre menace à la validité de notre étude est l'absence de l'évaluation de la qualité, car cette dernière est compliquée [Kitchenham et al. 2011], dans le cas d'une *systematic mapping* et n'est pas essentielle [Kitchenham et al. 2011]. Les préjugés (**biais**) des chercheurs lors de la sélection et l'extraction de données est une vraie menace à la validité. Pour éviter cela, nous avons défini le protocole dans la première phase. Le protocole a été discuté et validé par tous les participants à l'étude. L'extraction de données a été réalisée par un doctorant puis vérifiée et validée par des chercheurs expérimentés. Dans le cas d'une revue contenant un nombre important d'articles, Brereton et al. [2007] proposent de diviser le rôle des lecteurs en *extracteur*, qui se charge de l'extraction de données, et en *vérificateur*, qui vérifie les données extraites. Nous avons adopté cette démarche lors de l'extraction de données.

La validité interprétative de la conclusion peut être menacée si cette dernière ne reflète pas les données traitées [Petersen et al. 2015]. L'interprétation des données est à la charge des chercheurs et de leurs jugements. Pour réduire le biais des chercheurs, l'interprétation des données a été discutée par les différents chercheurs avant d'être reportée.

La généralisabilité peut être interne (au sein du même groupe) ou externe (entre les groupes) [Petersen et Gencel 2013]. Dans notre étude, la généralisabilité interne est assurée par la diversité des travaux réalisés dans différents domaines. La généralisabilité externe est une menace dans la mesure où nos résultats ne peuvent pas être généralisés au domaine du matériel ou des réseaux. Par contre, le protocole peut être utilisé pour conduire des revues systématiques dans ces domaines.

La répétabilité exige que le processus de la *systematic mapping* soit bien détaillé pour qu'il puisse être réutilisé. Pour faciliter cette répétabilité, nous avons reporté le processus dans ce chapitre et nous avons utilisé l'outil *Start* pour conduire l'étude. Cet outil stocke toutes les données de chaque étape. Nous avons mis les données sur le site de l'entreprise pour leur utilisation future et pour qu'une vérification ultérieure de celles-ci soit possible.

2.5 Étape 2 : Conduction

Après avoir défini et validé le protocole par trois professeurs et deux ingénieurs industriels, nous avons conduit la *systematic mapping*. Les informations, liées à l'étape de conduction, sont résumées dans le tableau 2.2.

Premièrement, un ensemble de 513 articles a été récupéré en appliquant les requêtes (tableau A.1 dans l'annexe A) sur les différentes bases d'articles en décembre 2014. 344 (67%) des articles ont été extraits de IEEE Xplore, 117 (23%) de SpringerLink, contre seulement 31

(6%) d'ACM et 21 (4%) de ScienceDirect (tableau 2.2). Ces articles ont été ensuite importés dans le logiciel StArt. Cet outil permet de détecter et de supprimer les articles doubles. 21 doublons ont été ainsi supprimés, ce qui représente 4% de l'ensemble des articles récupérés initialement. Ensuite, nous avons conduit la sélection des articles, qui s'est déroulée en trois étapes (cf. les trois sélections du tableau 2.2). Dans chaque étape, les critères d'inclusion et d'exclusion ont été appliqués. Chaque article a été lu initialement par deux chercheurs. Dans le cas d'un conflit, l'article est relu par un troisième chercheur. L'article est inclus ou exclu selon l'avis de la majorité (2 chercheurs sur 3).

Source	Récupéré	Nombre d'articles			Indexation	Précision
		sélection 1	sélection 2	sélection 3		
IEEE	344 (67%)	193	96	62	76%	18%
ACM	31 (6%)	16	4	4	5%	13%
ScienceDirect	21 (4%)	11	7	4	5%	19%
SpringerLink	117 (23%)	41	14	11	14%	9%
Total	513	261	121	81	100%	-

Tableau 2.2 – Provenance des articles et sélections

Lors de la première étape de sélection, l'inclusion et l'exclusion des articles sont déterminées après la lecture du titre, du résumé et des mots clés. À la fin de cette première étape, 252 (49%) articles ont été éliminés et 261 (51%) articles ont été retenus (voir tableau 2.2). Ces 261 articles ont fait l'objet d'une deuxième sélection basée cette fois-ci sur la lecture du résumé, de l'introduction et de la conclusion. À la fin de cette deuxième étape, l'ensemble d'articles acceptés a été réduit à 121 articles (tableau 2.2). La lecture entière de chaque article, lors de la troisième et dernière étape de sélection, a permis de réduire le nombre d'articles retenus pour l'étude à 81, ce qui représente 16% des 513 articles sélectionnés initialement. La figure 2.3 résume l'évolution de l'ensemble des articles pertinents et les différentes étapes de sélections.

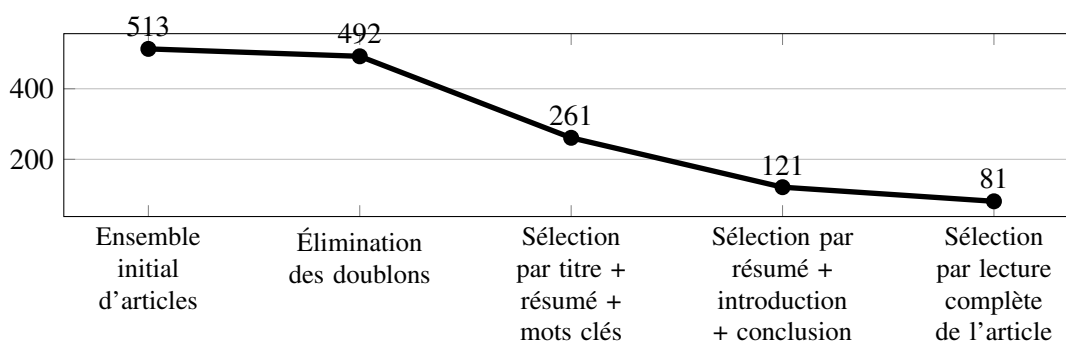


FIGURE 2.3 – Évolution de la sélection des articles pertinents

2.6 Étape 3 : Rapport de synthèse

Dans cette section, nous décrivons premièrement quelques résultats démographiques de la revue. Ensuite, nous présentons notre proposition de classification des langages formels.

2.6.1 Résultats démographiques

La majorité des articles retenus pour cette étude proviennent principalement des conférences (75% sur la figure 2.4), contre 14% des journaux et, seulement 11% des chapitres. Ces articles ont été publiés entre 1990 et 2014. La figure 2.5 montre un pic de publications pour les années 2001, 2004, 2005 et 2012.

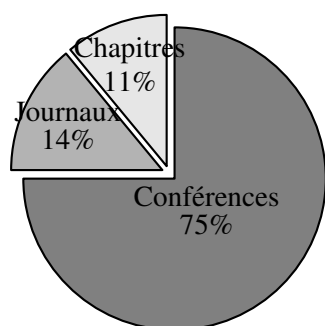


FIGURE 2.4 – Provenance des articles

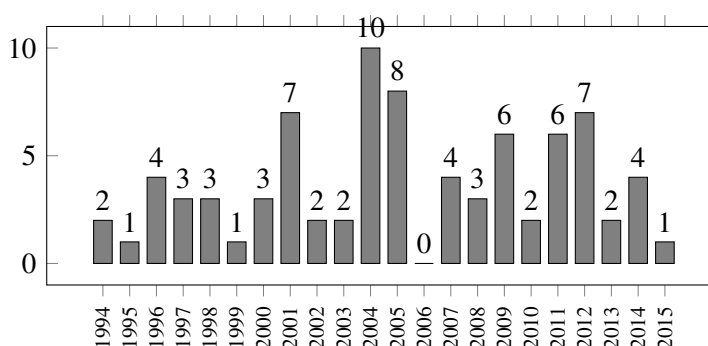


FIGURE 2.5 – Nombre d'articles par an

Pour chaque base d'articles, nous avons calculé les taux d'indexation et de précision par les formules respectives (2.1) et (2.2). Ces taux permettent de donner un aperçu sur le pouvoir d'indexation et de précision de chaque base (tableau 2.2).

$$\text{Indexation (\%)} = \frac{\text{Nombre d'articles pertinents récupérés à partir de la base}}{\text{Nombre total d'articles pertinents}} \times 100 \quad (2.1)$$

$$\text{Précision (\%)} = \frac{\text{Nombre d'articles pertinents récupérés à partir de la base}}{\text{Nombre des articles récupérés à partir de la base}} \times 100 \quad (2.2)$$

Les résultats montrent que IEEE Xplore est la base la mieux indexée car 76% des articles pertinents proviennent de cette base (tableau 2.2). ACM et ScienceDirect sont les bases les moins indexées avec un taux d'indexation de 5%. Les résultats montrent aussi que ScienceDirect et IEEE Xplore ont les meilleures précisions avec respectivement un pourcentage de 19% et de 18%. SpringerLink reste la base la moins précise avec un taux de précision égal à 9%. Nous déduisons que IEEE Xplore reste la base la mieux indexée et la plus précise.

Nous avons construit un nuage de mots à partir des mots clés des 81 articles (figure 2.6). Le nuage montre que les mots clés tels que : Système (190), langage (178), formel (176), spécification (156), vérification (137), logiciel (99), modélisation (84), contrôle (65) et logique (51) apparaissent plus de 50 fois. Ce nuage de mots permet d'illustrer une certaine correspondance entre les articles retenus et les questions de recherche.

Le tableau 2.3 illustre le nombre d'articles ayant un nombre de citations entre 5 et 100. 25% des articles sont cités plus de 30 fois et 40% des articles sont cités plus de 10 fois. Ces résultats montrent la qualité de notre étude.



FIGURE 2.6 – Nuage des mots clés

Citation	Nombre	Pourcentage
≥ 100	3	4%
≥ 50	7	9%
≥ 30	20	25%
≥ 20	26	32%
≥ 10	32	40%
≥ 5	47	58%

Tableau 2.3 – Citations

En fonction des résultats obtenus et de l'état de l'art des classifications existantes, nous proposons une nouvelle classification des langages formels. En effet, les classifications existantes ne nous permettent pas de classer tous les langages formels recensés dans cette revue.

2.6.2 Proposition d'une classification des langages formels

Pour pouvoir classer les langages formels, nous avons combiné et adapté les classifications de Almeida et al. [2011] et de Lamsweerde [2000]. Nous avons gardé la classification : langages basés sur les modèles et langages basés sur les propriétés. Néanmoins, les résultats de la *systematic mapping* nous ont permis de raffiner chaque catégorie. Cette nouvelle classification est illustrée dans la figure 2.7.

2.6.2.1 Langages basés sur les modèles

Les langages basés sur les modèles permettent de décrire le système à travers ses états internes [Almeida et al. 2011]. La notion d'état est centrale dans ce type de langage. Les langages basés sur les modèles sont ainsi classés en :

Langages basés sur les états. Cette classe de langage peut être aussi divisée en deux groupes : les langages basés sur les états abstraits et les langages basés sur les ensembles. Les langages basés sur les états abstraits comme ASM et B permettent de décrire le système à travers ses états abstraits ainsi que les conditions qui permettent au système de passer d'un

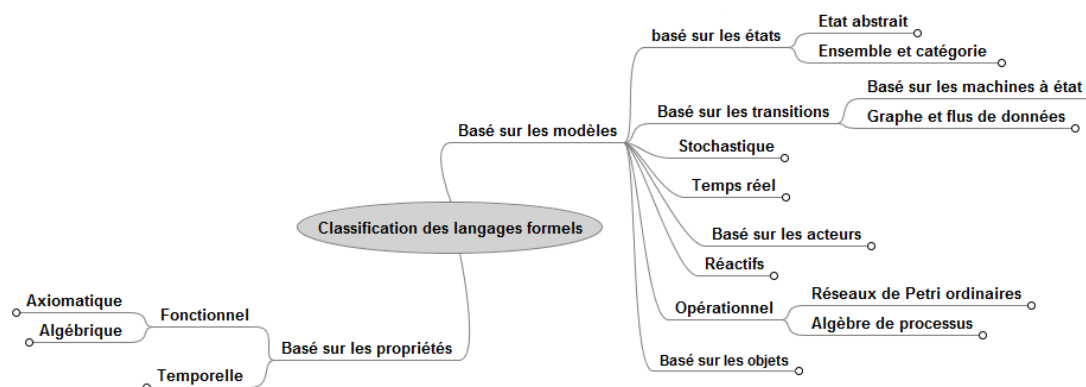


FIGURE 2.7 – Classification des langages formels

état à un autre. Les langages basés sur les ensembles, comme Alloy, VDM et Z, permettent de décrire les états du système à travers des structures mathématiques comme les ensembles, les relations et les fonctions [Almeida et al. 2011]. Les transitions sont exprimées, dans ce cas, par des invariants ou des pré et post-conditions. Les préconditions capturent les conditions nécessaires sur les états pour que les opérations soient appliquées [Lamsweerde 2000]. À contrario, les post-conditions expriment les changements sur les états une fois les opérations appliquées [Lamsweerde 2000].

Langages basés sur les transitions. Ce type de langage se focalise sur les transitions entre les différents états. En effet, le système est décrit à travers un ensemble de fonctions de transition. Une fonction de transition donne pour chaque état de départ et un événement déclencheur, l'état de sortie correspondant [Lamsweerde 2000]. Ces langages peuvent être classés en langages basés sur les machines à états, comme les automates, StateCharts et PROMELA ; ou bien, les langages basés sur les graphes et les flux de données, où les données peuvent être représentées dans le modèle. Les langages tels que : Lustre, SDL [Rockstrom et Saracco 1982] et SCR [Alspaugh et al. 1992] s'appuient sur ce paradigme.

Langages stochastiques. Ces langages décrivent le système par un modèle probabiliste, où chaque transition est équipée d'une probabilité, i.e. une valeur réelle entre 0 et 1. Les langages comme les chaîne de Markov adhèrent à ce paradigme.

Langages pour les systèmes temps réel. Ces langages doivent être capables de spécifier des propriétés dites physiques telles que : le temps, la température, l'inclinaison, l'altitude, etc. [Almeida et al. 2011]. Parmi ces langages, on peut citer : les automates temporisés, utilisés souvent pour la modélisation des systèmes dont le comportement est capturé, en plus des états, par des horloges. Ces langages peuvent eux aussi être classés selon plusieurs critères [Wang

et al. 2004].

Langages basés sur les acteurs. Ces types de langages permettent de spécifier l'aspect structurel d'un système par les relations et les dépendances entre les différents acteurs du système. Un acteur est une entité active qui exécute des actions pour atteindre ses objectifs [Fuxman et al. 2004]. TROPOS [an Fuxman 2001] et le langage i^* [Yu 1997] figurent parmi ce type de langages.

Langages réactifs. Le système est modélisé par des entités qui s'exécutent parallèlement, dont le rôle est de répondre en permanence aux événements externes provenant de l'environnement [Harel et Pnueli 1985]. Parmi ces langages, on cite : REBECA [Sirjani et al. 2004] et ESTEREL [Berry et Gonthier 1992].

Langages opérationnels. Ces langages spécifient le système par un ensemble de processus pouvant s'exécuter en parallèle. Nous avons identifié deux catégories dans ces langages : les réseaux de Petri ordinaires et l'algèbre de processus comme CSP et CCS.

Langages basés sur les objets. Ces langages adoptent la notion d'objet et offrent des formalismes pour modéliser le système à travers ses objets. Object-Z [Smith 2012] est une extension du langage Z permettant la modélisation orientée objet. D'autres langages comme Disco [Jarvinen et Kurki-Suonio 1991] et TROL [Bucci et al. 1994] adoptent eux aussi la modélisation orientée objet.

2.6.2.2 Langages basés sur les propriétés

Les langages basés sur les propriétés décrivent le système à travers les données et les fonctions qui manipulent ces données. Nous classons ces langages comme suit :

Langages fonctionnels. Ces langages permettent de décrire le système sous la forme de collections mathématiques de fonctions [Lamsweerde 2000]. La notion de fonction est centrale dans ce type de langage. Ces langages sont de deux types : algébrique et axiomatique. Les fonctions, dans les langages algébriques, sont groupées selon les types d'objets de leur domaine. Les propriétés sont écrites par des équations. Les langages comme LOTOS [Bolognesi et Brinksma 1987], CASL [Astesiano et al. 2002] figurent dans ce paradigme. Les langages axiomatiques décrivent les fonctions du système par des logiques généralement d'ordre supérieur [Almeida et al. 2011]. En effet, les fonctions peuvent avoir des paramètres [Lamsweerde 2000]. Parmi ces langages, on peut citer : HOL, PVS, COQ.

Langages basés sur l'historique. Ce type de langage décrit les traces admissibles du comportement du système à travers le temps [Lamsweerde 2000]. Les propriétés sont, ainsi,

décrites par des assertions logiques portant sur les objets du système. Ces langages peuvent être : temporels ou stochastiques. Les logiques temporelles comme LTL et CTL permettent de décrire les traces temporelles du système. À contrario, les logiques stochastiques comme PCTL¹⁰ permettent de décrire la probabilité des traces temporelles.

2.7 Résultats obtenus

Dans cette section, nous présentons et discutons, pour chaque question de recherche, les différentes données récoltées. Un bilan, permettant de discuter les résultats statistiques, est dressé pour chaque question.

2.7.1 QR1 : Quels sont les langages de spécification utilisés pour la vérification formelle ?

Les langages de spécification sont utilisés dans deux contextes : soit pour la spécifier les facettes du système comme le comportement et la structure ; soit pour spécifier les propriétés que ce dernier doit satisfaire.

2.7.1.1 QR1.1 : Quels langages ont été utilisés pour la modélisation du système ?

Pour la spécification du système, les langages formels restent les plus utilisés avec 56% (48 articles sur les 81). Les langages semi-formels sont utilisés dans 41% (35 articles) des articles, contre seulement 2% (2 articles) qui adoptent les langages naturels ou textuels (figure 2.8). Nous tenons à préciser que certains travaux combinent les langages formels et les langages semi formels. Dans ce cas, l'article est compté deux fois (formel et semi-formel).

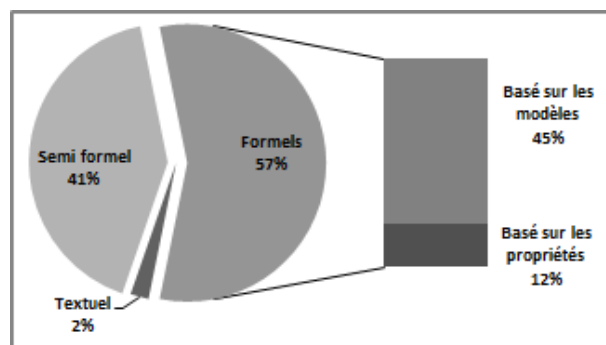


FIGURE 2.8 – Type des langage de spécification des systèmes

10. PCTL : Probabilistic Computation Tree Logic

2.7.1.1.1 Les langages formels

Pour les langages formels, les résultats montrent que les langages basés sur les modèles sont les plus utilisés pour la spécification des systèmes. En effet, les langages basés sur les modèles sont utilisés trois fois plus que les langages basés sur les propriétés (figure 2.8).

En outre, pour les langages basés sur les modèles, les résultats de la revue systématique montrent que, les machines à état restent les plus utilisées (figure 2.9), suivies par les langages pour les systèmes temps réels. Ces résultats confirment que les langages pouvant s'écrire sous forme graphique sont plus expressifs et de ce fait, sont les plus utilisés.

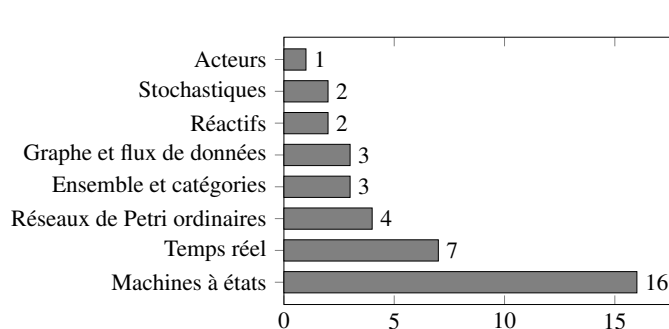


FIGURE 2.9 – Langages basés sur les modèles

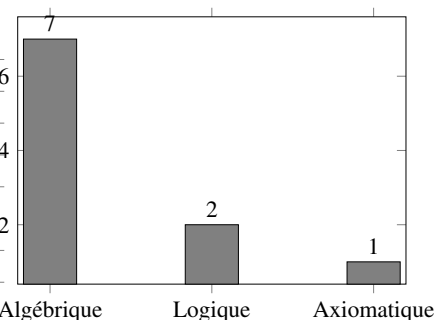


FIGURE 2.10 – Langages basés sur les propriétés

Pour les langages basés sur les propriétés (figure 2.10), les langages algébriques sont les plus utilisés dans la spécification du système, spécialement la méthode OTS/CafeObj [Ogata et Futatsugi 2003]. Cette dernière est utilisée dans 3 articles sur les 7 articles adoptant les langages algébriques. Cette méthode permet de spécifier le système par un formalisme proche des états transitions, écrit par des formules algébriques en CafeObj. Ce résultat rejoint celui concernant les langages formels. En effet, les langages supportant un formalisme proche des systèmes d'états et de transitions (des nœuds et des arcs) sont plus faciles à prendre en main [Seino et al. 2004].

2.7.1.1.2 Les langages semi-formels

UML reste le langage semi-formel le plus utilisé dans la spécification des systèmes (figure 2.11). En effet, sur les 35 articles utilisant les langages semi-formels, UML a été utilisé dans 31% des cas (11 articles). Les langages dédiés (DSL¹¹) sont de plus en plus utilisés pour la spécification des systèmes (20%). En effet, ces langages proposent des formalismes pour répondre aux contraintes d'un domaine spécifique. De ce fait, les syntaxes de ces langages sont réduites et faciles à prendre en main par les experts du domaine. Ce qui explique cette utilisation importante des DSL dans la spécification des systèmes.

11. DSL : Domain Specific Language

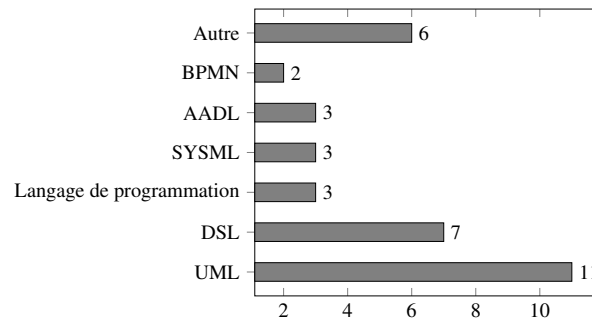


FIGURE 2.11 – Langages semi-formels

La transformation de ces langages semi-formels est nécessaire pour accomplir la vérification formelle. Par conséquent, ces langages sont souvent transformés en langages formels (figure 2.12, figure 2.13).

Les résultats montrent aussi, que même les langages formels, utilisés pour la spécification des systèmes, sont aussi transformés en d'autres langages formels. En effet, la transformation d'un langage vers un autre a été appliquée sur 64 des 81 articles. Ce qui représente un pourcentage de 79%. Sur la figure 2.13, on peut voir que les langages ont été transformés, le plus souvent (78% , 52 articles), en langages basés sur les modèles.

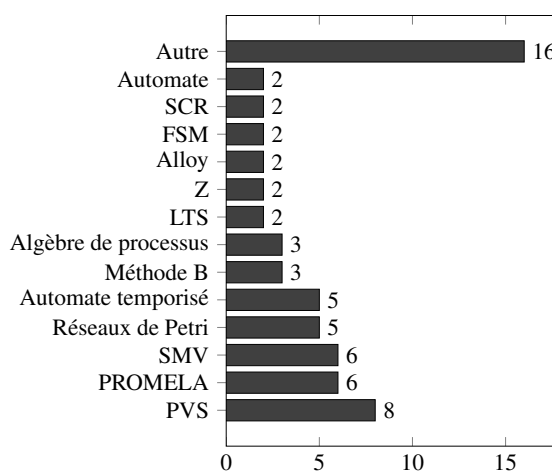


FIGURE 2.12 – Types des langages formels

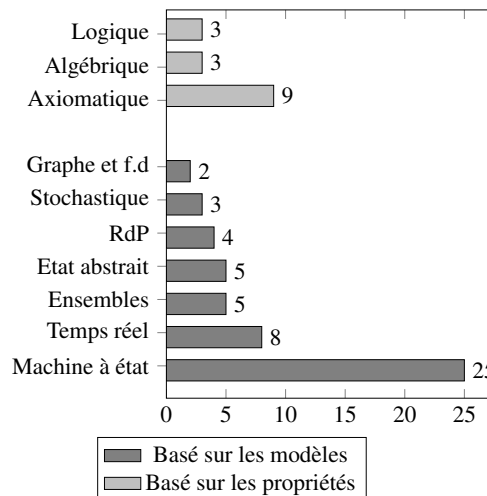


FIGURE 2.13 – Classes des langages formels

Les langages ont été transformés, dans la majorité des articles (45%) vers les langages des différents outils de vérification. Les langages les plus utilisés sont PVS, PROMELA, SMV et les automates temporisés (figure 2.12). Nous avons remarqué, aussi, que le langage PVS est utilisé dans 7 articles sur les 8 adoptant les langages axiomatiques. Le système PVS consiste en un langage de spécification, des vérificateurs de types, un assistant de preuve, ainsi que plusieurs bibliothèques et outils [Shankar et al. 2001]. PVS fournit un environnement intégré pour le dévelop-

pement et l'analyse de spécifications formelles permettant, donc, de supporter l'analyste dans ses activités de spécification, d'analyse, de modification et de la vérification formelle. La diversité des outils offerts par PVS et la haute expressivité de son langage de spécification explique sa forte utilisation.

2.7.1.2 Quels langages ont été utilisés pour la spécification des exigences ?

Pour la spécification des propriétés, les langages formels basés sur les propriétés sont utilisés dans 75% des articles (figure 2.14). Les logiques temporelles ont été utilisées dans la moitié des cas. En effet, les logiques LTL et CTL sont utilisées respectivement dans 23% et 25% des cas. Nous avons remarqué que les logiques stochastiques (5%) commencent à être utilisées pour la spécification des propriétés. Les langages basés sur les modèles sont utilisés dans 25% des articles (figure 2.14). Dans cet ensemble figurent les observateurs (6%) et les langages basés sur les ensembles (7%).

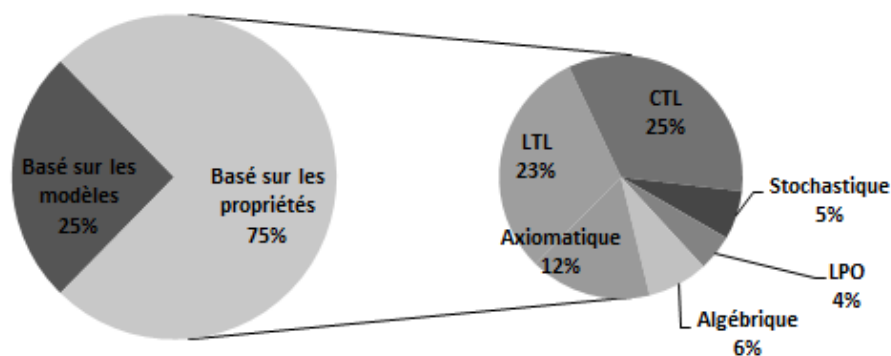


FIGURE 2.14 – Type des langages de spécification des propriétés

2.7.1.3 Observations

Ces résultats confirment l'utilisation importante des langages graphiques pour la modélisation des systèmes. En effet, les langages graphiques, qu'ils soient formels ou semi-formels sont les plus utilisés. Il est clair que l'information graphique est mieux perçue par l'être humain que l'information textuelle [Bajovs et al. 2013]. Nous avons aussi constaté que les auteurs combinent plusieurs langages pour modéliser plusieurs facettes du système, comme par exemple, la modélisation de la structure du système par des diagrammes statiques d'UML et le comportement par des réseaux de Petri. Nous avons aussi remarqué plusieurs transformations de modèles vers des modèles formels supportés par des outils (par exemple PVS, PROMELA,...). Ce constat rejoint ceux de Woodcock et al. [2009], Bjørner et Havelund [2014], Bennion et Habli [2014] qui insistent sur l'utilité des outils formels. Nous avons aussi constaté une émergence des DSL pour la spécification des systèmes.

Les logiques temporelles, surtout LTL et CTL, restent les langages les plus utilisés pour la spécification des propriétés. Dans 5 articles, les logiques LTL et CTL sont utilisées simul-

tanément. Nous avons aussi constaté de nouvelles tendances comme les observateurs et les logiques stochastiques. Une autre approche qui commence à être utilisée dans la spécification des propriétés est celle des pattern [Esteve et al. 2012, Ghazel et al. 2009, Dhaussy et al. 2009].

2.7.2 QR2 : Comment a été réalisée la vérification formelle ?

2.7.2.1 QR2.1 : Quelles sont les méthodes formelles utilisées ?

Le *Model-Checking* reste la méthode formelle la plus utilisée. En effet, sur les 81 articles retenus pour l'extraction des données, 75% (60 articles) utilisent le *Model-Checking*. 23% (19) seulement adoptent le *Theorem-Proving*. Les deux méthodes ont été utilisées conjointement dans seulement 2% (2 articles) articles.

2.7.2.2 QR2.2 : Ces techniques sont-elles supportées par un outil ?

Comme le *Model-Checking* est la méthode la plus utilisée pour la vérification formelle, les *Model-Checkers* sont plus présents que les *Theorem-Provers*. Les *Model-Checkers* les plus utilisés sont : SPIN, SMV, UPPAAL, FDR, NuSMV (figure 2.15). PVS est le *Theorem-Prover* le plus utilisé, suivi par CafeOBJ, B et Z (figure 2.16).

Les figures 2.17, 2.18 illustrent respectivement l'utilisation des *Model-Checkers* et *Theorem-Provers* à travers le temps. SMV, SPIN et UPPAAL sont les premiers *Model-Checkers* utilisés dans la vérification formelle et restent utilisés à ce jour (figure 2.17). Nous remarquons l'apparition de nouvelles techniques, à partir de l'année 2005, comme les réseaux de Petri, Alloy, SMT et les outils stochastiques.

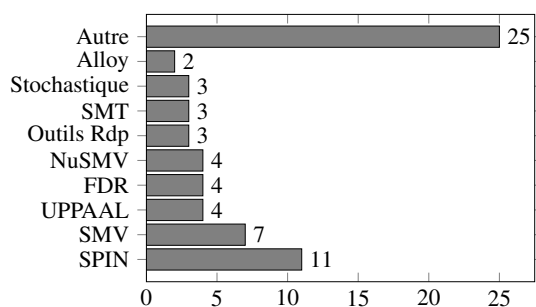


FIGURE 2.15 – Les *Model-Checkers*

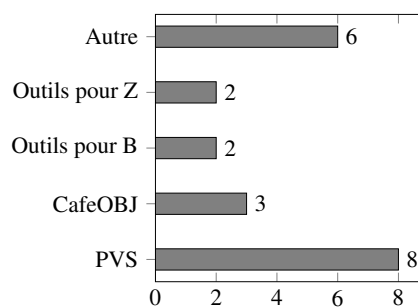


FIGURE 2.16 – Les *theorem provers*

Les premiers *Theorem-Provers* utilisés sont PVS et Z (figure 2.18). Cependant, PVS est moins utilisé ces cinq dernières années. Les outils supportant Z semblent devenir obsolètes, car ils ne sont plus utilisés depuis l'an 2000. Par contre, les outils pour la méthode B semblent être la nouvelle tendance dans le *Theorem-Proving*.

Dans les deux cas (*Model-Checker* et *Theorem-Provers*), on constate que de nouveaux outils formels sont développés et utilisés sur la période 1994-2014.

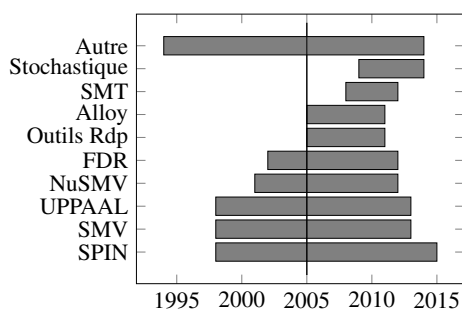


FIGURE 2.17 – Les *Model-Checkers* à travers le temps

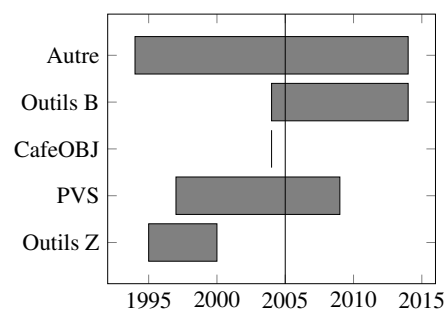


FIGURE 2.18 – Les *Theorem-Provers* à travers le temps

2.7.2.3 Observations

Clarke et Wing [1996] proposent, en 1996, de combiner le *Model-Checking* et le *Theorem-Proving* pour bénéficier des avantages de chaque méthode. Cette direction semblait très prometteuse. Mais, après 20 ans, les résultats de notre étude montrent que cette direction a été peu choisie. Au contraire, il ressort que la communauté scientifique penche sur l'amélioration des deux méthodes séparément.

Malgré le problème de l'explosion combinatoire, le *Model-Checking* reste la méthode la plus utilisée pour la vérification formelle. De nouvelles méthodes, telles que les SMT, Alloy et les méthodes stochastiques ont été développées afin de diminuer le problème de l'explosion combinatoire. Au niveau du *Theorem-Proving*, la méthode B semble être la plus utilisée ces dix dernières années. Nous déduisons aussi que les recherches de ces dix dernières années ont été plus axées sur le *Model-Checking* que le *Theorem-Proving*.

Bjørner et Havelund [2014] affirment dans leur article "*40 Years of Formal Methods*" que l'immaturation des outils formels est un vrai frein pour l'utilisation de la vérification formelle en industrie. Les résultats de notre étude confirment ce constat. En effet, nous avons constaté que les chercheurs modélisent leurs systèmes dans plusieurs langages de spécifications graphiques. Ils les transforment ensuite en langages formels supportés par des outils. L'utilisation de la vérification formelle dépend donc réellement de la capacité des outils formels. Nous pensons que, dans le futur, les outils formels devront proposer plus de fonctionnalités pour assister les concepteurs tout au long du cycle de développement. Ils doivent aussi proposer des langages graphiques faciles à interpréter et à utiliser. Les outils supportant la vérification et le raffinement (génération des spécifications plus détaillées, ou du code) comme la méthode B semblent être une perspective très intéressante pour l'intégration de la vérification formelle tout au long du cycle de développement.

2.7.3 QR3 : Quels sont les objectifs de la vérification formelle ?

Nous présentons dans les sections suivantes les différents objectifs de la vérification formelle, tels que le type des exigences vérifiées, le domaine d'application, la facette du système concernée par la vérification et le contexte de la vérification. Nous finissons par décrire le but général de la vérification formelle.

2.7.3.1 QR3.1 : Quelles sont les exigences vérifiées par ces méthodes ?

La figure 2.19 illustre les propriétés vérifiées par chaque méthode formelle. Nous avons recensé 157 propriétés vérifiées par des méthodes formelles. 82% (128) de ces propriétés ont été vérifiées par *Model-Checking* contre seulement 18% (29) vérifiées par *Theorem-Proving*. Concernant le type de propriétés vérifiées, les résultats (figure 2.19) montrent que plus de la moitié (89 propriétés, 57%) concernent la sûreté, la vivacité et l'absence de blocage. Les résultats montrent aussi que les propriétés telles que l'absence de blocage et l'atteignabilité sont vérifiées par *Model-Checking*. Les propriétés de cohérence et de correction sont vérifiées à égalité par *Model-Checking* et *Theorem-Proving*.

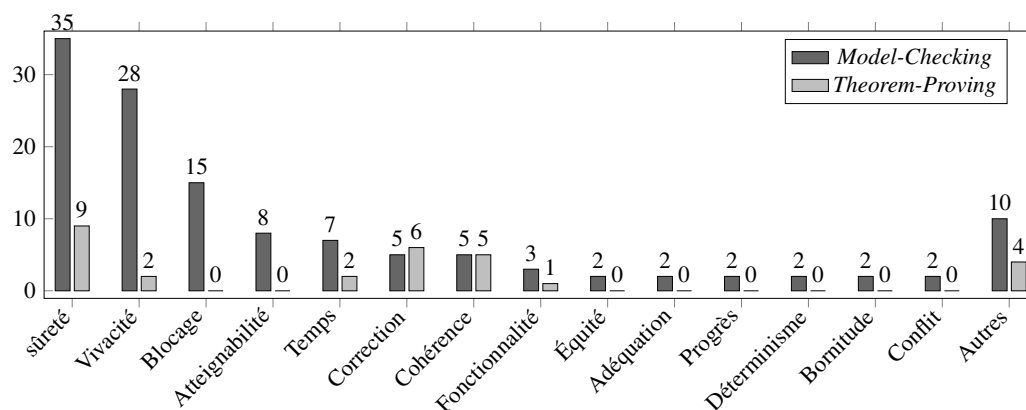


FIGURE 2.19 – Les propriétés vérifiées par chaque technique formelle

2.7.3.2 QR3.2 : Dans quel domaine la vérification formelle a été réalisée ?

Concernant le domaine de la vérification, cette dernière est plus utilisée dans le domaine du génie logiciels (17%, 14) et les systèmes temps réel (15%, 12) suivi des systèmes critiques (15%, 12), réactifs (14%, 11) et les systèmes de contrôle-commande (14%, 11), voir figure 2.20.

Le tableau 2.4 illustre la projection des propriétés vérifiées formellement par rapport à chaque domaine d'application. La propriété de sûreté est vérifiée à part égale dans le domaine du génie logiciel, des systèmes temps réel, des systèmes réactifs et du contrôle-commande. Néanmoins, la propriété de vivacité est souvent vérifiée dans les systèmes temps-réel. Sur les systèmes de contrôle-commande, les propriétés de sûreté, vivacité, l'absence de blocage et

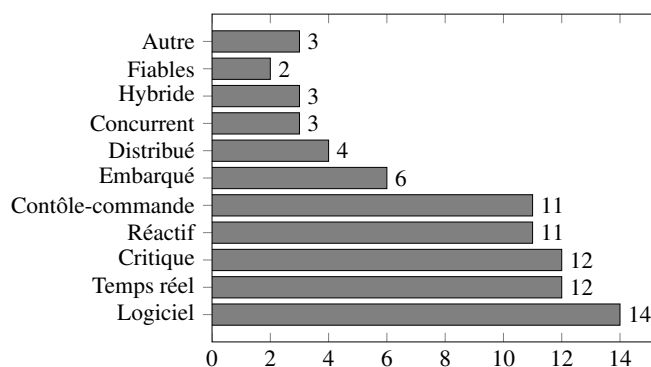


FIGURE 2.20 – Le domaine de la vérification

l'atteignabilité sont les plus vérifiées. Les propriétés de cohérence et d'adéquation sont le plus souvent vérifiées dans le domaine du génie logiciel. Dans les systèmes temps-réels, d'autres propriétés temporelles sont vérifiées en plus des propriétés de sûreté et de vivacité.

	Logiciel	T-Réel	Critique	Réactif	C-Cmd	Embarqué	Distribué	Concurrent	Hybride	Fiable
sûreté	6	6	7	7	6	2	2	2	2	2
Vivacité	2	7	2	4	3	4	2	3	0	0
Blocage	2	1	1	2	3	1	2	2	1	0
Correction	2	1	1	1	2	1	0	1	1	1
Cohérence	5	0	2	1	0	1	0	0	0	1
Temps	0	4	1	0	0	1	0	0	1	0
Atteignabilité	1	1	0	0	3	1	0	1	0	0
Fonctionnalité	1	0	0	0	2	0	0	0	0	0
Équité	0	1	1	0	0	0	0	0	0	0
Adéquation	2	0	0	0	0	0	0	0	0	0
Progrès	0	0	0	1	0	0	1	0	0	0
Déterminisme	1	0	0	0	0	0	0	1	0	0
Bornitude	0	1	0	0	0	1	0	0	0	0
Conflit	0	0	0	1	0	1	0	0	0	0

Tableau 2.4 – Les propriétés vérifiées par domaines d'application

2.7.3.3 QR3.3 : Quelles facettes du système ont été vérifiées ?

Dans 74% des articles (60), la vérification formelle porte sur la facette comportementale des systèmes. La structure a été vérifiée dans seulement 9% (7 articles). Cependant, la vérification conjointe du comportement et la structure a été pratiquée dans 15% des articles (12). Dans 2 articles, la facette du système vérifiée n'était pas reportée.

La figure 2.21 présente les différentes propriétés vérifiées au niveau du comportement et de la structure du système. Toutes les propriétés sont, le plus souvent, vérifiées au niveau comportemental qu'au niveau structurel, comme la sûreté et la vivacité. L'absence de blocage et l'atteignabilité ne sont vérifiées qu'au niveau comportemental. On déduit alors que ces propriétés sont des propriétés de comportement. La cohérence est vérifiée à part égale entre le comportement et la structure des systèmes.

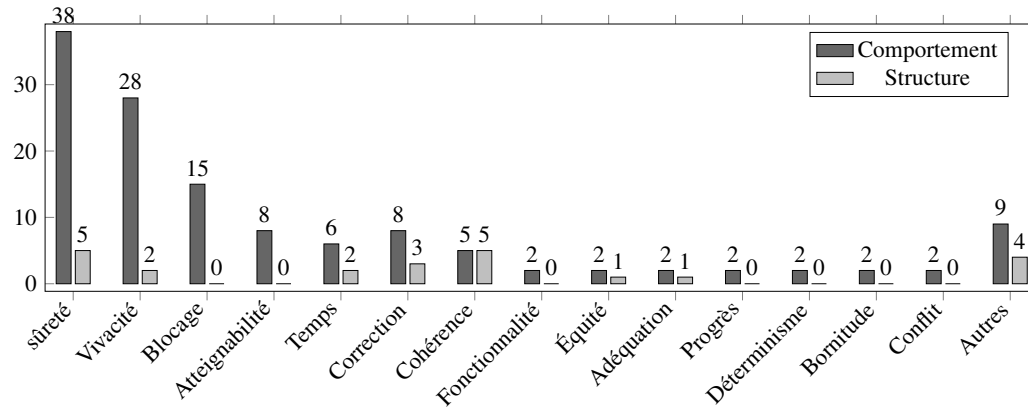


FIGURE 2.21 – Les propriétés vérifiées dans chaque facette du système

2.7.3.4 QR3.4 : Dans quels contextes (académique, industriel) la vérification a été réalisée ?

La vérification formelle a été plus pratiquée dans le domaine académique qu'en industrie. 73% (61) des articles proviennent des laboratoires de recherches académiques (figure 2.22). La vérification formelle en industrie est présente seulement dans 27% (22) des articles. Ce résultat confirme la faible utilisation des méthodes formelles dans le monde industriel. Dans ce dernier, le *Model-Checking* a été utilisé dans 82% (18) des cas contre seulement 18% (4) pour le *Theorem-Proving*. En académie, le *Model-Checking* est utilisé dans 72% (44) des cas contre 28% (17) pour le *Theorem-Proving*. De ce résultat, on déduit qu'en industrie le *Model-Checking* est la méthode formelle la plus pratiquée. Nous précisons que le *Model-Checking* et le *Theorem-Proving* ont été utilisés conjointement dans deux articles. Cela explique la somme $61+22=83$.

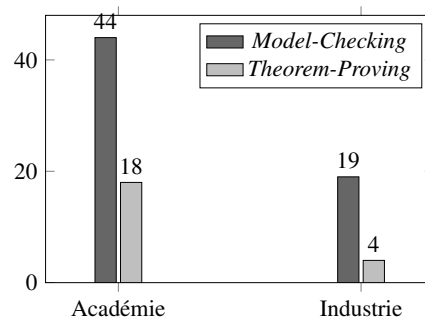


FIGURE 2.22 – Utilisation des méthodes formelles dans l'industrie et dans l'académie

2.7.3.5 QR3.5 : Quel est le but général de la vérification ?

La vérification est utilisée dans trois grandes approches : approche de génération du code, approche de vérification du code existant, et dans la modélisation. Pour la génération du code, le système est modélisé dans un langage formel, puis les propriétés de bon fonctionnement sont alors vérifiées sur ce modèle formel. À partir de ce dernier, le code est ensuite généré automatiquement. Dans l'approche de vérification du code existant, les modèles formels sont générés à partir du code afin de vérifier que ce dernier respecte certaines bonnes propriétés. La troisième approche utilise les méthodes formelles afin de s'assurer que les différentes modélisations du système, lors de la conception, sont correctes.

La figure 2.23 illustre l'objectif de l'utilisation des méthodes formelles. Dans plus de la moitié des articles (57%, 46 articles) les méthodes formelles ont été utilisées dans un objectif de modélisation. La génération du code et la vérification du code viennent en deuxième position avec respectivement 15% (12 articles) et 14% des articles (11 articles). L'objectif de la vérification n'est pas rapporté dans 15% des articles (12 articles).

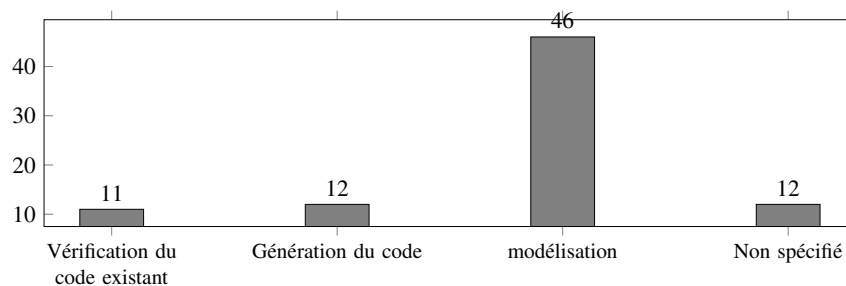


FIGURE 2.23 – L'objectif général de l'utilisation des méthodes formelles

2.7.3.6 Observations

Que ce soit par *Model-Checking* ou par *Theorem-Proving*, les propriétés de sûreté, de vivacité et de l'absence de blocage restent les propriétés les plus vérifiées au détriment des propriétés d'atteignabilité, de cohérence et de correction. Les propriétés telles que l'atteignabilité et l'absence de blocage sont vérifiées par *Model-Checking*. Certaines propriétés ne sont vérifiées qu'au niveau comportemental comme l'absence de blocage et l'atteignabilité. Les propriétés comme la sûreté et la vivacité sont vérifiées sur les deux facettes (comportement et structure) mais, elles sont vérifiées le plus souvent au niveau du comportement.

Le comportement reste la facette du système la plus vérifiée, comparée à la facette structurelle. Ce résultat est expliqué par la complexité et le dynamisme de cette facette qui rend la vérification manuelle de cette facette difficile voire même impossible. Néanmoins, la vérification conjointe des deux facettes semble être pratiquée par la communauté scientifique.

Notre étude fait également ressortir le fait que la vérification formelle porte majoritairement sur la seule modélisation du système au détriment de la vérification et de la génération du code

Plusieurs obstacles réduisent l'utilisation des méthodes formelles dans l'industrie. La complexité des méthodes formelles et l'absence ou l'immaturation des outils formels figurent parmi les premiers obstacles. Les résultats de cette revue systématique confortent cette analyse. En effet, les résultats montrent que l'application industrielle ne concerne qu'un tiers des travaux, les deux tiers restants ne concernant qu'une application purement académique. Il faut, toutefois, nuancer ce point car les industriels sont moins amenés à publier dans des revues scientifiques que les chercheurs du monde académique. Par ailleurs, les travaux concernant une application industrielle mettent cinq fois plus en œuvre le *Model-Checking* que le *Theorem-Proving*. Nous expliquons cela, par le niveau élevé d'automatisation dans le *Model-Checking*. Un autre obstacle qui ressort de cette étude est l'immaturation des outils formels. En effet, ces outils affichent des performances moyennes, voire insatisfaisantes, dans une vérification à grande échelle (industrielle).

2.8 Conclusion et Discussions

Dans ce chapitre, nous avons présenté une revue systématique de la littérature. Dans ce qui suit, nous discutons les résultats de cette étude à travers un bilan. Nous nous basons sur ces résultats pour définir des propositions qui permettent l'intégration de la vérification formelle dans un flot de conception automatisé.

2.8.1 Bilan

De cette étude, il ressort que les langages graphiques, qu'ils soient formels ou semi formels, restent les langages les plus utilisés pour la modélisation des systèmes. Une transformation de ces langages vers des langages formels supportés par des outils est la pratique usuelle pour conduire la vérification formelle. Il en ressort aussi que le *Model-Checking* reste la méthode formelle la plus utilisée pour vérifier le comportement et la structure d'un système. L'automatisation de cette méthode rend le processus de la vérification transparent à l'utilisateur et a permis au *Model-Checking* d'être adopté dans le monde académique et également industriel. Les recherches dans le domaine du *Model-Checking* ont permis de proposer de nouvelles solutions, telles que les solveurs SMT, pour limiter le problème de l'explosion du nombre des états dont souffre le *Model-Checking*.

2.8.2 Propositions

Nous rappelons que cette thèse CIFRE intervient dans un contexte industriel. Notre objectif général est d'intégrer la vérification formelle dans un contexte SCADA et de faciliter son utilisation par les concepteurs métiers. À la lumière de ces résultats, nous optons pour les solutions suivantes :

- L'utilisation des modèles métiers (DSL et langages de programmation) pour la spécification du système. Ces langages semi-formels sont maîtrisés par les concepteurs et faciles

- à utiliser par ces derniers.
- La transformation automatique des modèles métiers en modèles formels supportés par des outils formels inspirés de concepts de l'ingénierie dirigée par les modèles.
 - L'utilisation du *Model-Checking* pour la vérification. Le *Model-Checking* et l'obtention automatique des modèles formels permettront aux concepteurs métiers de procéder à des vérifications formelles des modèles métiers, sans être obligés de manipuler des formules mathématiques complexes. Nous souhaitons ainsi faciliter l'utilisation des méthodes formelles dans l'industrie.
 - La vérification des propriétés de sûreté, de vivacité et d'absence du blocage au niveau comportemental. Les propriétés de cohérence et de correction sont à vérifier au niveau structurel.
 - L'objectif général de la vérification formelle est de vérifier des codes et des modèles de conception existants.

Les parties suivantes présentent les contributions méthodologiques et techniques de cette thèse. Ces contributions sont illustrées et détaillées dans les chapitres 3, 4, 5 et 6. Les chapitres 3 et 4, de la deuxième partie, portent sur l'intégration de la vérification formelle dans un flot de conception automatisé.

Le chapitre 3 présente une approche pour la vérification formelle du comportement des composants de conceptions standardisés. En effet, le comportement de ces composants a été modélisé par des automates temporisés déduits à partir des codes existants. Des propriétés de sûreté, de vivacité et d'absence de blocage sont écrites en CTL et vérifiées par le *Model-Checker* UPPAAL.

Dans le chapitre 4, nous présentons une approche pour la vérification formelle de l'architecture structurelle d'un système sociotechnique. L'architecture en question a été modélisée par le langage Alloy et vérifiée automatiquement. Les modèles Alloy ont été déduits à partir d'un modèle métier normalisé, le P&ID.

Dans la troisième partie, qui comprend les chapitres 5 et 6, nous décrivons l'implémentation et la validation des deux approches. Dans le chapitre 5, les différentes transformations développées pour la génération automatique des modèles formels sont détaillées et expliquées. Nous reportons ensuite, dans le chapitre 6, les évaluations de l'application de nos deux approches pour la vérification d'un cas d'étude industriel.

Deuxième partie

Approches de vérification formelle : propositions méthodologiques

3

Vérification formelle d'une chaîne de contrôle-commande élémentaire

Résumé : Ce chapitre présente la mise en œuvre de vérifications formelles sur une bibliothèque d'éléments standards utilisés pour la conception des systèmes de contrôle-commande. La vérification de cette bibliothèque permettra de s'assurer que la conception des composants respecte les propriétés nécessaires. L'originalité des travaux porte sur la vérification conjointe de la chaîne de contrôle-commande complète des éléments standards. En utilisant l'IDM, le programme de commande et l'interface de supervision sont transformés automatiquement en automates temporisés. Le comportement de l'élément physique est modélisé manuellement par des automates temporisés. De même, le comportement de l'utilisateur face à l'interface de supervision, initialement capturé par des modèles de tâches, ensuite, est transformé automatiquement en automates temporisés. Les exigences que l'élément doit satisfaire sont écrites en CTL et vérifiées par Model-Checking (UPPAAL).

Sommaire

3.1	Introduction	62
3.2	Exemple illustratif : V2VM	62
3.2.1	Comportement de l'utilisateur	63
3.2.2	L'interface de supervision	63
3.2.3	Le programme de commande	64
3.2.4	La partie opérative	64
3.2.5	Les exigences de conception	65
3.3	Contexte et travaux connexes	65
3.3.1	Vérification formelle des programmes de commande	65
3.3.2	Vérification formelle des interfaces de supervision	66
3.3.3	Manques dans les travaux existants et problématique associée	67
3.4	Vérification formelle d'une chaîne de contrôle-commande élémentaire	67
3.4.1	Concepts utilisés	68
3.4.2	Approche générale	70
3.4.3	Formalisation d'une chaîne de contrôle-commande élémentaire	72
3.4.4	Vérification formelle d'une chaîne de contrôle-commande élémentaire	83
3.5	Conclusion	83

3.1 Introduction

Dans le cadre de la génération automatique des programmes de commande et des interfaces de supervision, plusieurs auteurs s'appuient sur une bibliothèque de composants de conception standardisés pour la spécification du système et la génération des applicatifs [Mouchard 2002, Lallican 2007, Bignon 2012, Bévan 2013]. Le système est conçu ainsi par agrégation de ces composants standardisés. Chaque composant se compose de vues représentant chacune une facette de la conception. Par exemple, la vue de supervision d'un composant correspond au composant logiciel permettant de le représenter sur l'IHM de supervision [Bignon 2012, Goubali et al. 2014]. Cependant, une réutilisation efficace n'a d'intérêt que si les composants standardisés ont les qualités requises.

L'objectif des travaux présentés ici est de proposer une démarche de vérification formelle des composants standardisés en intégrant leur caractère multi-vues. Nous espérons garantir ainsi la qualité de la chaîne de contrôle-commande complète de chaque composant. Nous portons une attention particulière à la modélisation formelle des différentes vues et à la génération automatique des modèles formels à partir de ces vues du composant.

Dans les sections suivantes, nous présentons un exemple illustratif d'un composant standard concret et constitué de plusieurs vues. Nous présentons ensuite, le contexte et les travaux connexes portants sur la vérification formelle des programmes de commande et des interfaces de supervision. Ensuite, nous présentons notre approche de vérification formelle de toute la chaîne de contrôle-commande. Nous terminons ce chapitre par une conclusion.

3.2 Exemple illustratif : V2VM

Nous appuyons nos travaux sur un composant standard des systèmes de gestion de fluide [Bignon 2012]. Il s'agit d'une vanne deux voies motorisée notée V2VM (figure 3.1) dans la suite du chapitre.

V2VM est un composant essentiel et répandu des procédés industriels. Une vanne joue le rôle de sectionnement dans un réseau de tuyauterie, comme l'interrupteur dans un réseau électrique. La vanne peut être actionnée à distance et dispose de détecteurs de position (ouvert ou fermé). La vue de supervision associée à la vanne permet de la commander et d'en afficher l'état. La vue de commande est positionnée entre la supervision et la partie opérative. Elle interprète les requêtes issues de la supervision en ordre de commande permettant de manœuvrer la vanne en tenant compte de son état remonté par ses détecteurs de position.

Pour assurer la communication entre les différentes vues de la vanne, les concepteurs métiers utilisent un ensemble de variables booléennes. Ces dernières, présentées dans le tableau 3.1, assurent l'échange d'informations entre l'utilisateur (opérateur), l'interface de supervision, le programme de commande et la partie opérative (système physique) de V2VM.

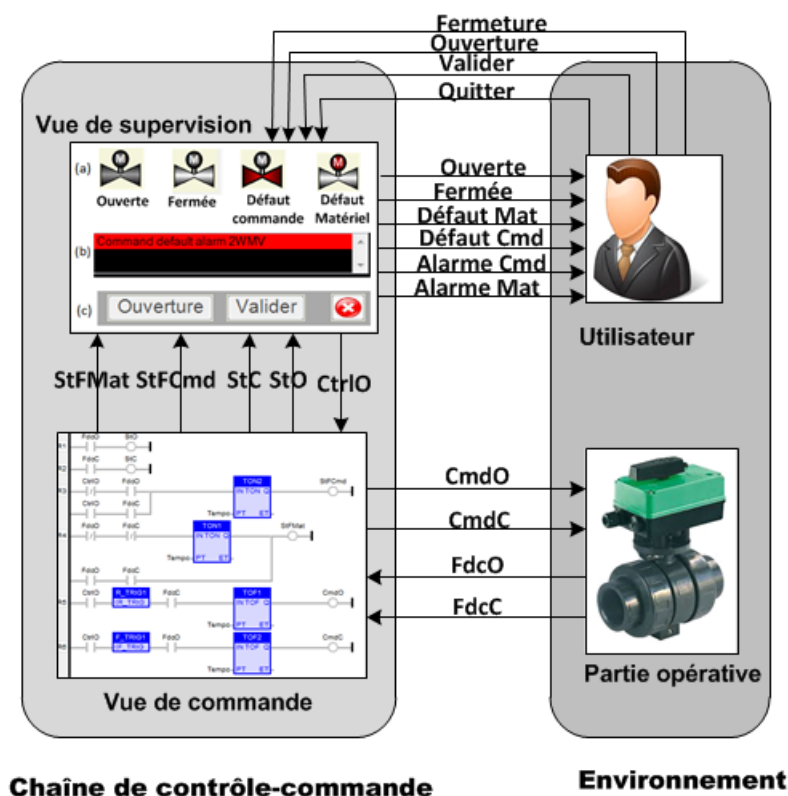


FIGURE 3.1 – Les différentes vues de la V2VM

3.2.1 Comportement de l'utilisateur

L'opérateur est celui qui pilote la vanne. Il a pour rôle de la commander à distance. Il interagit avec l'interface de supervision à travers des objets graphiques (figure 3.1). L'opérateur peut commander l'ouverture ou la fermeture de la vanne. Il peut aussi surveiller l'état de la vanne. La vanne peut être affichée ouverte, fermée, en défaut matériel ou en défaut de commande. Dans le cas de défaut, des alarmes de défaut de commande et de défaut de matériel sont aussi affichées à l'utilisateur.

3.2.2 L'interface de supervision

L'IHM de V2VM se compose de trois zones (figure 3.1) : a) Etat de la vanne, b) Zone d'alarmes, c) Bandeau de commande. Le widget de la vanne est présenté suivant plusieurs états (figure 3.1 a). La vanne est ainsi affichée (*Ouverte*, *Fermée*, *En défaut de commande*, *En défaut matériel*) si les variables booléennes *StO*, *StC*, *StFCmd*, *StFMat* sont respectivement à 1. La zone d'alarmes (figure 3.1 b) affiche en temps réel les alarmes de *Défaut de commande* ou de *Défaut de matériel* si les variables *StFCmd* et *StFMat* sont respectivement à 1. Si l'utilisateur clique sur le widget, le bandeau de commande, permettant à l'utilisateur de commander la

Variable	Description
CtrlO	= 1 Si l'opérateur demande l'ouverture de la vanne = 0 Si l'opérateur demande la fermeture de la vanne
StO	Vanne ouverte
StC	Vanne fermée
StFMat	Vanne en défaut matériel
StFCmd	Vanne en défaut de commande
CmdO	Ordre d'ouverture vers la vanne physique
CmdC	Ordre de fermeture vers la vanne physique
FdcO	Fin de course ouverture (la vanne est physiquement ouverte)
FdcC	Fin de course fermeture (la vanne est physiquement fermée)

Tableau 3.1 – Les différentes variables booléennes de la V2VM

vanne à distance, est affiché (figure 3.1 c). Il est composé de quatre boutons (*Ouverture*, *Fermeture*, *Valider*, *Quitter*). Les boutons *Ouverture* et *Fermeture* sont affichés de manière exclusive, l'un de l'autre.

3.2.3 Le programme de commande

Le programme de commande de V2VM est écrit en *LD* [IEC 2013] (figure 3.1) et s'exécute de manière cyclique. Il est constitué de plusieurs temporisateurs (2 TON et 2 TOF) et de deux blocs fonctionnels (RTRIG, FTRIG) [IEC 2013]. Lors de chaque cycle, il lit les entrées qui proviennent des capteurs de la vanne physique (FdcO, FdcC) et de la supervision (CtrlO). À partir de toutes ces informations, il calcule les sorties qui commandent l'ouverture (CmdO) ou la fermeture (CmdC) de la vanne. Il met également à jour les variables booléennes transmises à l'interface de supervision (StO, StC, StFCmd et StFMat), qui permettent l'animation du widget de la vanne dans l'interface de supervision.

3.2.4 La partie opérative

La partie opérative représente la vanne physique, ainsi que deux capteurs : le capteur de fin de course ouverture et le capteur de fin de course fermeture. Ces deux capteurs gèrent respectivement les variables booléennes FdcO et FdcC, pour indiquer respectivement l'état d'ouverture ou de fermeture de la vanne. Quand la vanne reçoit un ordre d'ouverture (CmdO=1) du programme de commande, elle commence à s'ouvrir. Quand elle atteint la position ouverte, le capteur fin de course ouverture met la variable FdcO à 1. Si elle reçoit un ordre de fermeture (CmdC=1), elle commence à se fermer et dès qu'elle atteint la position fermée, le capteur de fin de course fermeture met la variable FdcC à 1 et l'envoie au programme de commande pour indiquer la position fermée de la vanne.

3.2.5 Les exigences de conception

La vanne doit répondre à plusieurs exigences. Ces dernières peuvent être fonctionnelles, comme l'envoi d'un ordre d'ouverture (CmdO) à la vanne physique après une demande d'ouverture de l'opérateur ; ou bien non-fonctionnelles, comme le maintien de cet ordre d'ouverture (CmdO à 1) pendant 2 secondes.

3.3 Contexte et travaux connexes

La vérification formelle des programmes peut être utilisée dans deux contextes différents : soit dans une démarche de conception des systèmes sûrs, soit dans une démarche de vérification de programmes existants. Dans les deux cas, elle est basée sur une étape en amont permettant, d'une part, d'extraire du cahier des charges les propriétés à vérifier et d'autre part, de modéliser formellement le programme. La conception de systèmes sûrs permet d'avoir des programmes sûrs, générés ou raffinés à partir de modèles formels sur lesquels les propriétés ont déjà été vérifiées. À contrario, la vérification des programmes existants permet d'extraire un modèle formel à partir d'un programme déjà existant, puis, de vérifier la validité des propriétés sur le modèle formel. C'est dans ce cas que s'inscrivent les travaux présentés dans ce chapitre.

Nous avons évoqué le caractère multi-vues des composants standards. La chaîne de contrôle-commande d'un composant se compose d'un programme de commande au sens de la norme IEC 61131-3 [IEC 2013] et d'une IHM représentant le composant. Notre état de l'art reprend cette décomposition et s'intéresse d'abord à la vérification formelle des programmes de commande et ensuite à celle des IHM de supervision.

3.3.1 Vérification formelle des programmes de commande

Dans cette partie, nous nous intéressons aux programmes de commande implémentés dans des APIs. La spécificité des APIs réside dans le fait que le code est exécuté d'une manière cyclique. Ces programmes peuvent être vérifiés en ligne (programme en exécution) ou bien hors ligne, où le programme est vérifié avant son implémentation [Marangé 2008].

La modélisation et la vérification formelle des programmes de commande avant le développement de ces derniers n'est pas une pratique usuelle dans l'industrie. Les concepteurs des programmes de commande les développent à partir des exigences décrites dans le cahier des charges en se basant sur leurs expériences et leur savoir-faire. Afin d'établir la vérification formelle des programmes de commande produits par les concepteurs métiers, nous nous intéressons plus particulièrement aux travaux qui portent sur l'extraction de modèles formels à partir des programmes de commande déjà existants.

Ainsi, Gerber et al. [2010] ont réalisé une vérification formelle de la partie opérative et du programme de commande écrit initialement en IL. Afin d'en permettre la vérification, le programme est transformé en un modèle formel TNCES (une variété de Réseau de Petri). Dans [Sülflow et Drechsler 2008], les programmes de commande initialement écrits en IL

ont été transformés en SystemC. Puis, les exigences initiales ont été vérifiées en utilisant les solveurs SAT. Dans [Soliman et Frey 2011], les auteurs proposent une démarche qui permet de transformer des programmes de commande écrits en FBD en automates temporisés sous UPPAAL, afin d'en vérifier des propriétés de sûreté.

D'autres auteurs se sont intéressés à vérifier des programmes de commande écrits en LD. Roussel et Denis [2002] proposent d'exprimer les programmes LD en Algèbre *II* (une Algèbre de Boole adaptée aux signaux binaires) pour la vérification par *Theorem-Proving* des propriétés de sûreté. Da Silva Oliveira et al. [2011] proposent une méthode de transformation des programmes LD en réseau de Petri colorés (RdPC), afin de les vérifier par *Model-Checking*. Dans [Bender et al. 2008], les concepts de l'IDM ont été utilisés afin de transformer les programmes LD en réseau de Petri temporisés (RdP T-Temporisés). Les propriétés à vérifier sont écrites en LTL et sont vérifiées par le *Model-Checker* de TINA. Mokadem et al. [2010] proposent une méthode basée sur la réécriture des programmes LD en automates temporisés communicants sous UPPAAL, afin de vérifier des propriétés de sûreté et de vivacité.

3.3.2 Vérification formelle des interfaces de supervision

Le comportement de l'utilisateur face à une IHM peut conduire le système dans un état de défaut. Dans le domaine des systèmes interactifs, le comportement de l'utilisateur face à une IHM est capturé par des notations connues sous le nom de *modèles de tâches*. Le modèle de tâches est une représentation hiérarchique des tâches utilisateur. Chaque tâche décrit un objectif que l'utilisateur veut atteindre à partir d'un état donné du système. Parmi ces modèles de tâches, on peut citer : CTT [Paternò et al. 1997], KMAD [Kmad], HAMSTERS [Barboni et al. 2010], EOFM [Bolton et al. 2011], etc.

La vérification formelle des IHMs a suscité l'attention de plusieurs recherches. Paterno [1997] a utilisé le langage LOTOS pour la spécification des IHM et le *Model-Checking* comme méthode de vérification de propriétés d'utilisabilité (atteignabilité, visibilité, propriétés liées à la tâche) écrites en ACTL. Abowd et al. [1995] ont proposé une approche basée sur le *Model-Checking* pour la vérification formelle du dialogue. Ce dernier est spécifié par une table de représentation puis transformé en modèle d'entrée pour le *Model-Checker* SMV, afin de vérifier des propriétés d'atteignabilité, de fiabilité, et des propriétés liées à la tâche. Palanque et Bastide [1995] ont utilisé les ICO (une variété des réseaux de Petri) afin de modéliser le dialogue des IHM. Aït-Ameur et al. [1999] ont utilisé la méthode B pour spécifier et ensuite vérifier des propriétés d'atteignabilité, de visibilité et de fiabilité des IHM. Bolton et al. [2011] proposent un modèle formel de tâche nommé EOFM. Les modèles EOFM sont transformés en code sous SAL (*Model-Checker* et vérifiés par *Model-Checking* avec les différents modèles du système, sur lesquels des propriétés de sûreté écrites en LTL ont été vérifiées.

3.3.3 Manques dans les travaux existants et problématique associée

À notre connaissance, il n'existe pas à ce jour de travaux proposant une vérification conjointe des programmes de commande et des IHM de supervision. La vérification unitaire de l'une ou l'autre des parties ne garantit pas le bon fonctionnement de l'ensemble à l'intégration. En plus, le comportement de l'utilisateur face à l'IHM de supervision peut conduire le système à un état de défaut. Dans ce sens, nous souhaitons vérifier l'ensemble de la chaîne de contrôle-commande composée de l'IHM de supervision et le programme de commande. Notre modélisation doit permettre de vérifier un composant de contrôle-commande en tenant compte de ses échanges avec son environnement composé de l'utilisateur et du composant physique.

Pour la vérification conjointe de toute la chaîne de contrôle-commande, trois contraintes sont à prendre en compte dans le processus de modélisation. La première concerne le langage formel à utiliser. Les comportements de l'utilisateur, de l'IHM de supervision, du programme de commande et du composant physique doivent être modélisés dans le même langage formel. Cependant, ces différentes parties n'ont pas la même nature de comportement. Le comportement du programme de commande, par exemple, est temporel, alors que celui de l'IHM ne l'est pas. Le défi consiste, ici, à contourner les manques du langage formel choisi, pour l'adapter aux caractéristiques des différentes parties. Le deuxième défi est la génération automatique des modèles formels à partir des vues du composant pour faciliter l'intégration de vérification formelle dans un contexte industriel. La modélisation doit être modulaire pour faciliter cette génération automatique. La troisième contrainte est l'explosion combinatoire du nombre des états. En effet, comme expliqué dans la section 2.8 du chapitre 2, nous avons choisi d'utiliser le *Model-Checking* comme méthode de vérification pour faciliter l'intégration des méthodes formelles dans un contexte industriel. En revanche, cette méthode souffre du problème de l'explosion combinatoire, surtout dans le cas d'un système industriel complexe. Des mesures sont alors à prendre en compte afin d'éviter cette explosion combinatoire du nombre des états.

3.4 Vérification formelle d'une chaîne de contrôle-commande élémentaire

Nous avons choisi le *Model-Checking* comme technique de vérification formelle. Ce choix est guidé par la facilité d'intégration du *Model-Checking* dans une méthode industrielle car il ne nécessite pas l'intervention d'un expert pendant le processus de vérification.

Afin de gérer le comportement temporel de la commande, nous avons choisi les **automates temporisés** comme langage formel commun pour la modélisation. Nous justifions ce choix par la disponibilité de nombreux *Model-Checkers* basés sur ce formalisme. Nous proposons des solutions pour contourner le manque de ce langage dans la modélisation du parallélisme dans les interfaces de supervision et pour la modélisation du comportement de l'utilisateur.

Pour faciliter l'obtention des modèles formels, nous proposons l'utilisation des techniques de l'IDM afin de générer les modèles formels à partir des modèles métiers maîtrisés par les

concepteurs.

Pour résoudre les problèmes d'explosion combinatoire déjà pénalisant à l'échelle d'un composant et pour réduire l'indéterminisme, le comportement du composant physique est également modélisé sous la forme d'automates temporisés. Cette modélisation a pour objectif de supprimer les combinaisons des entrées non significatives qui sont physiquement impossibles.

3.4.1 Concepts utilisés

Dans cette section, nous présentons le langage formel utilisé pour la spécification du comportement du composant standard avec ses vues (les automates temporisés), les exigences à vérifier (CTL) sur cet composant et le *Model-Checker* UPPAAL.

3.4.1.1 Automate temporisés

Les *Automate Temporisés* (AT) sont des machines à états qui gèrent le temps physique à travers un ensemble fini d'horloges. Ces horloges sont associées aux transitions et évoluent de manière synchrone [Alur et Dill 1990].

Formellement, un AT est un 6-uplet $A = \langle L, L_0, \Sigma, X, I, E \rangle$, où : L est un ensemble fini de localités, $L_0 \subseteq L$ est l'ensemble des localités initiales, Σ est un ensemble fini d'actions, X est un ensemble fini d'horloges, $I(l)$ dans $\phi(X)$ est un invariant sur les horloges, associé aux localités. Il peut être écrit dans l'une des deux formes : $x \leq c$ ou bien $x < c$, avec x une horloge et c une constante. $E \subseteq L \times \Sigma \times 2^X \times \phi(X) \times L$ est un ensemble de transitions. Une transition représente un arc orienté reliant la localité l à une autre localité $l' \in L$. Généralement écrite $l \xrightarrow{a, \varphi, \lambda} l'$, avec a une action dans Σ et φ une contrainte (guard) d'horloge sur X qui détermine l'activation de la transition, l'ensemble $\lambda \subseteq X$ englobe les horloges à réinitialiser lors de la transition [Alur et Dill 1990].

La sémantique de l'automate temporisé A est définie par un système de transitions, où les états sont définis par une paire (l, v) avec l une localité et v la valeur des horloges de X telle que v satisfait l'invariant $I(l)$ de la localité l . (l_0, v_0) est un état initial si $l_0 \in L_0$ et $v_0(x) = 0$ pour toute horloge x . Le système de transition évolue de deux manières :

- Ecoulement du temps : pour un état (l, v) et un délai $\delta \geq 0$, $(l, v) \xrightarrow{\delta} (l, v + \delta)$ si $v + \delta$ satisfait l'invariant $I(l)$.
- Actions : pour un état (l, v) et une transition $l \xrightarrow{a, \varphi, \lambda} l'$ tel que v satisfait φ , $(l, v) \xrightarrow{a} (l', v[\lambda := 0])$.

3.4.1.2 UPPAAL

L'outil UPPAAL [Larsen et al. 1997] est un *Model-Checker* pour les systèmes temps réel modélisés sous forme d'un réseau d'automates temporisés communicants. Les différents automates communiquent par l'envoi ($m!$) et la réception ($m?$) de messages à travers des canaux de communication (m). L'émetteur et le récepteur du message évoluent simultanément si leur

condition ϕ est vraie. Dans ce chapitre, on considère deux propriétés spécifiques à UPPAAL : la localité instantanée (commit) et les messages en diffusion (broadcasts). La localité instantanée est une localité où l'écoulement du temps est égal à 0 (elle est représentée par une localité avec la lettre «C»). Les messages broadcasts sont utilisés pour faire communiquer simultanément avec d'autres automates. Dans ce cas, il y a un automate émetteur et plusieurs automates récepteurs.

La vérification de modèles s'effectue à l'aide de réseaux d'automates temporisés. Un exemple d'automate est donné en figure 3.2.

Les nœuds représentent les états que le système peut prendre. Les nœuds peuvent avoir un nom (ex : "Running" figure 3.2). Ils peuvent également avoir des invariants sur les horloges (ex : " $c1 < PT + 1$ ") qui rendent invalide l'état du nœud si la contrainte (l'invariant) est violée.

Les arcs représentent une transition d'états. Chaque arc est susceptible de contenir plusieurs informations : des gardes (vert clair) qui conditionnent l'activation de l'arc ; des assignations (bleu foncé) qui mettent à jour les variables ; des synchronisations de processus (bleu ciel).

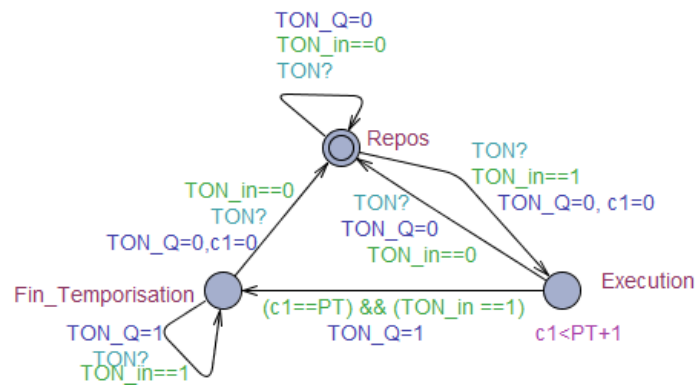


FIGURE 3.2 – Exemple d'automate temporisé dans UPPAAL

3.4.1.3 CTL

Cette logique arborescente et temporelle a été initialement introduite par Clarke et Emerson [1981]. Elle porte sur les états d'un système et les traces qui partent de ces états.

La syntaxe de CTL est définie comme suit :

$$\phi, \psi ::= \mathbf{EX}\phi \mid \mathbf{AX}\phi \mid \mathbf{E}\phi\mathbf{U}\psi \mid \mathbf{A}\phi\mathbf{U}\psi \mid \neg\phi \mid \phi \vee \psi \mid p \mid q \mid \dots$$

ϕ et ψ dénotent des formules, p et q dénotent des propositions atomiques. Les quantificateurs, existentiel (**E**) et universel (**A**) ont les significations respectives suivantes : "il existe un chemin dans lequel", et "tout chemin vérifie". La formule $\mathbf{EX}\phi$ signifie qu'il existe un chemin dans lequel un successeur immédiat de l'état courant satisfait la formule ϕ . $\mathbf{A}(\phi\mathbf{U}\psi)$ signifie que pour tout chemin partant de l'état courant, ψ est vrai dans un état futur et que tous les états

entre l'état courant et cet état futur satisferont la formule φ . À partir de ces modalités, on peut définir d'autres modalités comme :

$$\begin{aligned} \mathbf{EF}\varphi &\equiv \mathbf{E}(\mathbf{True} \mathbf{U} \varphi) & \mathbf{AF}\varphi &\equiv \mathbf{A}(\mathbf{True} \mathbf{U} \varphi) \\ \mathbf{EG}\varphi &\equiv \neg \mathbf{AF} \neg \varphi & \mathbf{AG}\varphi &\equiv \neg \mathbf{EF} \neg \varphi \end{aligned}$$

$\mathbf{EF}\varphi$ signifie que pour quelques chemins, il existe un état dans lequel φ est vraie. Cette modalité porte sur *la possibilité*, i.e. il est possible d'atteindre un état qui vérifie la formule φ . $\mathbf{AF}\varphi$ signifie que pour tous les chemins, il existe un état qui satisfait la formule φ . Cette modalité signifie que φ est *inévitabile* [Markey 2003]. $\mathbf{EG}\varphi$, quant à lui, signifie qu'il existe un chemin, et tous les états de ce chemin satisfont la formule φ . $\mathbf{AG}\varphi$ décrit que pour tous les chemins et tous les états, la formule φ est vraie. La formule $\mathbf{AG}(\varphi \rightarrow \mathbf{AG}\varphi)$, par exemple, exprime la stabilité, i.e. dans tous les chemins, et dans tous les états, si φ devient vraie alors, elle continuera à rester vraie.

La sémantique de CTL est définie par un système $M = (S, A, \dots, A_k, L)$, où : S un ensemble d'états, $A_i \subset S \times S$ une relation sur S représentant les transitions possibles à partir de chaque état i (chemin) et L une évaluation des propositions atomiques vraies dans chaque état. Avec s_0 un état dans S , φ et $\psi \in CTL$ et p une variable propositionnelle atomique. La sémantique de CTL est définie comme suit [Clarke et Emerson 1981] :

$$\begin{array}{lll} s_0 \models p & \text{si, et seulement si} & p \in L(s_0) \\ s_0 \models \neg \varphi & \text{si, et seulement si} & s_0 \not\models \varphi \\ s_0 \models \varphi \wedge \psi & \text{si, et seulement si} & s_0 \models \varphi \text{ et } s_0 \models \psi \\ s_0 \models \varphi \vee \psi & \text{si, et seulement si} & s_0 \models \varphi \text{ ou } s_0 \models \psi \\ s_0 \models \mathbf{EX}\varphi & \text{si, et seulement si} & \text{pour un état } t \text{ tel-que } (s_0, t) \in A_i, t \models \varphi \\ s_0 \models \mathbf{A}[\varphi \mathbf{U} \psi] & \text{si, et seulement si} & \text{pour tout chemin } (s_0, s_1, \dots), \exists j \geq 0 : s_j \models \psi \\ & & \wedge \forall i \geq 0 \wedge i < j : s_i \models \varphi \\ s_0 \models \mathbf{E}[\varphi \mathbf{U} \psi] & \text{si, et seulement si} & \text{pour quelque chemin } (s_0, s_1, \dots), \exists j \geq 0 : s_j \models \psi \\ & & \wedge \forall i \geq 0 \wedge i < j : s_i \models \varphi \end{array}$$

Nous tenons à préciser que l'outil UPPAAL implémente juste un fragment de la logique CTL [Behrmann et al. 2004]. En effet, les modalités (**G**, **F**) de CTL sont représentées par des symboles différents dans l'outil UPPAAL (tableau 3.2), cependant les modalités (**X**, **U**) n'existent pas dans cet outil.

CTL	A	E	F	G	X	U	$\mathbf{AG}(p \rightarrow \mathbf{AF}q)$	\wedge	\vee	\neg	\Rightarrow
UPPAAL	A	E	<>	[]	-	-	$p \rightarrow q$	&& (and)	(or)	! (not)	imply

Tableau 3.2 – Correspondances des opérateurs entre CTL et UPPAAL

3.4.2 Approche générale

Notre approche de vérification formelle de la bibliothèque de composants standards est présentée sur la figure 3.3. Elle repose sur deux étapes : l'étape de formalisation et l'étape de

vérification.

L'étape de formalisation (Formalisation sur la figure 3.3) consiste en la modélisation de toute la chaîne de contrôle-commande en automates temporisés. Au niveau de l'IHM de supervision, nous proposons des modèles génériques pour modéliser le comportement des différents objets graphiques constituant une IHM. Ces modèles sont ensuite complétés et paramétrés à partir des données embarquées dans l'IHM de supervision de chaque composant. Notre approche de vérification permet aussi la génération automatique des automates temporisés à partir de programmes de commande écrits en LD. Pour cela, un ensemble de règles de transformations a été défini. Le comportement de l'opérateur face à l'interface de supervision est initialement capturé par un modèle de tâches, maîtrisé par les experts informaticiens. Ensuite, à partir de ce modèle de tâches et d'une collection de règles de transformations, que nous avons définis, nous générons automatiquement des automates temporisés. Nous modélisons, manuellement, en automates temporisés le comportement du composant physique. Les exigences à vérifier, issues du cahier des charges, sont écrites en logique temporelle (CTL). Seules les propriétés CTL et l'automate temporisé modélisant la partie opérative sont à concevoir manuellement. Les propriétés CTL et les automates temporisés sont ensuite utilisés, lors de la deuxième étape, comme données d'entrée pour le *Model-Checker* UPPAAL, afin de vérifier que le composant standard satisfait ses exigences. Ces étapes sont détaillées dans les sections suivantes.

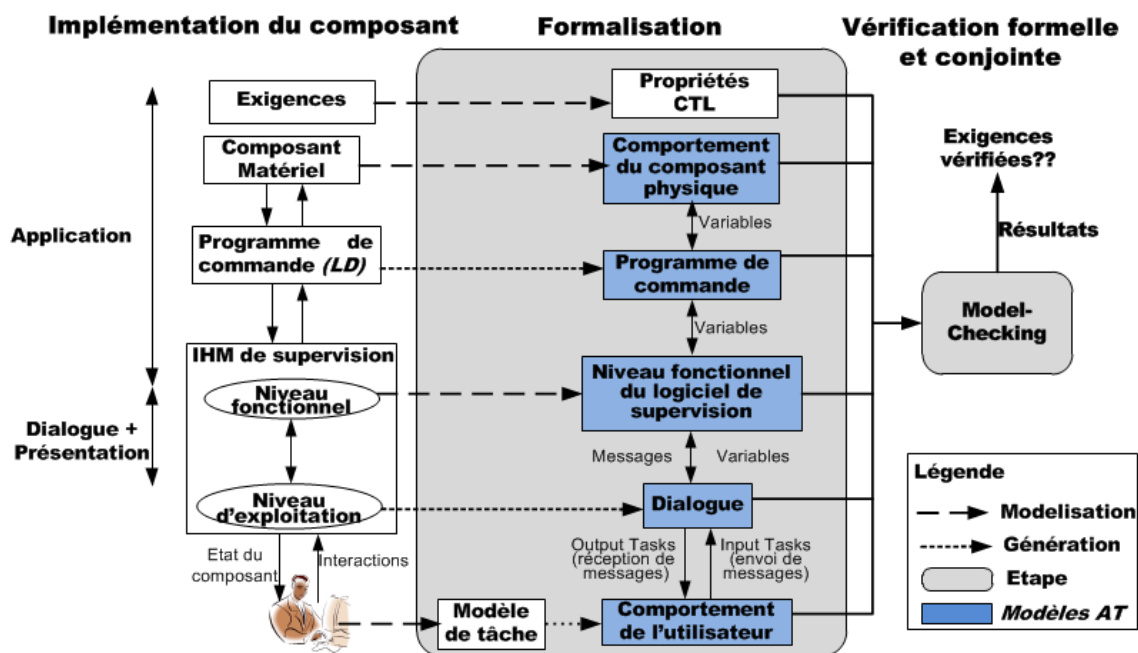


FIGURE 3.3 – Approche de vérification formelle des composants standards

3.4.3 Formalisation d'une chaîne de contrôle-commande élémentaire

Nous présentons la modélisation de toute la chaîne de contrôle-commande, représentant le comportement de l'interface, du programme de commande et de l'utilisateur, en automates temporisés qui sont déduits respectivement à partir du code de l'interface de supervision d'un composant standard, du code LD et du modèle de tâches. Le comportement du composant physique est aussi modélisé en automates temporisés. Les exigences que le composant doit satisfaire sont écrites en logique temporelle CTL.

3.4.3.1 Modélisation du comportement de l'utilisateur

3.4.3.1.1 Utilisation des Modèles de tâches

Le comportement de l'utilisateur face à l'IHM du composant standard est modélisé par un modèle de tâches. Ce dernier a la structure d'une arborescence. Le sommet de cette arborescence représente l'objectif général de la tâche (exemple : *Ouvrir une vanne*). Les branches décrivent les sous-objectifs qui peuvent eux aussi se décomposer en sous objectifs et ainsi de suite jusqu'à ce que les objectifs ne puissent plus être décomposés. Ce qui signifie que le niveau des tâches atomiques est atteint [Aït-Ameur et al. 2005]. Ce niveau correspond aux feuilles de l'arborescence. Les nœuds de l'arborescence correspondent aux opérateurs qui permettent de décrire l'enchaînement (séquence, parallélisme ...) des tâches.

La figure 3.4 présente un extrait du modèle de tâche qui décrit la tâche *Ouvrir la vanne*. Nous avons utilisé différents types de tâches élémentaires : *tâches utilisateurs* (actions de l'utilisateur sur l'IHM), des *tâches systèmes* (de l'IHM vers l'utilisateur) et des *tâches abstraites* (tâches composées d'autres sous-tâches de différents types).

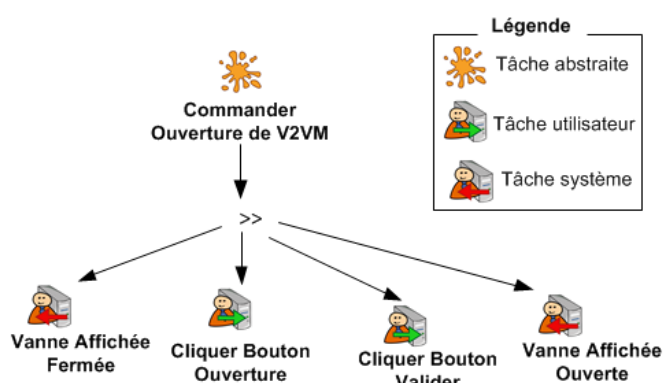


FIGURE 3.4 – Extrait de la tâche *Ouvrir la vanne* dans la notation Hamsters

Ces tâches sont reliées par différents opérateurs [Aït-Ameur et al. 2005]. L'opérateur $>>$ (appelé *Activation*) induit la séquence chronologique des sous-tâches de gauche à droite sur le modèle. L'opérateur \square signifie que seule l'une des sous-tâches est réalisable ; c'est un opérateur

équivalent du OU logique. L'opérateur $|||$, dit de *Concurrence*, induit le parallélisme des sous-tâches concernées ; c'est l'équivalent du ET logique. L'opérateur $[>$, appelé *Désactivation*, induit la notion de désactivation de la sous-tâche de gauche si la sous-tâche de droite dans le modèle est réalisée (i.e. la sous-tâche de droite désactive la sous-tâche de gauche).

3.4.3.1.2 Génération des automates temporisés à partir du modèle de tâche

Le modèle de tâche est composé essentiellement de tâches et d'opérateurs. La génération des automates temporisés à partir des modèles de tâches consiste alors à transformer les tâches et les opérateurs en états et transitions d'un automate temporisé.

Nous proposons de transformer les tâches systèmes et utilisateurs en canaux de communication sous UPPAAL (tableau 3.3). Ainsi, les tâches de type système (transmission de l'information du système vers l'utilisateur) sont modélisées par une réception de messages (exemple : *Tâche ?*) i.e. l'utilisateur reçoit l'information à partir du système. Les tâches utilisateurs (interactions de l'utilisateur) sont modélisées par des envois de messages (*Tâche !*) en UPPAAL, ce qui signifie que l'utilisateur envoie une information vers le système.







Tâches	Description	Automate
 Tâche	Tâche système	
 Tâche	Tâche utilisateur	
 Tâche	Abstraite (racine)	

Tableau 3.3 – Transformation des tâches en signaux

Nous nous sommes inspirés des travaux d'[Aït-Ameur et Baron \[2006\]](#) pour décrire les opérateurs en automates temporisés. Dans le tableau 3.4, les opérateurs basiques (Activation, Choix, Concurrence, Tâche itérative) d'un modèle de tâches sont décrits sous forme d'automates sous UPPAAL. Les autres opérateurs (Ordre indépendant, Tâche optionnelle, Désactivation) peuvent se déduire des opérateurs basiques [[Aït-Ameur et Baron 2006](#)].

L'opérateur d'activation est transformé en une succession d'états normaux reliés par des transitions. Notons que l'opérateur d'activation peut porter sur plusieurs tâches, dans ce cas, nous générons autant de localités qu'il y a de tâches.

L'opérateur du choix est transformé en un état qui est l'état de départ de plusieurs transitions représentant chacune une tâche concernée par cet opérateur.

Comme le parallélisme n'est pas géré par les automates temporisés, la modélisation de l'opérateur de concurrence pose problème. Nous utilisons les caractéristiques des états instantanés UPPAAL pour contourner ce problème. Nous modélisons l'opérateur de concurrence par un état instantané ayant autant de transitions, qui bouclent sur cet état, qu'il y a de tâches.

Opérateur	Description	Automate
$T\grave{a}che1 \gg T\grave{a}che2$	Activation	
$T\grave{a}che1 \square\square T\grave{a}che2$	Choix	
$T\grave{a}che1 T\grave{a}che2$	Concurrence	
$T\grave{a}che^N$	Itérative	

Tableau 3.4 – Transformation des opérateurs de tâches en automates temporisés

Nous utilisons des variables booléennes ($T1$ et $T2$ sur le tableau 3.4, ligne 3) pour s'assurer de l'exécution de toutes les tâches ($T1=1 \ \&\& \ T2=1$).

Les tâches itératives avec N itérations sont modélisées par un état normal ayant une seule transition qui boucle sur cet état et une variable entière correspondant au nombre d'itérations (N). Tant que N est supérieur à 0 alors une itération de la tâche est exécutée et le nombre N est décrémenté. Une fois toutes les itérations réalisées ($N==0$), l'automate quitte l'état de la tâche itérative.

La figure 3.5 illustre l'automate temporisé représentant la tâche *Ouvrir la vanne*, déduit à partir du modèle de tâches de la figure 3.4.

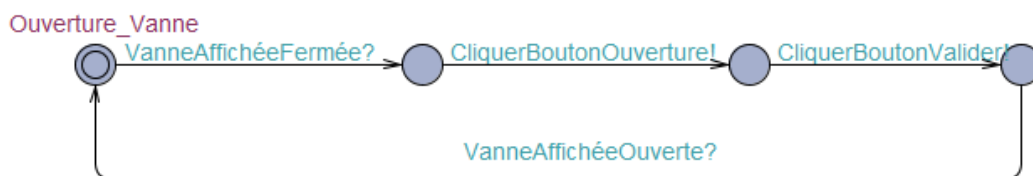


FIGURE 3.5 – Automate temporisé correspondant au modèle de tâches de la figure 3.4

3.4.3.2 Modélisation de l'IHM de supervision

Les interfaces de supervision développées avec les logiciels SCADA disposent d'une architecture logicielle à deux niveaux (figure 3.3). Le niveau d'exploitation gère l'interaction avec l'utilisateur, aussi bien concernant l'aspect graphique de l'application que concernant les dispositifs de commande. Le niveau fonctionnel constitue quant à lui le cœur de l'application. Il

comporte l'ensemble des traitements de supervision et gère également les échanges de données avec le programme de commande.

Si nous nous replaçons dans le contexte plus large des systèmes interactifs (section 1.3.3.1 du chapitre 1), les modules de présentation et dialogue correspondent au niveau d'exploitation des logiciels SCADA ; le module d'application correspond, quant à lui, au niveau fonctionnel des logiciels SCADA, au programme de commande et au comportement du composant matériel (figure 3.3).

Nous présentons ci-après, notre modélisation du module de dialogue et du module d'application. Le module de présentation n'est pas modélisé dans ce chapitre.

3.4.3.2.1 Modélisation du niveau d'exploitation (Module de dialogue)

Le comportement d'une IHM résulte du comportement de ses objets graphiques. Ces derniers sont classés par [Moussa et al. \[2002\]](#) en objet de commande et objet informationnel. L'objet de commande est un objet interactif qui permet à l'utilisateur de lancer des commandes, comme par exemple un bouton. Par contre, l'objet informationnel ne sert qu'à afficher des informations à l'utilisateur, comme par exemple un texte, une animation, etc.

Inspirés par les travaux de [Moussa et al. \[2002\]](#) et par analogie avec leur classification des objets interactifs en objet de commande et en objet informationnel, nous avons construit des automates temporisés génériques afin de modéliser le module de dialogue. La communication, entre ces objets et l'utilisateur, est modélisée par l'envoi et la réception de messages.

L'objet de commande. Par exemple un bouton. Cet objet est constitué de trois états : *Masqué*, *Affiché Activé*, *Affiché Désactivé*, voir figure 3.6. À l'état initial, l'objet de commande est dans l'état *Masqué*, qui modélise le fait que l'objet est masqué pour l'utilisateur, soit parce que la vue qui le contient (*VueMèreMasquée ?*) n'est pas affichée, ou bien parce que sa condition d'activation est fausse. Dès que la vue mère est affichée, il reçoit le message (*VueMèreAffichée ?*) et évolue à l'état *Affiché Activé* ou bien à l'état *Affiché Désactivé*, si la condition de son activation (respectivement de sa désactivation) est vraie. Puis il envoie à l'utilisateur le message (*ObjetActivé !*) ou bien (*ObjetDésactivé !*), ce qui signifie que le bouton est affiché activé (l'utilisateur peut cliquer sur le bouton), respectivement désactivé (le bouton est grisé). À l'état *Affiché Activé*, l'utilisateur peut cliquer sur l'objet, ce qui est modélisé par la réception du message (*Interaction ?*). L'objet appelle la fonction associée, comme définit dans le code de l'interface. Ces fonctions sont prédéfinies dans le logiciel SCADA. Cet appel de fonction est modélisé par un ensemble de localités instantanées. Cela signifie que l'objet retourne dans un de ses états (*Masqué*, *Affiché Activé*, *Affiché Désactivé*) instantanément. À la fin de cette action, l'objet teste sa condition d'activation ou de désactivation afin d'atteindre l'un des états *Masqué*, *Affiché Activé* ou *Affiché Désactivé*.

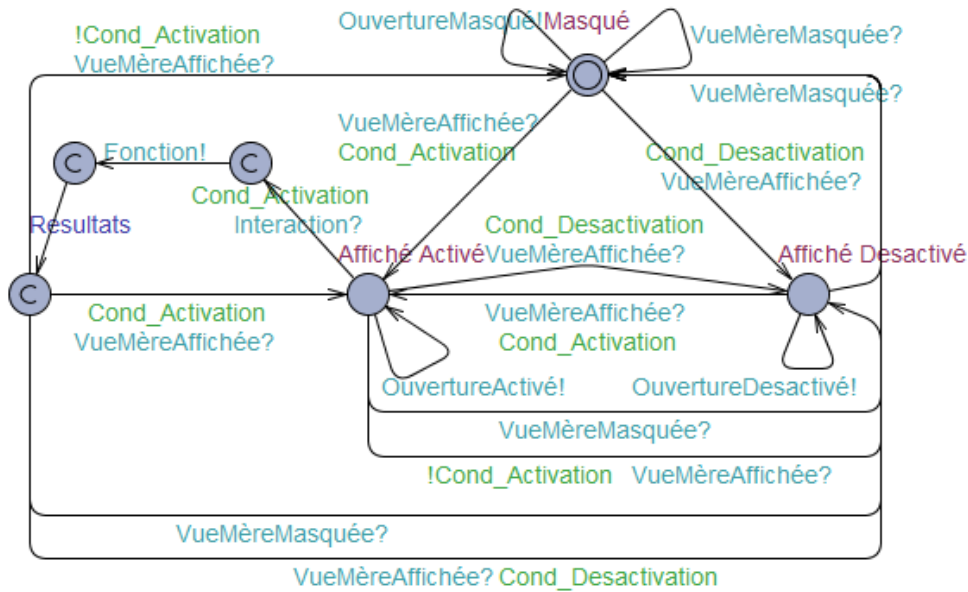


FIGURE 3.6 – Modélisation de l'IHM , Niveau exploitation : Objet de commande

L'objet informationnel. Il est caractérisé par deux états : *Affiché* et *Masqué* (figure 3.7). À l'état initial, l'objet est *Masqué*. L'objet est masqué soit parce que sa condition de visibilité (*Cond_Visibilité* sur la figure 3.7) est fausse ; soit parce que la vue qui le contient est masquée (réception du message *VueMèreMasquée*). Dans cet état, il envoie le message *ObjetMasqué!*. Ce dernier est capturé par l'automate décrivant le comportement de l'utilisateur et signifie que l'objet est masqué pour l'utilisateur. Si la condition de visibilité devient vraie et que la vue mère est affichée, l'objet passe à l'état *Affiché*. Dans cet état, il envoie à l'utilisateur le message *ObjetAffiché!* qui signifie que l'objet est affiché à l'utilisateur. L'objet retourne à l'état *Masqué* si sa condition de visibilité est fausse, ou bien si sa vue mère n'est plus affichée (*VueMèreMasquée?*). Notons que l'objet informationnel ne reçoit pas d'interactions des utilisateurs car il ne sert qu'à afficher des informations.

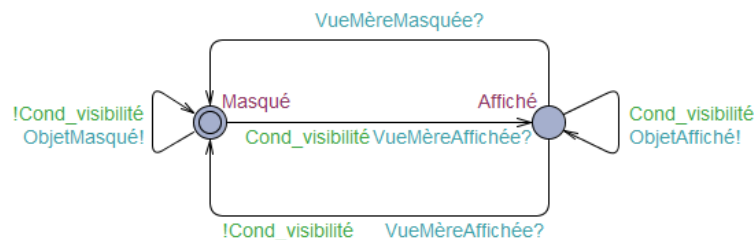


FIGURE 3.7 – Modélisation de l'IHM , Niveau exploitation : Objet informationnel

3.4.3.2 Modélisation du niveau fonctionnel du logiciel SCADA

Les actions associées aux objets de commande sont des fonctions basiques prédéfinies dans le logiciel SCADA. Nous modélisons ces fonctions par un automate temporisé et du code C dans UPPAAL. La figure 3.8 présente la modélisation de deux fonctions prédéfinies par l'automate temporisé de la figure 3.8-a et le code C correspondant à chaque fonction (figure 3.8-b). La fonction *Fonction1*, par exemple, permet d'inverser une valeur booléenne de 1 à 0 et de 0 à 1. La fonction *Fonction2* permet l'affectation d'une valeur à une variable booléenne.

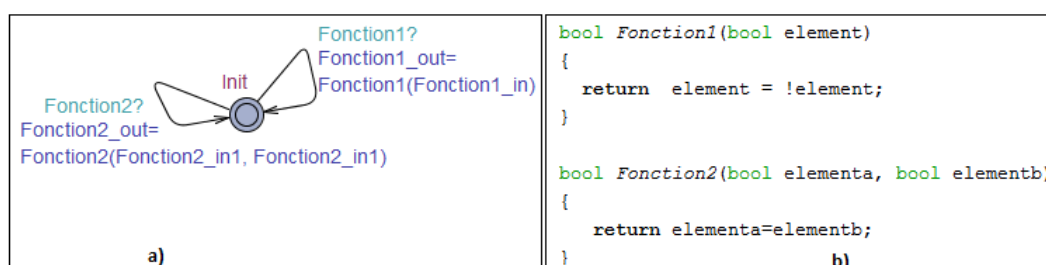


FIGURE 3.8 – Modélisation de l'IHM , Les fonctions prédéfinies : a) Automate temporisé des fonctions, b) Code C des fonctions

3.4.3.3 Modélisation de la commande

3.4.3.3.1 Le programme LD

Le langage LD ou langage à contact est un langage graphique très répandu dans la programmation des API. Le formalisme du langage LD a été inspiré des schémas électriques.

Un diagramme LD se compose de variables et d'échelons (ou *rung* en anglais, annotés R1, R2, etc. sur la figure 3.9). Les variables peuvent être des entrées qui proviennent de l'environnement ou des sorties vers cet environnement. Les rungs sont liés à une alimentation (ligne verticale tout à gauche) et se lisent de haut en bas. Chaque rung (ligne) est constitué de plusieurs éléments :

- Les contacts (entrées), comme FdcC et CtrlO sur la figure 3.9, permettent de lire une variable booléenne. Le contact peut être normalement ouvert ou normalement fermé. Un contact normalement ouvert est représenté par deux barres verticales (FdcC sur la ligne R1), et signifie qu'il est fermé si la variable booléenne associée est vraie, sinon, il est ouvert. Un contact normalement fermé est représenté par deux barres verticales et un slash ("/") entre les deux barres (CtrlO sur la ligne R2), signifie que le contact est ouvert si la variable booléenne associée est vraie, sinon, il est fermé.
- Les bobines (sorties), comme StC et StFCmd sur la figure 3.9, servent à écrire la valeur d'une variable booléenne.
- Les blocs fonctionnels, par exemple TON2 sur la figure 3.9, permettent l'exécution de fonctions plus avancées (temporisateurs, compteurs, opérations arithmétiques, etc.)

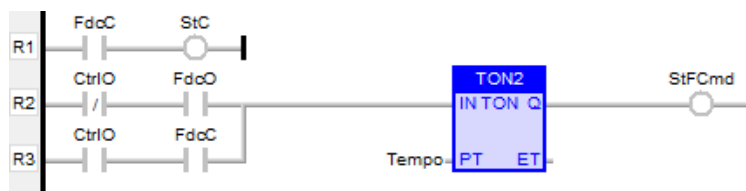


FIGURE 3.9 – Exemple d'un programme LD

Les éléments en séries (ET logique) sont sur la même ligne, a contrario, les éléments en parallèles (OU logique) sont sur plusieurs branches. Par exemple les lignes R2 et R3 de la figure 3.9 sont deux branches en parallèles, où chaque branche regroupe des éléments en séries (CtrlO et FdcC sur la deuxième branche). Chaque rung peut s'écrire sous la forme d'une équation logique. Par exemple, la ligne R1 peut s'écrire $StC := FdcC$.

3.4.3.3.2 Génération automatique des AT à partir du programme LD

L'approche est inspirée des travaux de Mokadem et al. [2010], Sarmiento et al. [2008] et de Bender et al. [2008]. Initialement, nous modélisons manuellement en automates temporisés un sous-ensemble de blocs fonctionnels proposés par la norme IEC 61131-3. Nous avons modélisé essentiellement des temporisateurs (TP, TON et TOF), des bascules (RS, SR) et des détecteurs de front montant (RTRIG) et descendant (FTRIG). La figure 3.10 présente les automates temporisés respectifs des temporisateurs TON et TOF.

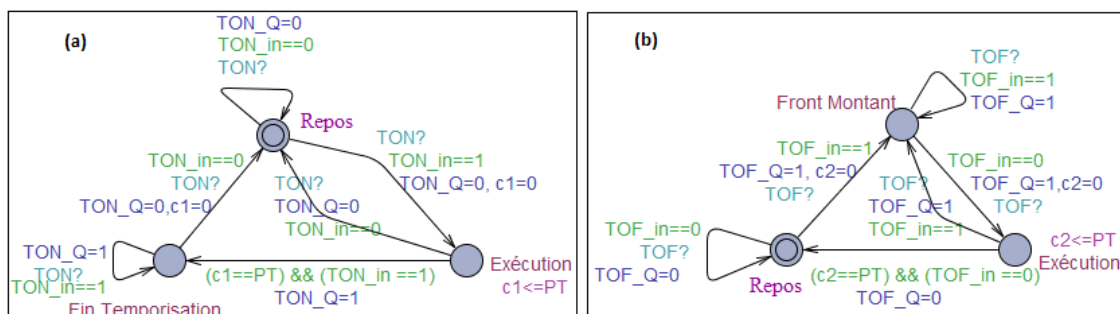


FIGURE 3.10 – Modélisation des temporisateurs ; (a) TON ; (b) TOF ;

Modélisation des temporisateurs. Le temporisateur TON a pour rôle de gérer les retards à l'enclenchement, i.e. que la sortie Q de ce temporisateur passe à 1 après une durée définie (temporisation) durant laquelle l'entrée (in) est à 1. La sortie Q passe immédiatement à 0 si l'entrée in est égale à 0.

Nous avons modélisé ce temporisateur par trois états (figure 3.10-a), deux variables booléennes (in et Q) et une horloge ($c1$). À l'état initial le temporisateur est au repos. Il reste dans cet état et maintient la sortie Q à 0 tant que in est fausse ($in==0$). Le temporisateur passe à

l'état *Traitements* si l'entrée est vraie et s'il a reçu un message (*TON?*) de la part de l'automate principal. Il reste dans cet état tant que la durée de temporisation n'est pas encore écoulée ($c1 \leq PT$), *PT* est une constante qui définit la durée de la temporisation. Une fois la durée écoulée ($c1 == PT$) et que l'entrée *in* est toujours vraie, le temporisateur atteint l'état *Fin Temporisation* et il reste dans cet état tant que l'entrée *in* est vraie. Il quitte cet état pour atteindre l'état initial si l'entrée *in* devient fausse.

Transformation des programmes LD en automates temporisés. Un cycle de l'automate est constitué des étapes de lecture des entrées, de calcul des sorties et d'écriture de celles-ci [Sarmiento et al. 2008]. Pour décrire cette exécution cyclique des API, nous proposons l'automate temporisé générique présenté à la figure 3.11.

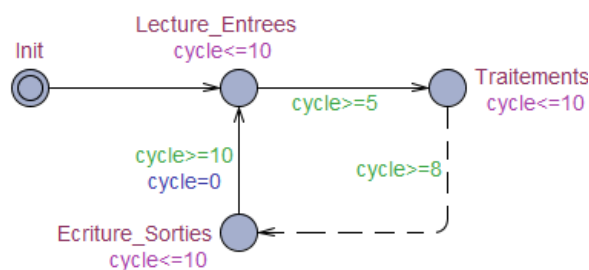


FIGURE 3.11 – Automate initial

Cet automate est constitué de trois états non immédiats (*Lecture_Entrées*, *Traitements*, *Écriture_Sorties*). L'état *Lecture_Entrées* permet de lire les entrées qui proviennent de l'environnement du programme LD à savoir l'IHM de supervision et le composant physique. L'état *Traitements* représente les différents calculs réalisés par le programme LD. À l'état *Écriture_Sorties*, les résultats des calculs sont écrits i.e. envoyés à l'environnement. Cet état marque la fin d'un cycle. Nous avons utilisé une horloge (*cycle*) afin de modéliser le temps s'écoulant au cours d'un cycle automate (*cycle*). L'état initial (*Init*) sert à initialiser les entrées lors du premier cycle.

Les règles suivantes permettent de compléter cet automate en ajoutant les informations (gardes et assignations) sur les arcs non pointillés. L'arc en pointillé, qui modélisent les différents traitements que le programme LD est censé réaliser, est remplacé par des états instantanés et des transitions générés à partir du code LD.

Des copies de chaque variable d'entrée sont créées et positionnées sur l'arc reliant l'état *Lecture_Entrées* et l'état *Traitements* (par exemple *FdcCX* est la copie de *FdcC*). *FdcCX* est une variable temporaire qui garde la même valeur durant tout le cycle. Si *FdcC* change, le changement ne sera pris en compte qu'au début du cycle suivant.

Les sorties sont transformées en expressions logiques et positionnées sur l'arc reliant l'état *Écriture_Sorties* et l'état *Lecture_Entrées*.

La communication entre l'automate principal, les temporisateurs et les blocs fonctionnels

est gérée par des messages. Au niveau de l'automate principal, deux états instantanés et trois arcs sont créés (figure 3.12). Le message permettant la synchronisation entre l'automate principal et l'automate de temporisateur est positionné sur l'arc qui relie les deux états instantanés (*exemple (TON2 !)*). Le deuxième arc relie le premier état instantané à l'élément qui le précède. Sur cet arc, sont positionnées des expressions logiques qui correspondent aux entrées du bloc. Le troisième arc relie le deuxième état avec l'élément qui le suit et porte les sorties du bloc sous forme d'expressions logiques.

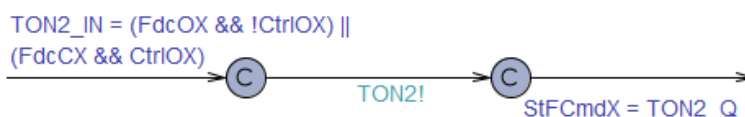


FIGURE 3.12 – Communication Automate principal et temporisateur

Après application de ces règles sur l'exemple de la figure 3.9, nous obtenons l'automate temporisé présenté sur la figure 3.13. Dans l'état initial (*Init*), les sorties sont initialisées. Ensuite le contrôleur atteint l'état *Lecture_Entrées*, dans lequel il lit et crée des copies des entrées (*CtrlO*, *FdcC*, *FdcO*) provenant respectivement de l'interface de supervision ou du capteur de fin de course-fermeture ou du capteur de fin de course-ouverture (figure 3.13). Il met aussi à jour les entrées du temporisateur TON2 ($TON2_in = (!CtrlOX \&\& FdcOX) \parallel (CtrlOX \&\& FdcCX)$) et commence le calcul (état *Traitements*) des sorties par invocation du temporisateur TON2 avec le message (*TON2 !*). Ces calculs sont représentés par des localités instantanées, ce qui signifie que ce sont des états immédiats. Ensuite, le contrôleur atteint l'état *Écriture_Sorties* qui signifie que toutes les sorties ont été calculées. Le contrôleur évolue vers l'état *Lecture_Entrées* et il met à jour les sorties calculées (*StC*, *StFCmd*) qu'il partage avec l'interface de supervision et la vanne physique.

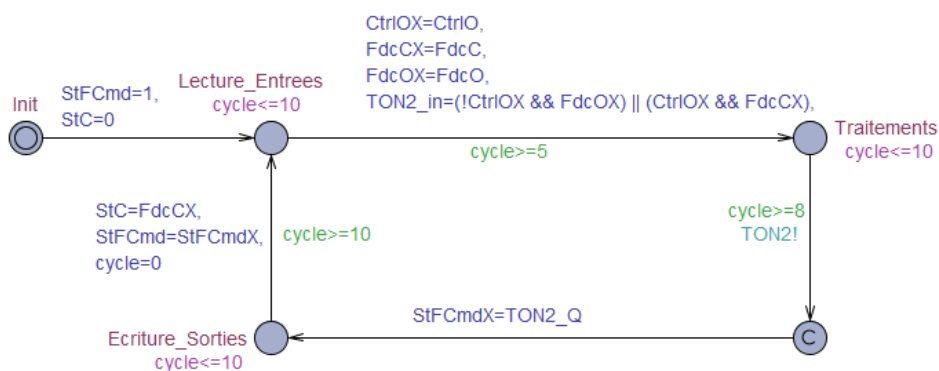


FIGURE 3.13 – Modélisation du programme LD de l'exemple 3.9

3.4.3.4 Modélisation de la partie opérative

Le composant physique est modélisé par un ensemble de localités en UPPAAL correspondant aux différents états du composant. La figure 3.14 illustre un exemple d'automate temporisé simpliste d'une vanne. Celle-ci est initialement fermée (état *Fermée*), les capteurs se chargent de transmettre son état au programme de commande. Après un ordre d'ouverture ($CmdO==1$) provenant du programme de commande, la vanne se met à s'ouvrir (état *En_Ouverture*). Elle atteint la position *Ouverte* après que le temps d'ouverture soit écoulé (par exemple $w==1$). La position ouverte de la vanne est communiquée au programme de commande à travers les capteurs. La vanne reste ouverte (état *Ouverte*) tant qu'elle n'a pas reçu un ordre de fermeture. Si cet ordre de fermeture ($CmdC==1$) est reçu, la vanne suit le même processus pour revenir à l'état *Fermée*.

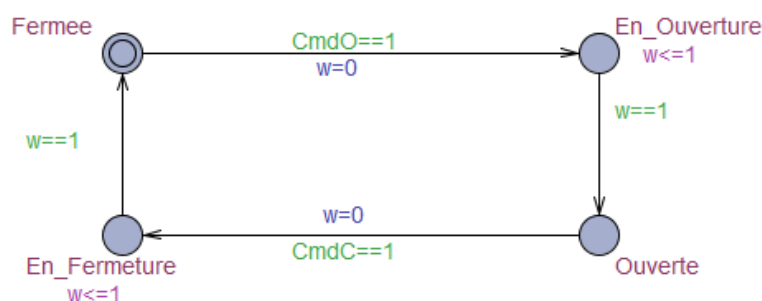


FIGURE 3.14 – Modélisation du comportement d'une vanne

3.4.3.5 Modélisation des exigences

Après modélisation du comportement de toute la chaîne de contrôle-commande élémentaire, nous présentons dans cette section, la modélisation des exigences qu'un composant de contrôle-commande doit vérifier.

3.4.3.5.1 Définition des propriétés à vérifier

Des entretiens semi-directifs ont été réalisés avec des experts afin d'extraire les exigences que chaque composant doit satisfaire. La *systematic mapping* (chapitre 2) a permis de les compléter et d'en définir les catégories. Nous classons les propriétés en trois catégories : propriétés liées à l'interface de supervision d'un composant, propriétés liées au programme de commande d'un composant et propriétés concernant tout le composant.

Propriétés de l'interface de supervision. L'utilisabilité de l'IHM peut être vérifiée formellement à travers plusieurs propriétés. Ces propriétés régissent le bon fonctionnement de

l'interface vis-à-vis de l'utilisateur et vis-à-vis du système. Selon Bolton et al. [2013], elles sont classées en 4 types :

- **Atteignabilité.** Représente ce que l'on peut faire sur l'IHM et comment on peut le faire [Aït-Ameur et al. 1999]. Dans le cas de l'IHM de supervision de V2VM, cela concerne, par exemple, la propriété : Est-il possible de commander l'ouverture de la vanne à partir de l'IHM ?
- **Visibilité.** Représente la réaction (affichage d'information sur l'interface) que l'utilisateur peut avoir. Par exemple, si l'utilisateur demande l'ouverture de la vanne, l'interface finira par afficher la vanne ouverte ou bien en défaut.
- **Reliée à la tâche.** Il existe au moins une séquence qui permet à l'utilisateur d'accomplir une tâche. Dans le cas de la vanne par exemple, pour ouvrir la vanne, l'utilisateur doit : Afficher le bandeau de commande, cliquer sur le bouton *Ouverture*, cliquer sur le bouton *Valider*.
- **Fiabilité.** Représente la manière dont l'interface fonctionne avec le système [Aït-Ameur et al. 1999]. Par exemple, l'interface ne doit jamais causer le blocage du composant.

Propriétés de la commande. La commande doit être sûre afin de ne pas conduire le composant dans un état de dysfonctionnement. Les propriétés à vérifier formellement sur les programmes sont de 5 types [Schnoebelen et al. 1999] :

- **Atteignabilité.** Une certaine situation peut être atteinte : par exemple, peut-on fermer la vanne (CmdC=1) ?
- **Sûreté.** Concerne tous les états indésirables que le système ne doit jamais atteindre. Dans notre cas, par exemple, la commande ne doit jamais envoyer simultanément une commande d'ouverture et de fermeture à la vanne.
- **Vivacité.** Représente tout ce que le système doit faire, c.à.d. que le système atteint toujours les états souhaités. Par exemple, si une demande d'ouverture est reçue (CtrlO=1) alors StO finira par arriver.
- **Absence de blocage.** La commande ne se bloque jamais.
- **Équité.** Une situation peut être atteinte (ou ne pas être atteinte) infiniment. Par exemple, la vanne sera ouverte (ou fermée) infiniment souvent.

Propriétés du composant. Les exigences concernant tout le composant sont de deux types [Juarez Orozco 2008] :

- **Qualitatives** : elles représentent le comportement désiré du composant, par l'ensemble de ses vues, tel que l'absence de blocage, sûreté, vivacité, etc. Par exemple, si la vanne est physiquement ouverte, elle doit être affichée ouverte sur l'interface de supervision.
- **Quantitatives** : elles portent sur l'évaluation des performances du composant (temps de réponse, débits, ...) et la sûreté de fonctionnement (probabilités d'erreur, taux de perte,...). Par exemple, si les deux capteurs, fin de course-fermeture et fin de course-ouverture, sont à 1 pendant plus de 2 secondes, l'alarme de défaut matériel (StFMat) se

déclenche.

3.4.3.5.2 Écriture des propriétés en CTL

Les propriétés sont écrites à l'aide de la logique temporelle CTL suivant les modalités offertes par UPPAAL. Par exemple, la propriété **P** qui stipule que le composant ne doit jamais atteindre une situation de blocage dans laquelle aucune évolution n'est possible, s'écrit en UPPAAL comme suit :

P. $A[] \textit{not deadlock}$.

Les autres types de propriétés peuvent s'écrire dans UPPAAL à travers les différentes modalités et aussi par des observateurs. Ces derniers sont le plus souvent utilisés pour la vérification des propriétés quantitatives. Un observateur dans UPPAAL encode une propriété et est modélisé sous la forme d'un automate temporisé comprenant un ensemble d'états dits d'erreurs. L'observateur est ensuite couplé (synchronisé) avec les automates du système à vérifier pour déterminer s'il existe une exécution qui mène à un état d'erreur i.e. la propriété n'est pas vérifiée. Ce principe est largement utilisé dans la littérature [Mekki 2012, Mokadem et al. 2010, Ghazel et al. 2009].

3.4.4 Vérification formelle d'une chaîne de contrôle-commande élémentaire

Les automates temporisés et les propriétés CTL, issus des étapes précédentes, sont utilisés comme entrées pour le *Model-Checker* UPPAAL. Un processus itératif est alors adopté pour corriger chaque composant standard : quand une propriété n'est pas vérifiée, on analyse le contre-exemple pour proposer une solution qui sera implémentée par les ingénieurs. Notre approche est ensuite utilisée afin de dériver les différents automates temporisés correspondant au nouveau code. Une deuxième phase de vérification par *Model-Checking* est alors exécutée, jusqu'à ce que le code satisfasse toutes les exigences du composant.

3.5 Conclusion

Dans le cadre d'une démarche assistée de conception d'application de contrôle-commande s'appuyant sur une bibliothèque de composants standards, nous avons proposé une méthodologie permettant de vérifier formellement et conjointement des vues de supervision et de commande des composants standards. Nous nous sommes appuyés sur un exemple illustratif concret, celui d'une vanne motorisée standardisée. L'interface de supervision, le programme de commande, le comportement de l'utilisateur et le comportement du composant matériel ont été modélisés sous la forme d'automates temporisés. Les exigences de conception ont été modélisées sous formes de propriétés en CTL et vérifiées formellement par *Model-Checking* (UPPAAL). La caractéristique multi-vues des composants standards rend compte de la complexité de la vérification tout en respectant les propriétés liées à chacune des vues.

Nous avons présenté, à travers ce travail, quatre contributions. La première consiste à proposer des automates génériques pour implémenter les opérateurs basiques d'un modèle de tâches et des règles de transformation des tâches. La deuxième porte sur la proposition d'une formalisation sous forme d'automates temporisés du fonctionnement d'une interface de supervision industrielle et d'un programme de commande LD. La troisième porte sur la génération automatique des modèles formels (AT) à partir des programmes LD, des IHM de supervision et des modèles de tâches. Finalement, nous avons proposé et illustré une approche pour la vérification formelle d'une chaîne de contrôle-commande élémentaire complète.

Les automates temporisés de l'interface de supervision et du programme de commande ont été générés à partir des modèles métiers (vues) des composants standards. Les transformations de modèles permettant cette génération automatique seront présentées au chapitre 5.

Les contributions présentées dans ce chapitre ont fait l'objet de deux publications dans les conférences internationales IFAC HMS 2016 [Mesli-Kesraoui et al. 2016a] et MOSIM 2016 [Mesli-Kesraoui et al. 2016b].

4

Vérification formelle des modèles de conception (P&ID)

Résumé : L'architecture d'un système est la colonne vertébrale qui supporte la mise œuvre fonctionnelle du système. Devant la complexité croissante des systèmes de contrôle-commande, le défi consiste à garantir la cohérence de l'architecture et la cohérence de sa mise en œuvre. Les modèles d'architectures sont souvent utilisés dans une approche de conception descendante pour générer les applicatifs. Suivant la nature du système, la description de son architecture peut varier. Ainsi dans le contexte d'un procédé industriel, cette architecture prend la forme d'un schéma P&ID. Si le langage de description d'un schéma P&ID est normalisé (ANSI/ISA-5.1), sa construction fait appel à des règles de l'art qui ne sont pas toujours formalisées et que seul l'expert métier maîtrise.

Nous proposons dans ce chapitre une approche formelle pour la vérification des propriétés essentielles que les architectures décrites par les schémas P&ID doivent respecter. Nous proposons aussi une solution pour afficher les contre-exemples retournés par l'outil d'analyse sur les diagrammes P&ID.

Sommaire

4.1	Introduction	86
4.2	Exemple illustratif : P&ID	86
4.3	Contexte et travaux connexes	87
4.3.1	Vérification des P&IDs	87
4.3.2	Les architectures logicielles	88
4.4	Approche proposée	89
4.4.1	Concepts utilisés	89
4.4.2	Vérification formelle des diagrammes P&ID	91
4.4.3	Un style architectural pour la norme ANSI/ISA-5.1	91
4.4.4	Génération des modèles Alloy à partir des P&ID	96
4.4.5	Formalisation des exigences et vérification formelles des P&ID	97
4.4.6	Aide à l'analyse des résultats	98
4.5	Conclusion	103

4.1 Introduction

La conception des systèmes de contrôle-commande est caractérisée par l'intervention de plusieurs concepteurs de domaines techniques très variés. Une étude exploratoire a montré que ces concepteurs se basent sur un diagramme technique abstrait qui détaille l'architecture du système et porte ce dernier tout au long de son cycle de développement [Bignon et al. 2013]. Ce diagramme représente un outil de communications incontournable entre les différents concepteurs. Dans l'ingénierie des procédés industriels, ce diagramme est le P&ID¹.

Définition 4.1 [McAviney et Mulley 2004] *Le P&ID est une description détaillée de l'architecture sous forme de flux de processus illustrant la tuyauterie, les composants et l'instrumentation associée au système.*

4.2 Exemple illustratif : P&ID

Un exemple de P&ID est présenté dans la figure 4.1.

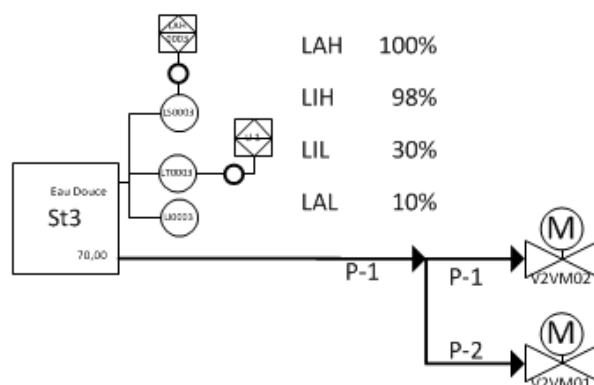


FIGURE 4.1 – Exemple de diagramme P&ID

Le P&ID est le diagramme normalisé qui schématise tous les composants et les connexions d'un procédé industriel. C'est une description abstraite de l'architecture physique du procédé. Chaque composant (exemple : St3, V2VM01 et V2VM02 sur la figure 4.1) est représenté par un symbole défini dans la norme ANSI/ISA-5.1 [ISA 1992]. Ces composants sont reliés par des connecteurs logiciels (liens avec des cercles sur la figure 4.1) et des matériels comme les tuyaux (P-1, P-2 sur la figure 4.1) et les câbles (lien simples sur la figure 4.1) [Bignon et al. 2013]. Les données échangées entre le système physique et le programme de commande (les instrumentations) sont aussi représentées dans ce diagramme.

De nos jours, les concepteurs de systèmes de contrôle-commande suivent des méthodes manuelles et informelles pour décrire l'architecture des systèmes étudiés. La diversité de ces

1. P&ID : Piping & Instrumentation Diagram

concepteurs engendre souvent une mauvaise interprétation des spécifications et des problèmes de communication qui se traduisent par des erreurs dans les modèles d'architecture (phase de conception) [Selby et Selby 2007]. L'utilisation de ces modèles d'architecture pour la génération des applicatifs dans une approche descendante ou mixte peut engendrer la propagation de ces erreurs sur les modèles détaillés et les applicatifs. La majorité de ces erreurs sont détectées tardivement dans les phases de tests [Pham 2007], ce qui augmente les coûts de re-conception et correction. Des étapes de vérification doivent être intégrées au niveau des modèles d'architectures, i.e. avant la génération automatique des applicatifs, afin de limiter les erreurs et les coûts de re-conception.

Malgré les efforts de standardisation des diagrammes P&ID par la norme ANSI/ISA-5.1, il n'existe pas une description formelle pour ces diagrammes qui permettrait une vérification automatique. Il est démontré que la vérification au niveau architectural permet de réduire significativement les coûts et les erreurs de conception [Medvidovic et Taylor 2000]. Nous présentons dans ce chapitre, une solution logicielle pour la formalisation et la vérification formelle des diagrammes P&ID.

Dans les sections suivantes, nous présentons l'état des travaux sur la vérification formelle des P&ID, y compris les techniques de modélisation et d'analyse, appliquées aux architectures logicielles. Nous nous inspirons de ces techniques pour proposer une approche automatisée pour la vérification des architectures modélisées sous forme de diagrammes P&ID. Nous utilisons le langage formel Alloy pour la spécification d'un style architectural pour la norme ANSI/ISA-5.1. Ensuite, nous générons automatiquement des modèles Alloy à partir des diagrammes P&ID. Nous utilisons les fonctionnalités offertes par l'outil d'Alloy afin de vérifier la compatibilité, la cohérence, la complétude et l'exactitude des diagrammes P&ID.

4.3 Contexte et travaux connexes

4.3.1 Vérification des P&IDs

Peu de travaux se sont intéressés à la vérification formelle des P&ID. Parmi eux, Yang et al. [2001] proposent une approche semi-automatique pour construire des modèles SMV à partir des diagrammes CDEP et P&ID. Ces modèles sont utilisés pour vérifier des propriétés de sûreté écrites en CTL et vérifiées par *Model-Checking*. [Krause et al. 2012] proposent une méthode pour extraire, à partir des diagrammes P&ID, des données de sûreté et de fiabilité pour des systèmes industriels. Ces données sont extraites sous forme de deux graphes : Netgraph et le graphe de fiabilité. Le NetGraph représente toutes les données liées aux composants et leurs connexions (structures). Par contre, le graphe de fiabilité représente les données sur la fiabilité de ces composants. Les deux graphes sont ensuite utilisés pour vérifier la fiabilité des systèmes.

Les travaux cités ci-dessus supposent que la structure architecturale soit correcte, avant d'initier la vérification formelle de la fiabilité et de la sûreté. La solution de vérification formelle proposée dans ce chapitre se place dans les phases en amont de ces travaux. En effet, nous

traitons la vérification formelle des diagrammes P&ID au niveau structurel, en nous appuyant sur la modélisation et la vérification formelle des architectures logicielles.

4.3.2 Les architectures logicielles

L'architecture est généralement spécifiée comme une **configuration** (topologie) d'un ensemble de **composants** et de **connecteurs**. Les composants représentent les unités de calcul ou de stockage de données dans le système. Ils sont caractérisés par un type, un ensemble d'interfaces pour leurs interactions avec l'environnement, une sémantique (comportement), un ensemble de contraintes, une évolution et des propriétés non-fonctionnelles [Medvidovic et Taylor 2000]. Les connecteurs (appels de fonction, protocole de communication, canalisation...) assurent l'interaction entre les composants. Comme les composants, les connecteurs sont également caractérisés par un type, un ensemble d'interfaces, une sémantique, une évolution, des contraintes et des propriétés non-fonctionnelles [Medvidovic et Taylor 2000]. L'architecture peut être décrite selon plusieurs points de vue : structurel, comportemental, physique, etc. [Oquendo 2004]. Du point de vue structurel, l'architecture est décrite par l'agencement structurel des différents composants et connecteurs constituant le système. Du point de vue comportemental, l'architecture peut être décrite par le comportement de ses connecteurs, ses composants et la façon dont ils interagissent, les actions que le système effectue et les relations entre ces actions [Oquendo 2004]. Le point de vue physique capture les composants physiques et leurs interactions à travers des connecteurs physiques.

Plusieurs langages formels ou semi-formels, appelés ADL², ou langages de description d'architecture en français, sont utilisés pour décrire les architectures logicielles. Wright [Allen 1997], par exemple, est un ADL basé sur l'algèbre de processus CSP. Π -ADL [Oquendo 2004] est basé sur le langage Π -calcul et permet la description des architectures mobiles. ACME [Garlan et al. 2010] est l'ADL de base soutenant l'échange de descriptions d'architectures.

Tous les ADL cités ci-dessus sont indépendants du domaine. En complément de ces ADL, d'autres ADLs spécifiques à un domaine existent, tels que, EAST-ADL [Debruyne et al. 2005] pour l'automobile embarquée. En ce sens, ANSI/ISA-5.1 pour la description des P&ID est un ADL spécifique aux procédés industriels. Cependant, les diagrammes P&ID avec ANSI/ISA-5.1 manquent de définition formelle, comme EAST-ADL.

Nous proposons, dans les sections suivantes, une description formelle de la norme ANSI/ISA-5.1 à travers la définition d'un style architectural. Ce style architectural fournit un vocabulaire de représentation commun et des règles pour les architectures décrites en termes de P&ID [Taylor et al. 2009]. Le style est ensuite utilisé pour vérifier plusieurs propriétés sur les diagrammes P&ID.

2. ADL : Architecture Description Languages

4.4 Approche proposée

Dans cette section, nous présentons un aperçu des concepts utilisés comme le langage formel "Alloy" et son outil que nous avons utilisé dans notre approche de vérification formelle des diagrammes P&ID.

4.4.1 Concepts utilisés

Le langage Alloy a été utilisé pour spécifier des styles architecturaux [Kim et Garlan 2010, Wong et al. 2008] et pour modéliser et vérifier des modèles d'architectures [Brunel et al. 2014, Khoury et al. 2010]. Les résultats de notre *systematic mapping* (chapitre 2) montrent que les solveurs SAT et SMT sont les plus utilisés ces dix dernières années. Pour cela, nous avons choisi d'utiliser le langage Alloy, qui est basé sur les solveurs SAT, et son Analyseur pour vérifier les diagrammes P&ID.

4.4.1.1 Le langage Alloy

Alloy est un langage formel déclaratif et structurel basé sur les relations et la logique du premier ordre [Jackson 2002]. L'idée derrière Alloy est de permettre la modélisation du système avec un modèle abstrait et minimaliste qui représente seulement les fonctionnalités importantes du système (micro modèle).

La logique d'Alloy. Elle est basée sur les concepts d'atomes et de relations. L'atome représente toutes entités élémentaires caractérisées par un type. La relation est un ensemble de tuples reliant des atomes. Ces relations sont combinées avec des opérateurs pour former des expressions. Il existe trois types d'opérateurs :

- Les opérateurs d'ensembles tels que l'union (+), la différence (−), l'intersection (&), l'inclusion (*in*) et l'égalité (=) ;
- Les opérateurs relationnels tels que le produit (−>), la jointure (.), la transposée (∼), la fermeture transitive (∧) et la fermeture transitive réflexive (*) ;
- Les opérateurs logiques comme la négation (!), la conjonction (&&), la disjonction (||), l'implication (−>) et l'équivalence (↔) .

Les contraintes Alloy sont formées à partir d'expressions et d'opérateurs logiques. Les contraintes quantifiées ont la forme : $Qx : e | F$, avec F une contrainte sur l'ensemble x , e une expression sur les éléments du type x et Q un quantificateur qui peut prendre les valeurs : **all** (chaque élément dans x), **some** (au moins un élément), **no** (aucun élément), **lone** (au plus un élément) et **one** (exactement un élément). Par exemple, **all** $x : e | F$ est vraie si tous les éléments de x satisfont F .

Les déclarations dans Alloy sont des relations sous la forme : `relation: expression`, où `expression` est une contrainte qui limite les éléments de la relation. Par exemple, $R : An \rightarrow mB$ avec $n, m \in \{\text{set (ensemble d'éléments), one, lone, some}\}$ et A, B des ensembles d'atomes. La

relation R définit que chaque élément de l'ensemble A est lié à m éléments de l'ensemble B et que chaque élément de l'ensemble B est lié à n éléments de l'ensemble A .

Le modèle Alloy. Les modèles en Alloy sont organisés en modules. Ces derniers regroupent un ensemble de signatures, de contraintes et de commandes.

Une signature, déclarée par (**sig**), introduit un ensemble d'atomes et définit les types de base, à travers un ensemble d'attributs et de relations avec d'autres signatures. Elle correspond à la notion de classe dans la modélisation orientée objet dans le fait qu'elle peut être abstraite et peut hériter d'autres signatures [Jackson 2012].

Les contraintes organisées en faits (**fact**), prédicats (**pred**), fonctions (**fun**) et assertions (**assert**) [Jackson 2012], restreignent l'espace des instances du modèle. Le **fact** est une expression booléenne que chaque instance du modèle doit satisfaire. Le **pred** est une contrainte réutilisable qui retourne des valeurs booléennes (vrai ou faux) lorsqu'elle est invoquée. Une **fun** est une expression réutilisable qui peut être invoquée dans le modèle. Elle peut avoir des paramètres et elle retourne des valeurs booléennes ou relationnelles. Une **assert** est un théorème sans argument qui nécessite une vérification.

Les commandes décrivent le but de la vérification. Les commandes peuvent consister à vérifier une assertion (checking) avec la commande **check**, ou bien simuler le modèle (avec la commande **run**) pour une solution (une instance) qui satisfait toutes les contraintes.

4.4.1.2 L'analyseur Alloy

Le langage Alloy est supporté par un outil de vérification automatique nommé l'analyseur Alloy [AlloyAnalyzer]. Ce dernier peut être utilisé comme un simulateur (commande **run**) ou comme vérificateur (commande **check**) [Jackson 2012]. L'architecture de l'analyseur Alloy est composée de trois couches. La première couche consiste à transformer les modèles Alloy en formules de la logique du premier ordre. Ensuite, des solveurs SAT (kodkod, SAT4J ...) sont utilisés pour évaluer les différentes formules. Un solveur SAT est un logiciel qui prend comme entrée une formule propositionnelle et détermine si la formule est satisfaite, c'est-à-dire qu'il existe une affectation de variables qui fait que la formule est évaluée comme vraie [Benavides et al. 2010]. La solution générée par le solveur SAT est analysée et traitée dans la deuxième couche de l'analyseur Alloy. Ensuite, dans la troisième couche, cette solution est affichée sur l'interface graphique de l'outil, sous forme d'instance dans le cas de la simulation ou d'un contre-exemple dans le cas de la vérification.

L'analyseur a été initialement inspiré des techniques du *Model-Checking*. À la différence des *Model-Checkers*, cet outil nécessite un scope pour limiter (bound) l'espace de recherche. Le *scope* représente le nombre d'itérations maximum pour trouver la solution. Le scope présente l'avantage de maîtriser l'explosion combinatoire, en réduisant l'espace de recherche, mais aussi la limite de la généralisation de la preuve. En effet, si un modèle est vérifié pour un scope x , le résultat ne peut pas être généralisé pour un scope supérieur à x .

4.4.2 Vérification formelle des diagrammes P&ID

Comme le P&ID est un diagramme architectural, il doit être complet, cohérent, compatible avec un style architectural et respecter ses exigences initiales [Taylor et al. 2009]. À cette fin, nous proposons une approche formelle sur quatre étapes pour la vérification formelle des diagrammes P&ID (Figur 4.2). Dans la première étape, nous formalisons les diagrammes P&ID comme un style architectural avec le langage Alloy. Nous utilisons le simulateur d'Alloy pour vérifier la cohérence du style (figure 4.2). Dans la deuxième étape et afin de vérifier les modèles architecturaux saisis sous forme de diagramme P&ID, nous générons à partir de ces diagrammes, en utilisant l'IDM, un modèle Alloy formel. Dans la troisième étape, nous vérifions la compatibilité des modèles générés avec le style défini dans la première étape, sa complétude, sa cohérence et son exactitude à l'aide de l'analyseur Alloy. Dans le cas d'un contre-exemple retourné par l'outil Alloy, nous proposons, dans la quatrième étape, une solution pour le visualiser sur les diagrammes P&ID, afin de faciliter la compréhension et la correction des erreurs par les concepteurs métiers.

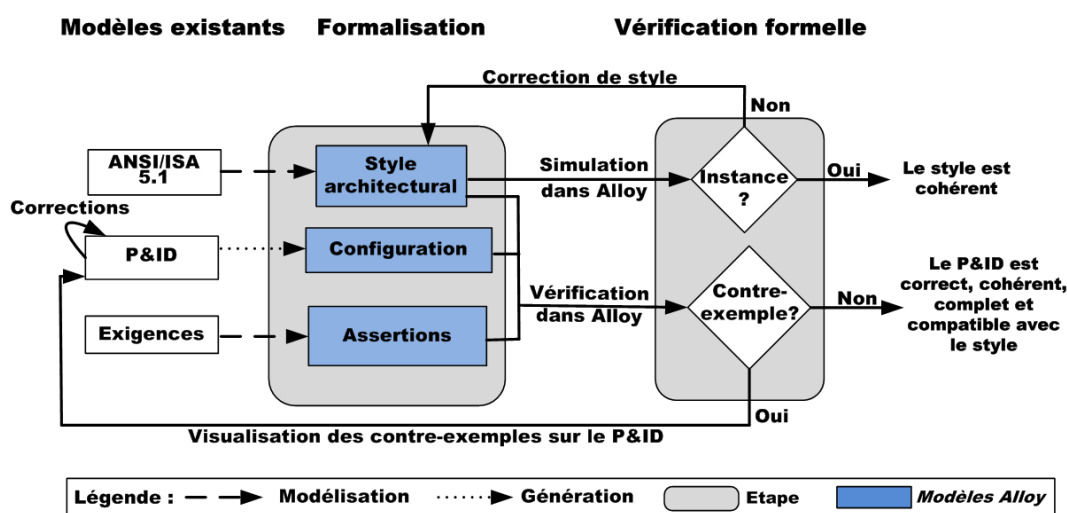


FIGURE 4.2 – Approche générale de vérification des P&ID

Deux modules ont été utilisés pour la vérification des P&ID. Le premier module, appelé bibliothèque, représente le style architectural et ses invariants. Le deuxième module, généré automatiquement à partir d'un P&ID, décrit le P&ID. Nous présentons ci-dessous la formalisation du style architectural pour les P&ID à la norme ANSI/ISA-5.1.

4.4.3 Un style architectural pour la norme ANSI/ISA-5.1

Pour modéliser formellement le style architectural des P&ID, nous nous sommes basés particulièrement sur les travaux de Kim et Garlan [2010] qui ont utilisé le langage Alloy pour modéliser et analyser les styles architecturaux de base dans le domaine du génie logiciel. Nous

avons adapté la formalisation des composants, des connecteurs, des rôles, des ports et de la configuration à nos besoins. Ensuite, nous avons étendu le style en formalisant les composants et les connecteurs spécifiques de la norme ANSI/ISA-5.1, ainsi que ses règles de conception.

4.4.3.1 Le composant

Nous modélisons les composants par les signatures `Composant` et `Port` (Listing 1). Le composant est une signature abstraite (`abstract sig`) qui contient un ensemble de ports, décrits par la relation `ports: set Port`, et un ensemble d'actions. Les actions, modélisées par la relation `actions: Port set -> set Port`, représentent l'action du composant sur le fluide à travers ses ports c.à.d. l'acheminement du fluide d'un port A vers un port B. La contrainte `actions.Port in ports` signifie que la jointure entre l'ensemble des actions et des `Port` est incluse dans l'ensemble des ports du composant. En d'autres termes, les actions dans le composant portent uniquement sur les ports de ce dernier. La signature `Port` décrit un port lié à un seul composant, modélisé par la relation `composant: one Composant`.

Listing 1

```

abstract sig Composant {
ports: set Port,
actions: Port set -> set Port
}{this = ports.composant
actions.Port in ports
actions[Port] in ports}

abstract sig Port {
composant: one Composant
}{this in composant.ports}

abstract sig Process extends
Composant{}

abstract sig Instrument
extends Composant{}

abstract sig PP extends Port{}
abstract sig PE extends Port{}
abstract sig PL extends Port{}

abstract sig V3VM extends Process{
p1: lone PP, p2: lone PP,
p3: lone PP, p4: lone PL
}{p1 + p2 + p3 + p4 = ports
actions = (p1->p2) + (p2->p1) +
(p1->p3) + (p3->p1)
lone ports&p1
lone ports&p2
lone ports&p3
lone ports&p4}

```

Il existe deux types de composants dans ANSI/ISA-5.1 : les `Process` (`abstract sig Process`) et les instrumentations (`abstract sig Instrument`, Listing 1). Les composants `process` représentent des composants qui ont un comportement impliquant un changement d'énergie, d'état ou d'autres propriétés dans le système [ISA 1992]. Dans le système de gestion des fluides, ces composants (par exemple, les pompes et les vannes) ont une action sur l'acheminement du fluide. D'autre part, des composants d'instrumentation (par exemple les indicateurs et les émetteurs) sont utilisés pour mesurer et/ou contrôler une variable [ISA 1992]. Chaque composant possède plusieurs types de ports décrivant le type de ses interactions. Nous modélisons

ces types par les signatures : PP (port type process), PI (port type instrumentation), PE (port électrique) et PL (port logiciel) qui héritent tous de la signature `Port` (Listing 1). Par exemple, un composant avec un PP peut ne interagir qu'avec des composants de type `Process`.

La vanne trois voies motorisée (V3VM), présentée à la figure 4.3, est un composant type `process` (extends `Process`) avec 3 PP (`p1`, `p2`, `p3`) et un port électrique (`p4`, Listing 1). La vanne est utilisée pour acheminer le fluide du port `p1` vers le port `p2` (`p1 -> p2`) et inversement (`p2 -> p1`). Elle permet également le routage du fluide de `p1` à `p3` et vice-versa. Cependant, elle bloque le passage du fluide du port `p2` vers le port `p3` et inversement. La contrainte (`1one ports & p1`) signifie que la vanne contient au plus un port du type `p1`. De la même manière, nous avons modélisé 30 autres composants de la norme ANSI/ISA-5.1.

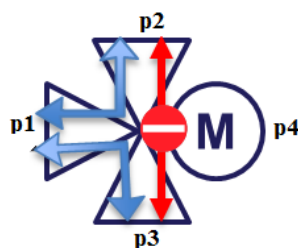


FIGURE 4.3 – Vanne trois voies motorisée (V3VM)

4.4.3.2 Le connecteur

Les connecteurs sont aussi décrits par deux signatures abstraites : `Connecteur` et `Role` (Listing 2). Un connecteur (abstract sig `Connecteur`) consiste en un ensemble de rôles. Chaque rôle (abstract sig `Role`) est lié à un seul connecteur décrit par la relation `connecté`: `one Port`.

Listing 2

```

abstract sig Connecteur {
  roles: set Role
  }{this = roles.connecteur
}

abstract sig LP extends Connecteur{}

abstract sig Role{
  connecteur: one Connecteur,
  connecté : one Port
  }{this in connecteur.roles
}

```

La norme ANSI/ISA-5.1 offre plusieurs types de connecteurs (figure 4.4). Dans le listing 2, nous modélisons les connecteurs suivants : les liens process (LP, Listing 2), liens d'instrumentation (LI), liens électriques (LE), ou liens logiciels (LL).









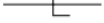

(1) INSTRUMENT SUPPLY * OR CONNECTION TO PROCESS		(6) CAPILLARY TUBE	
(2) UNDEFINED SIGNAL		(7) ELECTROMAGNETIC OR SONIC SIGNAL *** (GUIDED)	
(3) PNEUMATIC SIGNAL **		(8) ELECTROMAGNETIC OR SONIC SIGNAL *** (NOT GUIDED)	
(4) ELECTRIC SIGNAL	----- OR 	(9) INTERNAL SYSTEM LINK (SOFTWARE OR DATA LINK)	
(5) HYDRAULIC SIGNAL		(10) MECHANICAL LINK	

FIGURE 4.4 – Les connecteurs dans la norme ANSI/ISA-5.1

4.4.3.3 La configuration

Une configuration (Listing 3) est composée d'un ensemble de composants (composants : set Composant) et de connecteurs (connecteurs : set Connecteur) liés par la relation connecté (Listing 2). Donc, la configuration est une instance de toutes les signatures qui respectent les contraintes du style. Ces contraintes sont présentées dans la section suivante.

Listing 3

```

abstract sig Configuration{
  composants: set Composant,
  connecteurs: set Connecteur
}{connecteurs.roles.connecté in composants.ports
  composants.ports.~connecté in connecteurs.roles}

```

4.4.3.4 Les contraintes du style

Pour modéliser les contraintes du style, dérivées de la norme ANSI/ISA-5.1, nous utilisons les contraintes présentées dans le Listing 4. Le prédicat `pred EstDeType` retourne vrai si un élément `e1` est de type `e2`, sinon, il retourne faux. Le prédicat `pred Attaché` retourne vrai si un rôle `r` est attaché au port `p` par la relation `connecté`. Finalement, le prédicat `pred EstCompatible` détermine si un connecteur `c` est compatible avec le port `p` avec lequel il est attaché.

Différentes contraintes du style sont modélisées par le fact `fact StyleContraintes` (Listing 4), chaque élément du modèle Alloy doit satisfaire ce fact. Ces contraintes sont :

1. La compatibilité Connecteur/Port : chaque connecteur est attaché à des ports compatibles avec son type. Par exemple, un connecteur électrique (LE) ne doit être attaché qu'avec des ports électriques (PE). Cette contrainte est modélisée dans le Listing 4, avec le numéro (1).
2. La relation `connecté` est irréversible, concrètement, les extrémités de chaque connecteur ne doivent pas être attachées aux ports du même composant (le composant n'est pas connecté à lui-même), voir le Listing 4, numéro (2).

3. La connectivité des composants : chaque composant doit être connecté. Cette contrainte est codée en Alloy dans le Listing 4, numéro (3).
4. Les connexions doubles n'existent pas. En effet, il ne doit pas exister plusieurs connecteurs entre les mêmes ports (Listing 4, numéro 4).

Listing 4

```

pred EstDeType [e1:univ, e2:univ]{
  e1 in e2
}
pred Attaché [r:Role, p:Port]{
  r->p in connecté
}
pred EstCompatible [c:Connecteur, p:Port]{
  EstDeType [c, LP] => EstDeType [p, PP] else
  (EstDeType [c, LI] => EstDeType [p, PI] else
  (EstDeType [c, LE] => EstDeType [p, PE] else
  (EstDeType [c, LL] => EstDeType [p, PL]))
}

fact StyleInvariants{
  (1) all r:Role | some p:Port | Attaché [r,p] &&
    EstCompatible [r.connecteur, p]
  (2) all disj r1, r2:Role | (r1.connecteur = r2.connecteur )
    => (r1.connecté.composant != r2.connecté.composant)
  (3) all c:Composant | !(c.ports.~connecté = none)
  (4) all disj c1, c2:Connecteur | no (c1.roles.connecté & c2.roles.connecté)
}

pred show {#St>=1 #V2VM>=1 #Interface>=1}

run show for 8

```

4.4.3.5 Vérification de la cohérence du style

Le style est cohérent si et seulement si les contraintes du style ne sont pas contradictoires [Allen 1997]. Cela signifie qu'il existe au moins une configuration construite à partir des composants et des connecteurs du style et qu'elle respecte les contraintes de ce dernier [Kim et Garlan 2010]. Dans notre cas, pour vérifier la cohérence de notre style, nous utilisons le simulateur d'Alloy pour chercher une solution (configuration) qui satisfait toutes les contraintes. Si le simulateur retourne une solution, nous déduisons alors que le style est cohérent.

Nous exécutons la commande `run show for 8` (Listing 4) et nous obtenons une instance (figure 4.5 (a)). Cette instance correspond au diagramme P&ID illustré dans la figure 4.5 (b). Nous déduisons alors que notre style est cohérent.

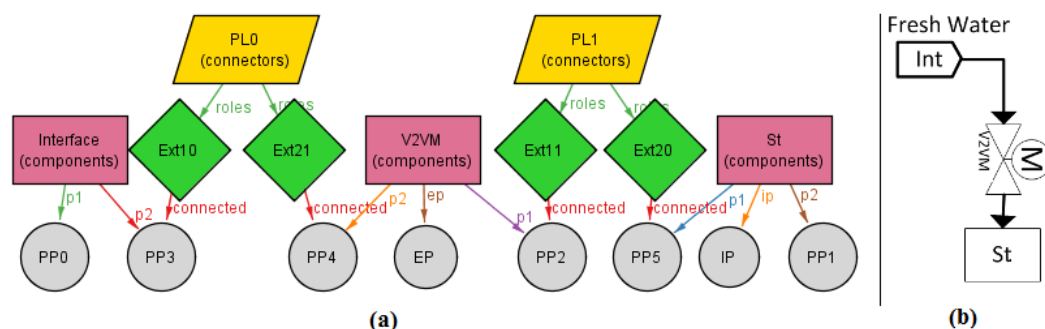


FIGURE 4.5 – La vérification de la cohérence du style : (a) L'instance retournée par Alloy ; (b) Le P&ID correspondant

4.4.4 Génération des modèles Alloy à partir des P&ID

Le diagramme P&ID n'est qu'une configuration d'un ensemble de composants et de connecteurs du style défini précédemment. Afin de vérifier un diagramme P&ID, nous proposons de le transformer en Alloy (modèle formel). Cette génération automatique permettra aux concepteurs métiers de vérifier les diagrammes P&ID sans être obligés de manipuler les formules mathématiques du langage Alloy.

À partir d'un diagramme P&ID, nous générons automatiquement une instance qui hérite de la signature `configuration` (exemple Listing 5) définie précédemment dans le style architectural. Les composants de cette configuration sont les instances qui héritent des composants définis dans le style. Par exemple, le composant `V3VM1` (Listing 5) est une instance qui hérite (`extends`) du composant `V3VM` (Listing 1). Les connecteurs de la configuration sont aussi des instances des différents connecteurs du style. Le connecteur `LP_3` est un connecteur `process` qui hérite du connecteur `LP` (Listing 2). Nous suivons le même processus pour la génération des ports (exemple `V3VM1_P1`) et des rôles (`LP_3_1`) dans le Listing 5. L'implémentation de ces transformations est reportée dans le chapitre 5.

Listing 5

```

one sig systeme extends Configuration{
  {composants = V3VM1 + ...
  connecteurs = LP_3 + ... }

one sig LP_3 extends LP{
  LP_3_1 + LP_3_2 = roles}

one sig V3VM1_P1 extends PP{
  composant = V3VM1}

one sig V3VM1 extends V3VM{
  {p1 = V3VM1_P1
  p2 = V3VM1_P2
  p3 = V3VM1_P3}

one sig LP_3_1 extends Role{
  {connecteur = LP_3
  connecté = V3VM1.p1}

```

4.4.5 Formalisation des exigences et vérification formelles des P&ID

Une analyse architecturale a quatre objectifs, à savoir : la complétude, la cohérence, la compatibilité et la correction [Taylor et al. 2009]. La complétude signifie que le modèle fournit assez d'informations pour prétendre à une analyse [Allen 1997], c'est-à-dire que le système sait toujours quoi faire. La cohérence garantit qu'il n'y a pas de contradiction entre les éléments du modèle. La compatibilité s'assure que le modèle d'architecture est conforme au style architectural et à ses contraintes. Enfin, la correction garantit que le modèle d'architecture répond aux spécifications du système. Pour atteindre ces objectifs, nous utilisons les différents outils de l'analyseur Alloy.

La complétude. La complétude dans un système porte sur la désignation des différents éléments, c'est-à-dire que les composants et les connecteurs doivent être identifiés. Cette propriété est vérifiée automatiquement par l'éditeur Alloy. Lorsque le nom de la signature générée est nul, l'éditeur d'Alloy détecte cette erreur comme une erreur syntaxique. Une autre propriété de complétude externe dans les diagrammes P&ID est l'instrumentation. L'absence de ces informations n'affecte pas le modèle d'architecture. Cependant, ces informations sont requises pendant le raffinement (par exemple, la génération automatique de l'interface de supervision à partir du P&ID). Cette exigence n'est pas vérifiée dans ce chapitre.

La cohérence. La cohérence porte sur les noms, les interfaces, les comportements, les interactions et le raffinement [Taylor et al. 2009]. Dans cette thèse, nous traitons simplement la cohérence des noms et des interfaces. La cohérence des noms signifie que les noms des composants et des connecteurs doivent être uniques pour éviter toute confusion. La cohérence des interfaces implique la cohérence entre les connecteurs et les ports, ce qui signifie que le type de connecteur doit être compatible avec le type de port auquel il est connecté. La cohérence des noms est vérifiée automatiquement par l'éditeur Alloy comme une erreur syntaxique. Par contre, la cohérence des interfaces figure parmi les contraintes du style (contrainte 1, Listing 4). Si le diagramme est compatible avec le style, alors il satisfait cette propriété.

La compatibilité. Les diagrammes P&ID doivent être compatibles avec la norme ANSI/ISA-5.1 et par conséquent, il doivent être compatibles avec le style défini dans la section 4.4.3. Pour vérifier la compatibilité des diagrammes P&ID, nous avons utilisé l'analyseur Alloy comme simulateur. Si le simulateur trouve une solution pour le modèle Alloy correspondant au diagramme P&ID et incluant la formalisation de style, cela signifie qu'il existe une compatibilité entre le diagramme P&ID et le style.

La correction. Le système dont l'architecture est décrite par les diagrammes P&ID doit répondre à des exigences fonctionnelles et non-fonctionnelles. Ces exigences doivent être élicitées et formalisées pour pouvoir être vérifiées. Nous modélisons ces exigences par les fonctions

(*fun*) et les prédicats (*pred*) dans Alloy. Nous modélisons ces exigences par des assertions et nous vérifions l'existence d'un contre-exemple par la commande *check*. Si l'analyseur Alloy ne retourne pas de contre-exemple, nous déduisons alors que le diagramme est correct. Dans le cas où un contre-exemple est retourné, nous proposons une approche automatique, que nous présentons dans la section suivante, pour la visualisation du contre-exemple sur les diagrammes P&ID afin que le contre-exemple soit compréhensible par les concepteurs métiers. Après une analyse du contre-exemple, les concepteurs pourront porter des corrections sur le P&ID afin que celui-ci devienne correct.

4.4.6 Aide à l'analyse des résultats

4.4.6.1 Le besoin

L'approche présentée dans les sections précédentes a pour objectif de faciliter l'utilisation des méthodes formelles dans un contexte industriel. En effet, la définition formelle de la norme ANSI/ISA-5.1 et la génération automatique des modèles Alloy à partir des diagrammes P&ID permettront aux concepteurs métier de bénéficier des avantages de la vérification formelle sans avoir besoin de manipulations directes des notions mathématiques. Cependant, cette approche reste incomplète si les contre-exemples retournés par l'analyseur Alloy sont incompréhensibles et par conséquent inutilisables par les concepteurs. La figure 4.6 présente un extrait d'un contre-exemple retourné par l'analyseur Alloy lors de la vérification d'un diagramme P&ID. Il est clair que la compréhension du contre-exemple impose des connaissances poussées du langage Alloy. Il devient alors nécessaire d'afficher le contre-exemple dans un langage compréhensible par les concepteurs afin de faciliter la compréhension et la correction des diagrammes P&ID.

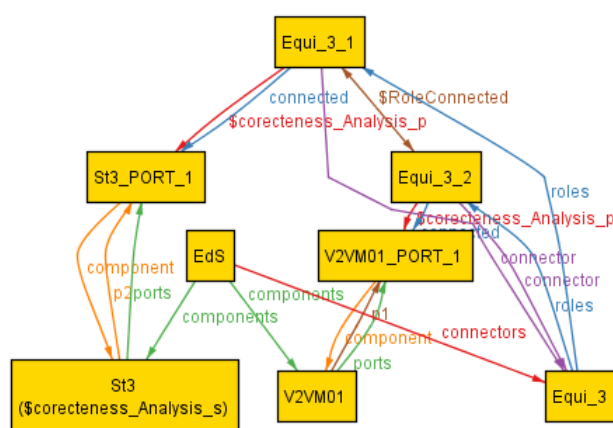


FIGURE 4.6 – Extrait d'un contre-exemple retourné par l'outil Alloy

4.4.6.2 État de l'art : visualisation des contre-exemples

Le P&ID est un langage de haut niveau que nous avons transformé en langage de bas niveau (Alloy). Le contre-exemple retourné par Alloy est en langage de bas niveau. Il est évident qu'un utilisateur d'un langage de spécification de haut niveau comme le P&ID est incapable de comprendre et d'analyser la trace d'exécution d'un langage de bas niveau (Alloy) [Aboussoror 2013].

Plusieurs travaux dans la littérature proposent la visualisation des contre-exemples retournés par les *Model-Checkers* dans un langage de haut niveau et proche de l'utilisateur. Gerking [2013] propose une approche, basée sur l'IDM pour la visualisation des traces d'erreurs générées par UPPAAL sur des modèles MechatronicUML. Gammaitoni et Kelsen [2014] proposent un outil "Lightning" pour visualiser les instances et les contre-exemples retournés par Alloy sous une forme graphique plus intuitive. Pour cela, un modèle de langage visuel basé sur Alloy a été proposé. Ce langage contient des éléments visuels de base tels que des formes et des textes qui peuvent être utilisés pour définir le visuel voulu des instances. En plus de ce langage visuel, ils proposent un langage de transformation [Gammaitoni et Kelsen 2015], basé sur Alloy, qui permet de transformer les instances et les contre-exemples Alloy en formes graphiques définies dans le modèle de langage visuel.

La généralité du modèle de langage visuel de "Lightning" lui permet d'être appliqué à plusieurs domaines d'application. Cependant, ce modèle offre un ensemble réduit de formes graphiques (cercle, rectangle,...) et il est difficilement extensible. Ce modèle ne nous a pas permis de modéliser l'aspect graphique des éléments constituant un diagramme P&ID. Il est nécessaire, alors, de définir une nouvelle approche pour visualiser les contre-exemples d'Alloy sur les digrammes P&ID.

4.4.6.2.1 Caractéristiques d'une visualisation des contre-exemples

Le contre-exemple généré par un outil d'analyse, spécialement les *Model-Checkers*, doit contenir toutes les informations nécessaires à l'utilisateur pour lui permettre d'atteindre les objectifs suivants [Aboussoror 2013] :

- Comprendre le contre-exemple dans sa globalité et son scénario. Pour atteindre cet objectif, l'utilisateur a besoin des informations sur les entités qui participent au scénario et les interactions entre elles.
- Comprendre l'erreur illustrée dans le contre-exemple. L'utilisateur doit être capable de comprendre la configuration et ses détails comme les états respectifs des entités de la configuration, ses données, ses messages...)
- Comprendre la cause de l'erreur. Pour cela, les informations sur les liens de causalité sont nécessaires.

Alloy est un langage structurel, ce qui fait que le contre-exemple retourné par Alloy n'est pas un scénario d'exécution et n'affiche pas des liens de causalité. En effet, le contre-exemple illustre seulement l'erreur. Nous traitons dans cette partie, la visualisation de la configuration

qui embarque l'erreur pour permettre à l'utilisateur de comprendre cette erreur (deuxième objectif).

4.4.6.2.2 Étude du contre-exemple Alloy

L'outil Alloy permet de sauvegarder le contre-exemple sous forme d'un fichier XML. Nous souhaitons exploiter ce fichier pour la visualisation du contre-exemple sur les P&ID. Un extrait de contre-exemple retourné par l'outil Alloy est présenté dans la figure C.1 dans l'annexe B. Le fichier contient les balises suivantes :

- Les signatures (sous le nom de *Sig*). Chaque signature est identifiée par un *ID*, un *label* (son type), un *parentID* (si elle hérite d'une autre signature). Ces signatures contiennent les atomes (instances).
- Les relations (sous le nom de *field*) décrivent les différents types de relations qui existent entre les différents atomes. Ils sont organisés en *tuples*. Chaque *tuple* représente une relation entre deux ou trois atomes.
- Les erreurs (*Skolem*) ont la même structuration que les relations. Seulement ces balises décrivent plutôt les relations indésirables (erreurs) par rapport aux propriétés vérifiées sur le modèle.
- La balise *Alloy* contient l'*instance* qui englobe toutes les balises ci-dessus et aussi le contenu du modèle Alloy (balise *source*).

Après étude du contre-exemple, nous déduisons que les informations contenues dans ce fichier permettent à l'utilisateur une compréhension partielle de l'erreur. Même si les éléments de la configuration sont affichés, l'interprétation des résultats reste une tâche difficile et nécessite des connaissances poussées dans le langage. En plus, lors de la vérification de plusieurs propriétés, le contre-exemple porte seulement sur une seule propriété. Il faudrait exécuter le modèle autant de fois qu'il y a de propriétés pour avoir tous les contre-exemples. Ce qui augmente considérablement les temps de vérification et rend cette dernière fastidieuse pour les concepteurs. Pour pallier ces problèmes, nous proposons dans les sections suivantes, une nouvelle approche pour la visualisation des contre-exemples d'Alloy sur les diagrammes P&ID.

4.4.6.3 Visualisation des contre-exemples Alloy sur les diagrammes P&ID

Nous nous sommes basés sur les travaux fondateurs de Chi [2000] pour définir notre approche de visualisation des contre-exemples Alloy sur les diagrammes P&ID. Chi [2000] propose un modèle de données manipulées tout au long d'un processus de visualisation de données : le *Data State Reference Model (DSRM)*, voir figure 4.7.

Ce modèle permet de distinguer les différents flux de données et les opérateurs qui manipulent ces données. En effet, les données sont regroupées par étage où chaque étage définit un niveau d'abstraction des données (figure 4.7). Les opérateurs permettent de transformer les données d'un étage vers un autre. La visualisation des données se fait sur trois grandes étapes :

- Transformation de données brutes en données classées (traitement, analyse, tables...).

- Transformation de données classées en structures visuelles indépendantes de toute plateforme (arbre, graphe, ...).
- Transformation des structures visuelles en vues (éléments graphiques) compatibles avec des plateformes graphiques cibles (boutons, fenêtre, ...).

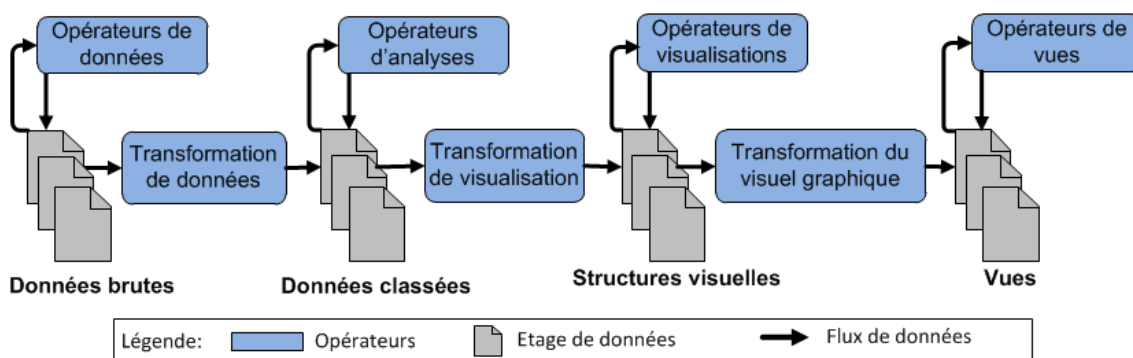


FIGURE 4.7 – Data State Reference Model (DSRM) de Chi [2000]

En plus des opérateurs de transformation, d'autres opérateurs peuvent exister dans chaque étage de données [Chi 2000]. Ces opérateurs sont : les opérateurs de données (figure 4.7), les opérateurs d'analyses, les opérateurs de visualisations et les opérateurs de vues. À la différence des opérateurs de transformations, ces opérateurs ne modifient pas la structure des données, ils permettent seulement de créer des sous-ensembles filtrés de ces données. Par exemple les opérateurs des vues permettent d'appliquer des rotations, des zooms, etc.

La séparation des préoccupations fournie par le modèle DSRM augmente la réutilisation des étages de données et des opérateurs [Aboussoror 2013]. En effet, l'étage des structures visuelles par exemple, est indépendant de toutes plateformes graphiques. Ce qui permet de définir plusieurs opérateurs de transformation qui permettent l'affichage de ces données dans plusieurs plateformes cibles.

L'approche que nous proposons pour la visualisation des contre-exemples sur les diagrammes P&ID est illustrée dans le tableau 4.1. Initialement, nous récupérons les données brutes de la vérification en interrogeant l'analyseur Alloy. Nous utilisons un opérateur de transformation pour parser les données brutes et extraire les entités (composants, connecteurs) impliquées dans l'erreur correspondant pour chaque propriété vérifiée. Ces données sont capturées sous forme d'une liste (Étage de données classées). En utilisant un opérateur de transformation visuelle, cette liste est ensuite complétée par les messages d'erreurs accompagnant chaque entité impliquée dans l'erreur. Lors de la transformation du visuel graphique, les entités impliquées sont décorées par des croix rouges et les messages d'erreurs sont transformés en zone de texte et bulles d'informations. Sur l'étage des vues, des croix rouges sont affichées sur tous les éléments (composants et connecteurs) qui violent une propriété. L'utilisateur pourra aussi utiliser l'opérateur d'affichage des messages d'erreurs pour visualiser ces derniers.

Etages	Opérateurs	Description
Données brutes		Le contre-exemple Alloy (texte)
	de transformation des données	Parser le contre-exemple et extraire des entités impliquées dans l'erreur pour chaque propriété vérifiée
Données classées		Liste des entités impliquées de chaque propriété vérifiée
	de transformation de visualisation	Compléter la liste des entités impliquées par les messages d'erreurs de chaque propriété
Structures visuelles		Liste complète des entités impliquées et message d'erreurs pour chaque propriété vérifiée
	de transformation du visuel graphique	- Créer des croix rouges pour toutes les entités impliquées - Créer des zones de textes pour les messages d'erreurs
Vues		Croix rouges sur les éléments du P&ID
	de vues	Afficher les messages d'erreurs et les bulles d'informations

Tableau 4.1 – Visualisation des contre-exemples Alloy sur le P&ID

La figure 4.8 illustre un exemple de visualisation des contre-exemples sur le P&ID. Le composant St3 n'est pas isolé en amont par des vannes, ce qui explique la présence d'une croix rouge sur ce composant. Le message accompagnant l'erreur est affiché en bas de la fenêtre. Le but de ce message est de faciliter la compréhension de l'erreur (présence des croix rouges).

4.5 Conclusion

Dans ce chapitre, nous avons proposé quatre contributions pour la vérification formelle des architectures physiques, d'un procédé industriel, modélisées par des P&ID. Premièrement, nous avons proposé de formaliser, avec le langage Alloy, la norme ANSI/ISA-5.1 sous forme d'un style architectural. Cette formalisation peut être utilisée pour la vérification de tout diagramme P&ID respectant cette norme. Deuxièmement, pour faciliter l'introduction des méthodes formelles dans l'industrie, nous avons proposé une approche pour générer automatiquement les modèles formels (Alloy) à partir des P&ID. La troisième contribution porte sur la vérification formelle des modèles générés. En effet, nous avons vérifié la compatibilité de ces modèles avec le style architectural (norme ANSI/ISA-5.1), leur cohérence, leur complétude et enfin leur respect au cahier des charges (exigences fonctionnelles et non-fonctionnelles). Nous avons aussi proposé une solution pour la visualisation des contre-exemples générés par l'outil Alloy sur des interfaces graphiques proches des concepteurs (quatrième contribution).

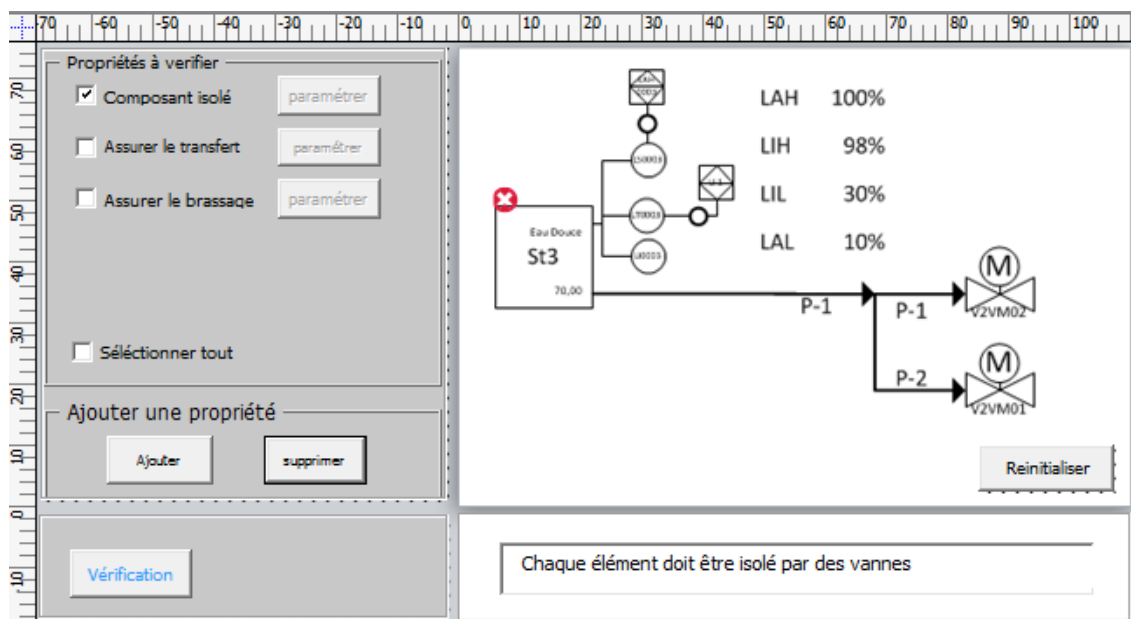


FIGURE 4.8 – Outil de vérification et de visualisation des erreurs

Les contributions présentées dans ce chapitre ont fait l'objet d'un article dans la conférence internationale ECSA (*European Conference on Software Architecture*) [Mesli-Kesraoui et al. 2016c].

Les deux propositions sur l'intégration de la vérification formelle se basent sur des modèles formels qui peuvent ne pas être aisés à construire par un expert métier. La suite du document (partie 3) s'intéresse à la mise en œuvre et notamment à l'aide à l'obtention de ces modèles.

L'implémentation des contributions sera présentée au chapitre 5. Nous présenterons, au chapitre 6, l'application de notre approche de vérification formelle des P&ID sur un cas d'étude industriel : le système d'eau douce sanitaire embarqué sur un navire.

Troisième partie

Approches de vérification formelle : mise en œuvre et applications

5

Implémentation

Résumé : Ce chapitre présente et détaille l'implémentation des approches de vérification, présentées dans les chapitre 3 et 4, sous forme de deux flots de vérifications semi-automatisés basés sur les concepts de l'ingénierie dirigée par les modèles.

Sommaire

5.1	Introduction	108
5.2	Méthodologie mise en œuvre	108
5.2.1	Concepts IDM utilisés	109
5.2.2	Outils et langages utilisés	110
5.3	Flot de vérification des composants standards	113
5.3.1	Opération de modélisation de la tâche utilisateur	113
5.3.2	Opération de transformation de HAMSTERS en AT intermédiaire (AT')	114
5.3.3	Opération de transformation des IHM SCADA en AT'	119
5.3.4	Opération de transformation des programmes LD en AT'	123
5.3.5	Opération de transformation de AT' en AT	131
5.3.6	Opération de modélisation du composant physique en AT	133
5.3.7	Opération de spécification des exigences en CTL	133
5.3.8	Opération de vérification formelle	134
5.3.9	Bilan	135
5.4	Flot de vérification formelle des diagrammes P&ID	135
5.4.1	Opération de formalisation de la norme ANSI/ISA-5.1	136
5.4.2	Opération de construction	137
5.4.3	Opération de spécification des exigences	138
5.4.4	Opération d'épuration	139
5.4.5	Opération de transformation de P&ID en Alloy	140
5.4.6	Opération de vérification	144
5.4.7	Opération de visualisation des erreurs	144
5.4.8	Opération de correction	147
5.4.9	Bilan	147
5.5	Conclusion	148

5.1 Introduction

Nous avons présenté dans les chapitres précédents deux contributions pour l'intégration de la vérification formelle dans une démarche de conception mixte (ascendante et descendante). Les approches portent, d'une part, sur la vérification d'une bibliothèque de composants standards, utilisée pour la conception des systèmes de contrôle-commande, et d'autre part, sur la vérification des modèles des architectures physiques (P&ID) de ces systèmes.

Nous rappelons que cette thèse s'intègre dans un contexte industriel, où l'utilisation et l'intégration des méthodes formelles n'est pas toujours évidente [Bennion et Habli 2014, Bjørner et Havelund 2014]. Dans les chapitres précédents (3 et 4), nous avons présenté des approches qui ont pour but de faciliter l'obtention des modèles formels et de faciliter ainsi l'utilisation de la vérification formelles dans l'industrie. La mise en œuvre de ces approches est importante pour leur introduction dans le monde industriel. C'est la troisième contribution présentée dans ce chapitre. Cette contribution porte sur l'automatisation de nos deux approches de vérification par la génération automatique des modèles formels à partir des modèles de conception.

5.2 Méthodologie mise en œuvre

La figure 5.1 résume la méthodologie mise en œuvre, visant à introduire la vérification formelle dans une démarche de conception mixte. Cette méthodologie reprend nos trois contributions majeures ; elle est donc présentée selon trois grands axes.

Le **premier axe** porte sur la vérification formelle des composants standards. Ces composants sont utilisés pour la conception des systèmes de contrôle-commande. Ils sont donc constitués d'une vue de supervision et d'une vue de commande. Le comportement de l'utilisateur face à l'IHM de supervision est capturé par un modèle de tâche.

Le **deuxième axe** consiste en la vérification formelle des diagrammes P&ID (Vérification formelle sur la figure 5.1) qui adoptent la norme ANSI/ISA-5.1.

Le **troisième axe** (génération automatique des modèles sur la figure 5.1) se focalise sur la génération automatique des modèles formels utilisés dans les deux premiers axes.

La génération automatique des modèles formels à partir des modèles de conception est basée sur l'utilisation des concepts de l'ingénierie dirigée par les modèles (IDM). Au niveau de la bibliothèque, des automates temporisés sont générés automatiquement à partir de l'IHM de supervision, des programmes LD et des modèles de tâche d'un ensemble de composants standards. Au niveau du P&ID, un modèle Alloy est généré automatiquement à partir des diagrammes P&ID.

Pour l'implémentation de la génération automatique, nous avons utilisé la bibliothèque de composants standards du projet Anaxagore ainsi que l'outil de saisie des diagrammes P&ID proposés dans ce projet.

Nous présentons, dans les sections suivantes, les concepts et les outils utilisés pour la mise en œuvre de cette génération automatique.

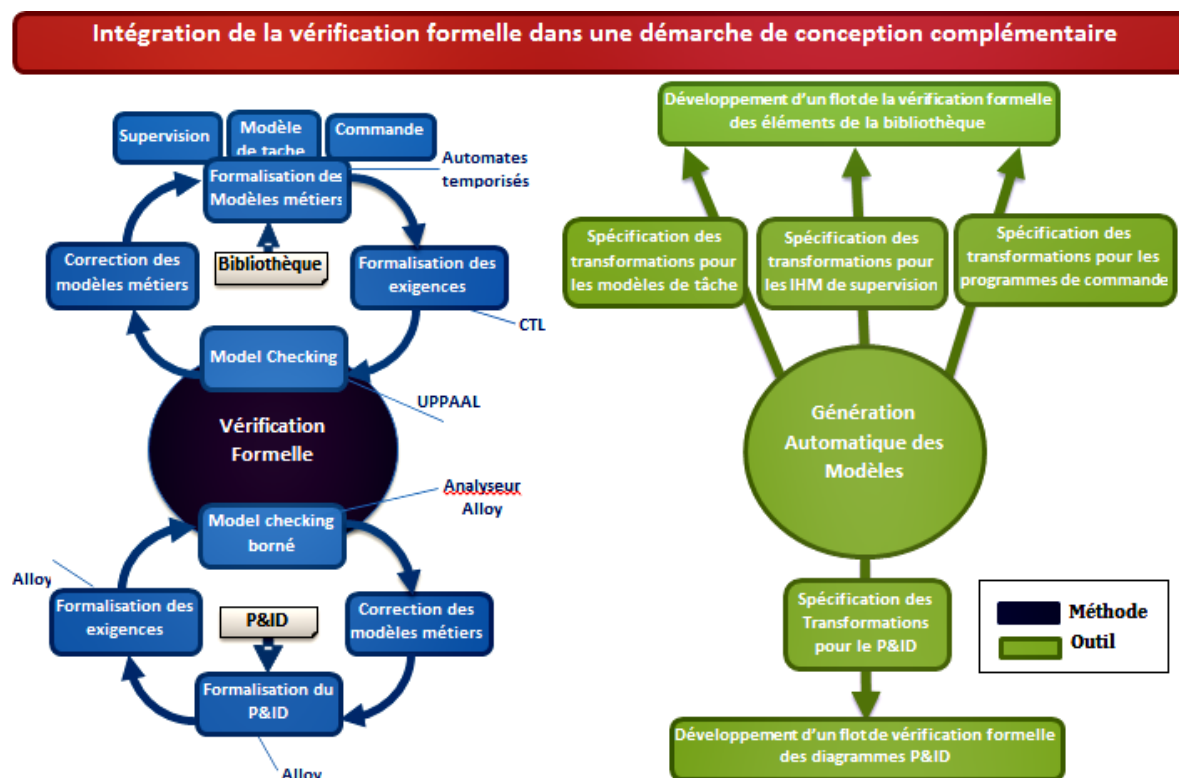


FIGURE 5.1 – Méthodologie mise en œuvre dans le projet de recherche

5.2.1 Concepts IDM utilisés

Comme évoqué précédemment, les concepts de l'IDM ont été utilisés pour la génération automatique des modèles formels. Ces concepts sont présentés brièvement dans ce qui suit.

L'IDM est une approche de conception basée sur des préoccupations plus abstraites que la programmation classique [Combemale 2008]. Dans l'IDM, une application ou une partie de celle-ci est engendrée à partir de modèles. Ces derniers sont des abstractions du système. L'activité de conception est alors décomposée en modèles pour exprimer séparément le système selon les différents points de vue portés à celui-ci. Nous retenons la définition suivante du modèle :

Définition 5.1 [Combemale 2008] *Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.*

De cette définition, on déduit qu'il existe une relation importante entre le modèle et le système qu'il représente. Cette relation est nommée dans l'IDM, *représentation*. La *représentation* stipule qu'un modèle (abstraction) représente un système réel. Le modèle est spécifié à

partir d'un méta-modèle (une syntaxe). Un *méta-modèle* est aussi un modèle (une abstraction) qui décrit clairement le langage d'expression (le vocabulaire) du modèle [Combemale 2008]. Le modèle et son méta-modèle sont liés par une relation de *conformité*. Cette relation stipule qu'un modèle doit être toujours conforme à son méta-modèle.

Ainsi, la méta-modélisation est hiérarchisée et se compose de 4 niveaux d'abstractions [Bézivin et Gerbé 2001]. Le système réel correspond au niveau le plus bas de la hiérarchie (niveau M0). Le niveau M1 correspond aux modèles qui représentent le système réel (niveau M0). Le niveau M2 est celui des méta-modèles. Le niveau M3 est celui des méta-méta-modèles. Ces derniers permettent de spécifier des méta-modèles. Pour cadrer et uniformiser la spécification des méta-modèles, l'OMG¹ propose un langage, sous forme d'un méta-méta-modèle, le MOF². Ce dernier a la caractéristique de se décrire lui-même, il est ainsi conforme à lui-même.

La spécification MDA³ complète les concepts précédents en introduisant le concept de *transformation de modèles* qui permet de générer, à partir d'un modèle source conforme à un méta-modèle source, un modèle cible conforme à un méta-modèle cible. L'objectif principal des transformations de modèles est de permettre la génération des modèles spécifiques à des plateformes à partir de modèles indépendants de toutes plateformes et ainsi favoriser l'interopérabilité [Bézivin et Blanc 2002].

5.2.2 Outils et langages utilisés

Nous nous basons sur la bibliothèque de composants standards du projet Anaxagore [Bignon et al. 2013, Bignon 2012]. Ce projet propose une bibliothèque standardisée pour la conception des systèmes de contrôle-commande reconfigurables. À des fins opérationnelles, les différentes vues des composants standards ont été implémentées sous des logiciels spécifiques. La vue de supervision a été créée sous le logiciel de supervision industrielle *Panorama E2*. La vue de commande a été programmée en langage LD de la norme IEC 61131-3 sous le logiciel *Straton*. Le choix de ces outils a été guidé par leur adoption du format d'échange XML qui favorise l'utilisation des techniques de l'IDM [Bignon 2012].

La figure 5.2 illustre les différents outils utilisés dans l'implémentation des deux flots de vérification. Ces outils sont présentés ci-après.

*Straton*⁴ est un IDE⁵ commercialisé pour le développement des programmes de commande des API. Il prend en charge les cinq langages normalisés (SFC, FBD, LD, ST, IL) de la norme IEC 61131-3. Il propose aussi une machine virtuelle permettant l'exécution des programmes de commande sur un ordinateur avant de les transférer sur des API. Il intègre aussi plusieurs protocoles de communications, comme OPC, TCP/IP..., pour faire communiquer les API avec les IHM de supervision.

-
1. OMG : Object Management Group
 2. MOF : Meta-Object Facility
 3. MDA : Model Driven Architecture
 4. <http://www.copalp.com/fr/>
 5. IDE : Integrated Development Environment

La norme IEC 61131-3 a permis de standardiser la programmation des API à travers les cinq langages normalisés. Cependant, les fournisseurs des API proposent des IDE présentant, en plus de leur côté commercialisé, des inconsistances entre leurs implémentations *Beremiz* [Tisserant et al. 2007]. Cela rend le transfert des codes de commande d'un IDE à un autre très difficile, voire même impossible (nécessite une réécriture du code). C'est dans le but de favoriser l'homogénéité et l'interopérabilité des codes de commandes que l'outil *Beremiz* [Tisserant et al. 2007] a été proposé. Cet outil est libre (ouvert) et intègre complètement la norme IEC 61131-3.

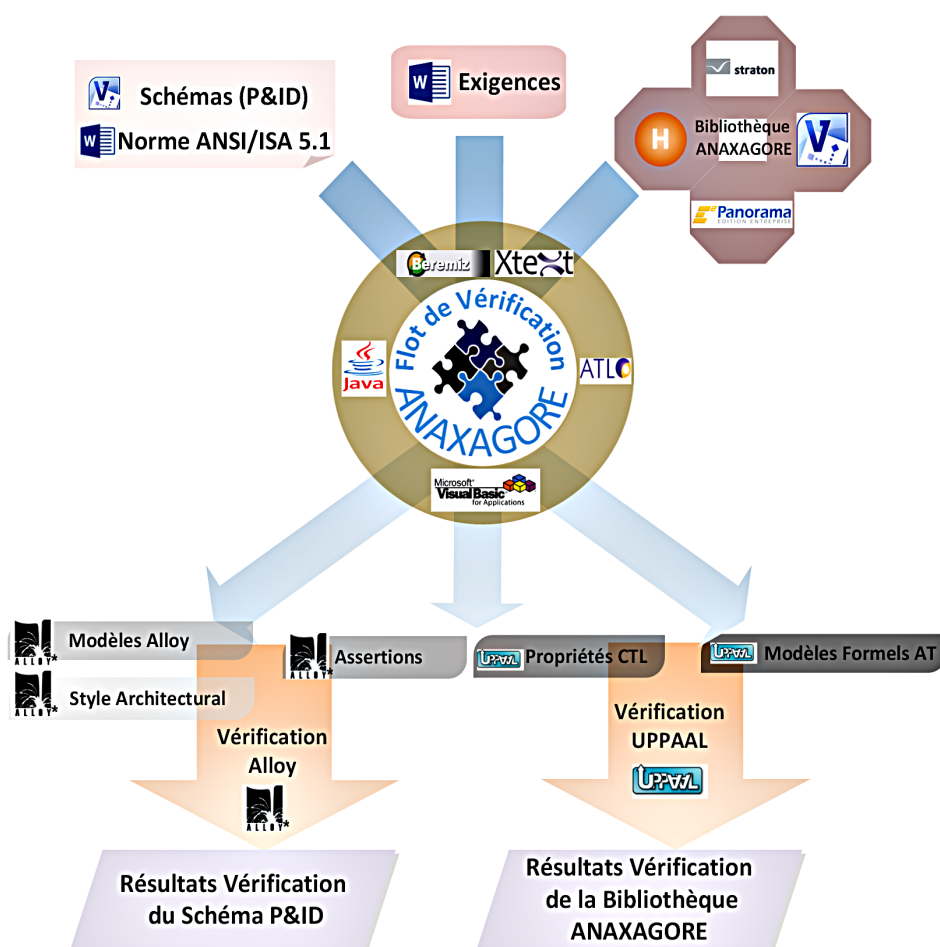


FIGURE 5.2 – Outils

Panorama E2[©], commercialisé par la société CODRA⁶, est un logiciel pour le développement des systèmes SCADA. Il offre un ensemble d'objets prédéfinis permettant à l'utilisateur de développer et d'exécuter des applications de supervision industrielles. L'architecture de Panorama est découpée en deux niveaux : le niveau exploitation et le niveau fonctionnel. Le

6. fr.codra.net/

niveau exploitation gère l'aspect graphique de l'application (widgets, alarme, etc.), en utilisant les informations provenant du niveau fonctionnel. Il permet aussi d'intercepter les interactions de l'utilisateur et d'activer les fonctions nécessaires afin d'accomplir les tâches de l'utilisateur. Le niveau fonctionnel joue le rôle d'un coordinateur entre l'interface graphique et le procédé. En effet, ce niveau est doté d'un module *Acquisition* qui a pour rôle d'établir la communication entre le niveau exploitation et le procédé. Les informations du procédé remontées par le module d'acquisition sont traitées au niveau fonctionnel puis transmises à l'opérateur à travers l'affichage ou le masquage d'objets graphiques du niveau exploitation. Lors d'une action de l'utilisateur (exemple : clique sur un bouton), le niveau exploitation demande, à travers un bus logiciel, au niveau fonctionnel l'exécution des fonctions nécessaires pour accomplir la demande de l'utilisateur. Ces fonctions peuvent être locales (niveau fonctionnel) ou peuvent être des scripts VB⁷.

Pour modéliser le comportement de l'utilisateur face à l'IHM de supervision, nous avons choisi d'utiliser la notation de tâche HAMSTERS [Barboni et al. 2010]. Ce choix est guidé par le caractère formel des opérateurs de cette notation qui sont basés sur le langage formel LOTOS [Aït-Ameur et Baron 2006]. En effet, les différents opérateurs de HAMSTERS ont des sémantiques formelles claires et précises, ce qui facilite leur transformation en automates temporisés. En plus, HAMSTERS permet la distinction entre les tâches utilisateur vers le système (*InputTask* dans HAMSTERS) et les tâches du système vers l'utilisateur (*OutputTask*) ce qui nous a permis de modéliser les interactions de l'utilisateur avec une IHM de supervision.

Microsoft[®] Visio est un outil commercialisé et largement utilisé dans la modélisation et la visualisation d'informations. Cet outil a été choisi pour la modélisation des diagrammes P&ID dans le projet Anaxagore.

ATL⁸ est un langage pour la spécification et l'exécution des transformations de modèles. Les transformations de modèles sont écrites en ATL à travers des règles déclaratives (rules). Ces dernières peuvent faire appel à des fonctions (helper) qui peuvent être récursives. Compte tenu de son applicabilité à plusieurs domaines de développement ([Bignon et al. 2013, Goubali et al. 2014], nous retenons ce langage pour le développement de toutes nos transformations de modèles.

En complément de l'IDM qui manipule des modèles (génération du code, transformation de modèle), Xtext permet la création de modèles spécifiques (DSL) [Efftinge et Völter 2006]. Il est basé principalement sur une grammaire (méta-modèle) qui définit la syntaxe (vocabulaire) du langage. En plus de la définition des éditeurs textuels pour la grammaire, Xtext permet : la génération d'un méta-modèle à partir de la grammaire et la génération des modèles XML à partir d'une syntaxe textuelle conforme à la grammaire.

7. VB : Visual Basic

8. ATL : Atlas Transformation Language

5.3 Flot de vérification des composants standards

Pour l'intégration de la vérification formelle dans une démarche de conception ascendante, nous proposons un flot de vérification composé de huit opérations (figure 5.3).

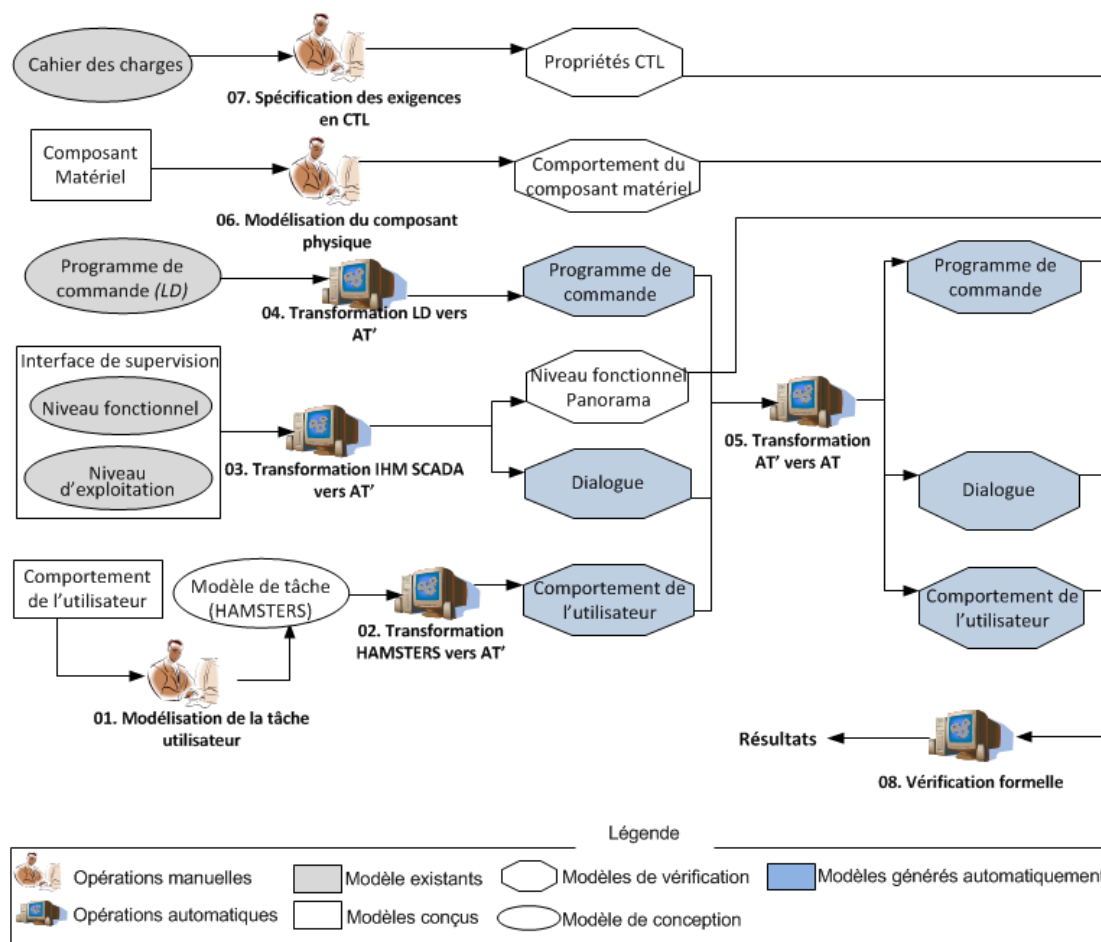


FIGURE 5.3 – Flot de vérification des composants standards

La première opération a pour objectif de capturer le comportement de l'utilisateur face à une IHM par un modèle de tâche. Cette opération manuelle est réalisée par l'expert informaticien. Lors de la deuxième opération, le modèle de tâche résultant de la première opération est transformé automatiquement en automates temporisés intermédiaires (format XML). La troisième et la quatrième opération consistent respectivement en la transformation automatique des IHM de supervision développées avec Panorama E2 et les programmes de commandes développés avec Straton, en automates temporisés intermédiaires. Ces derniers sont transformés automatiquement, lors de la cinquième opération "Transformation de AT' vers AT", en automates temporisés exploitables par l'outil UPPAAL (format textuel). La sixième opération porte

sur la modélisation manuelle du comportement du composant physique en automates temporisés. Des exigences de sûreté et de vivacité sont écrites manuellement par l'analyste en CTL. Ces exigences sont utilisées avec les automates temporisés des précédentes opérations pour conduire la vérification formelle (neuvième opération) par le *Model-Checker* UPPAAL.

Dans les sections suivantes, nous présentons chacune de ces opérations et nous détaillons les différents artefacts qu'elles manipulent.

5.3.1 Opération de modélisation de la tâche utilisateur

Dans la littérature, les modèles de tâches ont été utilisés pour modéliser le comportement de l'utilisateur face à une IHM [Bolton et al. 2013] permettant ainsi leur vérification [Bolton et al. 2011, Aït-Ameur et al. 1999, Aït-Ameur et Baron 2006].

L'opération de modélisation de la tâche utilisateur (figure 5.4), réalisée par un expert informaticien, permet de décrire sous forme arborescente les objectifs et les sous-objectifs que l'utilisateur veut atteindre sur l'interface.

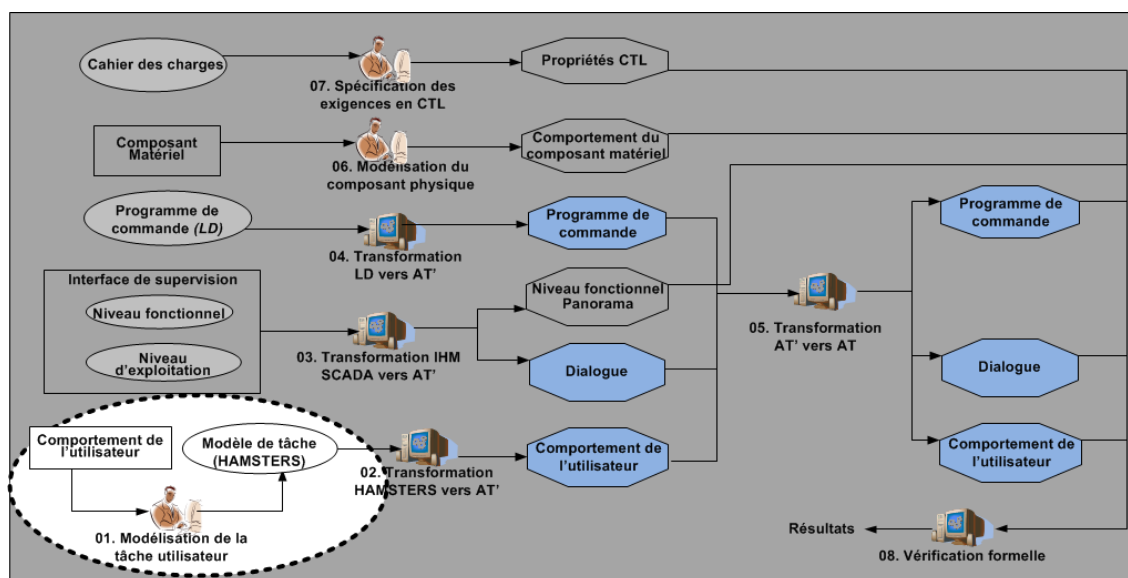


FIGURE 5.4 – Modélisation de la tâche utilisateur

En plus du caractère formel de ses opérateurs, HMASTERS intègre la majorité des opérateurs et différencie bien les tâches utilisateurs des tâches systèmes. En plus, il permet une sauvegarde du modèle de tâche en format XML, ce qui est incontournable pour l'application des concepts de l'IDM (transformation de modèles). La figure 5.5 présente un extrait de modèle de tâche avec la notation HAMSTERS.

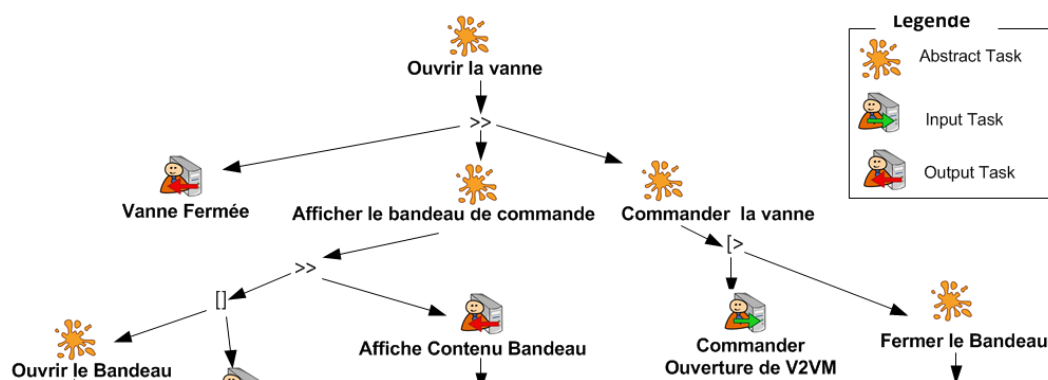


FIGURE 5.5 – Extrait de modèle de tâches HAMSTERS

5.3.2 Opération de transformation de HAMSTERS en AT intermédiaire (AT')

Cette opération a pour objectif de transformer automatiquement les modèles de tâches HAMSTERS en automates temporisés intermédiaires (figure 5.6). L'implémentation de cette opération a été réalisée par une transformation de modèle.

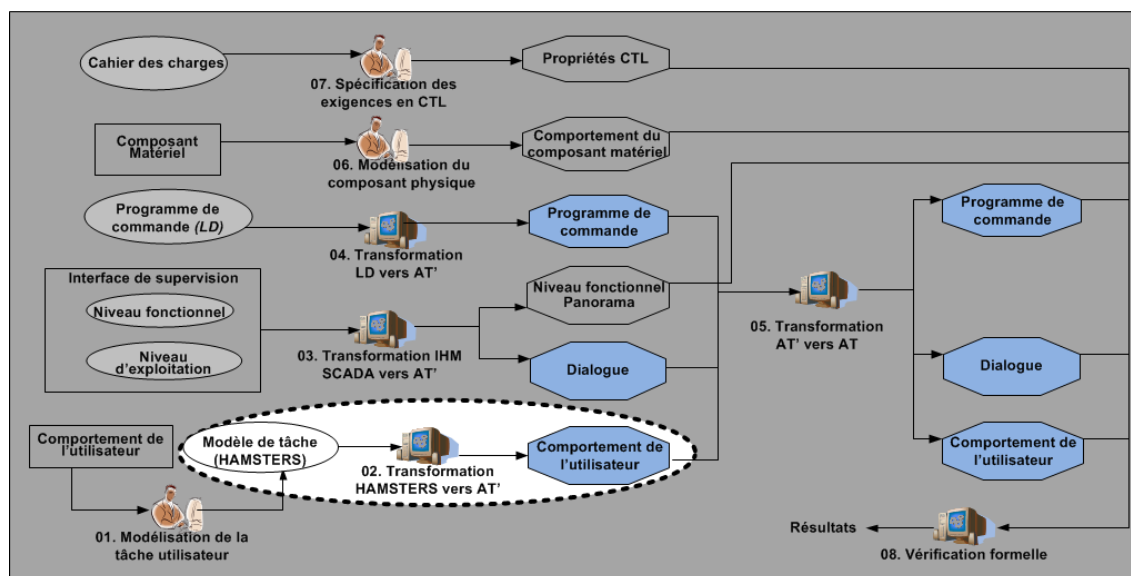


FIGURE 5.6 – Transformation de HAMSTERS en AT'

L'opération de transformation de modèle de tâches HAMSTERS en automates temporisés intermédiaires génère à partir d'un méta-modèle HAMSTERS et d'un modèle HAMSTERS, un modèle AT'(intermédiaire) conforme au méta-modèle AT'. Ces deux méta-modèles sont présentés ci-après.

5.3.2.1 Le méta-modèles HAMSTERS

Le modèle de tâches Hamsters de la figure 3.4 est sauvegardé par l'outil HAMSTERS en fichier XML. Un extrait de ce fichier est illustré ci-après.

```
<hamsters version="4">
  <nodes>
    <task copy="false" critical="0" folded="false" help="" id="66" iterative="false"
maxexectime="0" minexectime="0" name="Ouvrir la vanne" nb_iteration="0" optional="false"
type="abstract" x="41" y="-247">
      <operator id="65" knowledgeproceduraltype="" type="enable" x="89" y="-167">
        <task copy="false" critical="0" folded="false" help="" id="33" iterative="false"
knowledgeproceduraltype="" maxexectime="0" minexectime="0" name="Commander la vanne"
nb_iteration="0" optional="false" type="abstract" x="203" y="-87">
          <operator id="34" knowledgeproceduraltype="" type="disable" x="274" y="-7">
            ...
          </operator>
        </task>
      </operator>
    </task>
  </nodes>
</hamsters>
```

Après étude des modèles HAMSTERS (XML) et la description de leur notation fournie dans [Martinie De Almeida 2011], nous avons construit un méta-modèle (figure 5.7) qui engendre les modèles de tâches HAMSTERS.

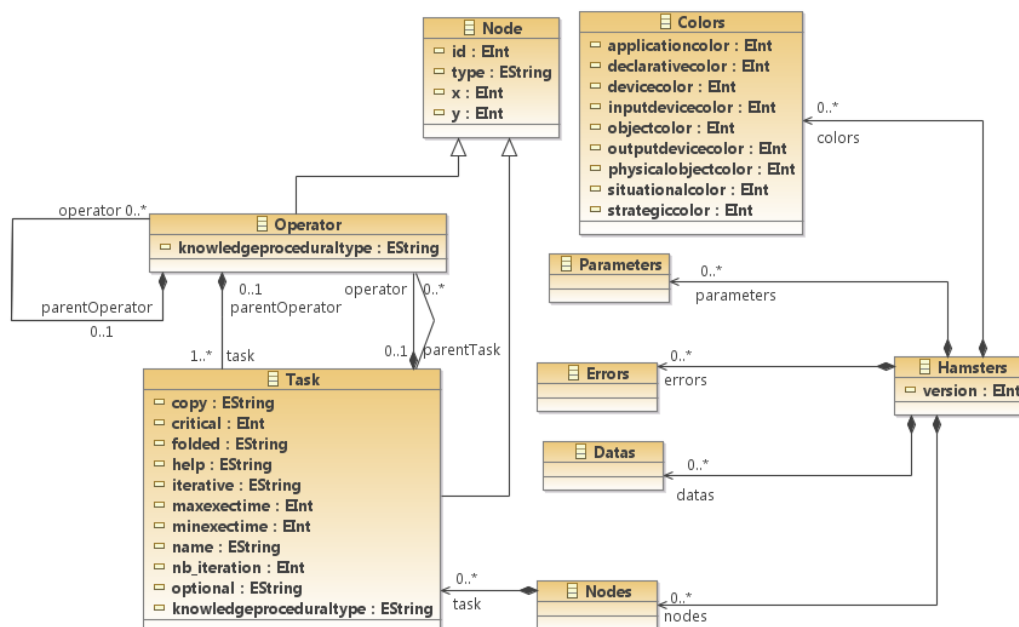


FIGURE 5.7 – Méta-modèle HAMSTERS

Ce méta-modèle est composé généralement d'une racine (Nodes sur la figure 5.7) contenant des tâches (Task). Une tâche peut contenir plusieurs opérateurs, où chaque opérateur est lié à une seule tâche maximum. Un opérateur peut aussi se décomposer en une ou plusieurs tâches, et/ou en aucun ou plusieurs opérateurs. Cette décomposition permet de construire l'arborescence du modèle. Ainsi, la racine et les feuilles de l'arborescence sont des tâches, par contre, les branches de cette arborescence sont constituées de tâches ou d'opérateurs. Par conséquent, les tâches et les opérateurs sont des nœuds d'un arbre (ils héritent de la classe Nœud) et ils sont caractérisés par un identifiant (id) et un type.

5.3.2.1.1 Le méta-modèles AT'

Les modèles générés par des transformations de modèles sont conformes à un méta-modèle (XML). Dans ce dernier, le texte est contenu dans des balises. Or, les modèles UPPAAL présentent un défaut majeur car les informations sont présentées sous forme de texte brut entre des balises XML (l'extrait ci-après), ce qui complique considérablement la transformation.

```
<declaration>
  bool BeingSet, bool CtrlC;
  broadcast chan AfficherFilsV2VM, MasquerFilsV2VM;
  broadcast chan AfficherFilsBandeau, MasquerFilsBandeau;
  ...
</declaration>
```

Cette contrainte implique nécessairement une transformation supplémentaire qui consiste en la transformation des modèles AT intermédiaires (XML) en modèles AT définitifs (exploitables directement par UPPAAL). Cette transformation fait l'objet de la cinquième opération ("*Transformation de AT' vers AT*") dans le flot de vérification (figure 5.3).

En résumé, les modèles HAMSTERS sont transformés, dans un premier temps en automates temporisés intermédiaires (AT') qui sont transformés à leur tour, dans un second temps, en automates temporisés définitifs. Nous présentons dans cette section la première transformation.

Pour transformer les modèles HAMSTERS en AT', nous avons utilisé le méta-modèle d'UPPAAL fourni dans Gerking [2013]. Ce méta-modèle (figure 5.8) propose une description plus précise des éléments UPPAAL (variable, système, expressions...).

Globalement, un modèle UPPAAL (intermédiaire) (NTA sur la figure 5.8) est composé d'une déclaration globale (globalDeclarations), d'une déclaration du système (systemDeclarations) et de plusieurs automates temporisés (template). La déclaration globale permet de définir les variables et les messages globaux. La déclaration du système permet de définir les templates et leurs instanciations. Le template correspond à l'automate temporisé et il est constitué d'un ensemble de localités (location) et d'arcs (edge). Le template peut contenir aussi des déclarations locales (declarations).

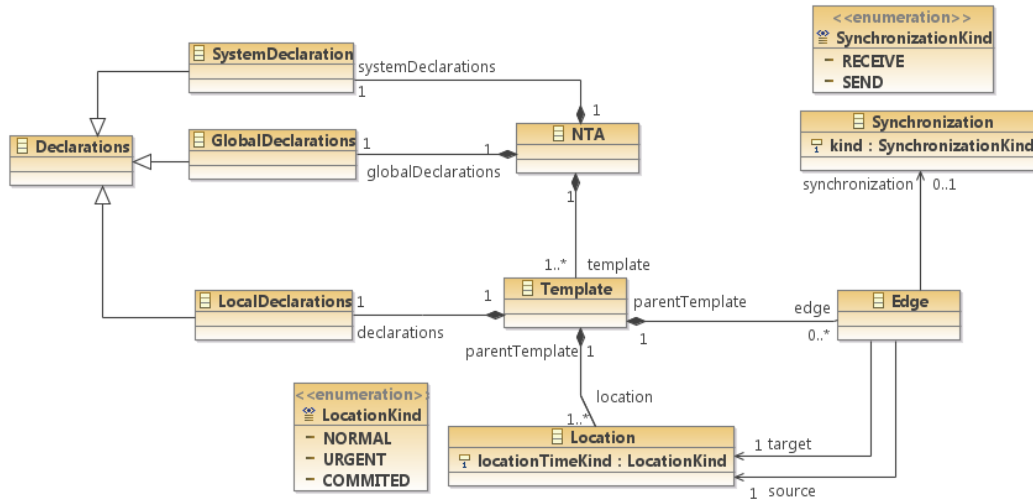


FIGURE 5.8 – Méta-modèle UPPAAL intermédiaire [Gerking 2013]

5.3.2.2 La transformation de modèle HAMSTERS en AT'

Le but de cette transformation est de transformer les tâches et les opérateurs HAMSTERS en un template composé de localités et d'arcs. L'automate temporisé résultant doit reproduire le même comportement que le modèle HAMSTERS. La transformation d'éléments HAMSTERS suit globalement les principes suivants :

Hamsters2NTA. Le nœud principal du modèle HAMSTERS (nœud `Hamsters` sur la figure 5.7) est transformé en NTA (balise principale dans les modèles UPPAAL) qui contient un `Template`. Ce dernier regroupe l'automate temporisé (ensemble de localités et d'arcs). Les `location` et les `edge` du `Template` correspondent respectivement aux localités et aux arcs générés à partir des règles ci-après.

Opérateur2Localité. Chaque opérateur est transformé en localité ou en un ensemble de localités selon le tableau 3.4 de correspondance présenté dans le chapitre 3.

Tâche2Arc. Chaque tâche est transformée en arc reliant la localité, générée à partir de son opérateur parent (`parentOperator` dans la figure 5.7), vers la prochaine localité. Le problème majeur est de déterminer la "prochaine localité". Celle-ci fait forcément référence à un opérateur, puisque les opérateurs sont les seuls à être transformés en localités. Par ailleurs, les tâches sont de deux types : tâche composée et tâche atomique. Une tâche composée, comme son nom l'indique, est composée d'autres tâches (composées ou atomiques). Une tâche atomique est une feuille dans l'arbre du modèle de tâches (elle ne contient pas d'autres tâches).

Par conséquent, la recherche de la "prochaine localité" est traitée de manière différente selon le type de la tâche (atomique ou composée).

- Dans le cas d'une tâche composée, la "prochaine localité" est tout simplement la localité créée par l'opérateur fils de cette tâche.
- Dans le cas d'une tâche atomique, il faut pouvoir remonter l'arbre pour trouver l'opérateur recherché. Pour ce faire, nous proposons un algorithme qui permet, en partant d'une tâche, de retrouver la "prochaine localité". Cet algorithme est décrit ci-dessous dans un pseudo code. Si l'on se retrouve au sommet de l'arbre (`parentOperator()` retourne "NULL") (L.5-7), l'objet retourné est la localité initiale de l'automate UPPAAL. Cette dernière est évidemment créée avant l'appel de cette fonction. Sinon, l'opérateur retourné (L.12, 15, 18, 20) par cette fonction est transformé automatiquement en localité par la règle *Opérateur2Localité*. Les tâches atomiques peuvent être des *InputTask* ou *OutputTask*. Une synchronisation est aussi générée pour chaque tâche atomique. Si la tâche est de type *InputTask*, alors la synchronisation consiste à l'envoi de message vers l'interface de supervision, sinon (*OutputTask*) la synchronisation est une réception de message à partir de l'interface.

```

1 Context Task|Operator
2   Function FindNextLocalite returns Operator
3   Begin
4       Let p = self.parentOperator();
5       If p == NULL Then
6           //le noeud Uppaal initial sur lequel on boucle
7           return initialLocalite
8       Else
9       Switch(p.type) {
10          Case 'choice': // taches optionnelles []
11          Case 'disable': //taches desactivees [>
12              return p.FindNextLocalite();
13          Case 'iterative': //taches iteratives (repetees N fois)
14          Case 'concurrent': // taches concurrentes |||
15              return p;
16          Case 'enable': //taches sequentielles >>
17              If p.getChildren().last() = self Then
18                  return p.FindNextLocalite();
19              Else
20                  return p.getChildren().nextAfter(self);
21              Endif
22          }
23      Endif
24  End

```

À l'issue de la transformation du modèle de tâche HAMSTERS de la figure 3.4, nous avons obtenu un modèle UPPAAL intermédiaire. Nous en présentons un extrait ci-après.

```

<uppaalG:NTA>
<globalDeclarations>

```

```

...
</globalDeclarations>
<template name="OuvrirIvanne" init="/0/@template.0/@location.0">
<location name="initial"/>
<location name="68" locationTimeKind="COMMITTED">
  <position x="75" y="75"/>
</location>
...
<edge source="/0/@template.0/@location.1" target="/0/@template.0/@location.3"/>
<edge source="/0/@template.0/@location.2" target="/0/@template.0/@location.0"/>

```

5.3.3 Opération de transformation des IHM SCADA en AT'

Cette section présente l'opération de transformation des IHM d'un composant standard du projet Anaxagore en AT' (figure 5.9).

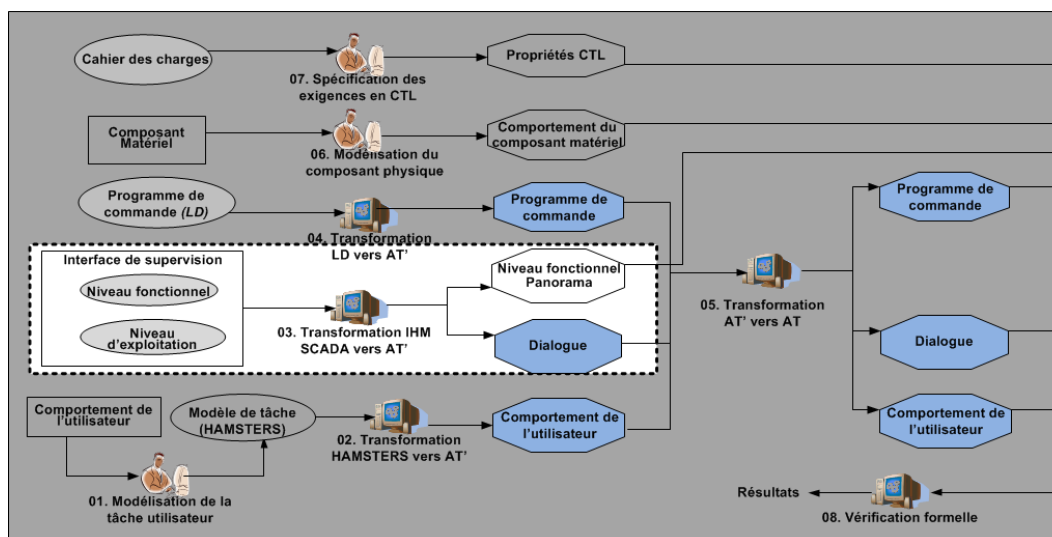


FIGURE 5.9 – Transformation des IHM SCADA en AT'

Les IHM de supervision des composants standards sont implémentées à l'aide du logiciel Panorama E2. Chaque composant (e.g V2VM) possède plusieurs sous-vues. Chaque composant possède au moins une vue portant le nom du composant (e.g V2VM) et qui comporte le symbole du composant. En plus de cette vue, certains composants standards possèdent également une vue qui gère le bandeau de commande de ces composants. Ce bandeau présente à l'utilisateur l'ensemble des actions qu'il peut effectuer sur le composant ou sur le bandeau lui-même (ex. pour une vanne "Ouvrir", "Fermer", "Valider", "Fermer bandeau" etc...). Les boutons du bandeau contrôlant l'état du composant standard (e.g Ouverture/Fermeture, Marche/Arrêt) sont superposés. Les deux boutons ayant des fonctions opposées ne sont évidemment pas activables simultanément.

5.3.3.1 Méta-modèle Panorama

Cette section décrit la structure globale du modèle d'une IHM de supervision élémentaire implémentée dans Panorama E2 et un extrait de son méta-modèle. Le méta-modèle Panorama (figure 5.10) possède à sa racine des vues graphiques (`GraphicView`) composées au plus d'une seule vue (`View`). La vue est constituée de plusieurs objets graphiques (`GraphicObjects`). Ces derniers englobent un ensemble d'items (`Item`). Les `Items` sont soit des ellipses, des rectangles, des polygones ou des boutons. Ils sont utilisés pour représenter la vue supervision. Chaque `Item` possède deux types de propriétés :

- `Animation Properties` : les propriétés liées à l'animation et l'affichage des `Item` sur l'interface de supervision.
- `Command Properties` : les propriétés liées aux commandes envoyées au programme de commande après une action de l'utilisateur.

5.3.3.2 Transformation de modèle Panorama en AT'

La transformation des modèles Panorama en AT' portent essentiellement sur la génération du module de dialogue (cd. section 3.4.3.2.1, chapitre 3) de l'IHM. L'objectif est de transformer les éléments graphiques de l'IHM en objets de commandes et objets informationnels. Les éléments qui nous intéressent le plus pour cette transformation, dans le modèle Panorama, sont les `Item`. Ces derniers sont transformés de la façon suivante :

Alarme2ObjetInformationnel. Chaque alarme Panorama est transformée en objet informationnel.

ItemInfo2ObjetInformationnel. Chaque `Item` ayant seulement une partie animation (`Animation Properties`) est transformé en objet informationnel avec les `Animation Properties` qui sont transformées en condition d'activation et de désactivation (`gard` dans le modèle AT').

ItemCmd2ObjetCommande. Tout `Item` ayant une `Animation Properties` et une `Command Properties` non vides est transformé en objet de commande. Les `animationProperties` sont transformées en conditions d'activations et de désactivations, par contre, les `Command Properties` sont transformées en appels de fonctions C. Ces dernières sont soit prédéfinies dans Panorama, soit générées à partir des scripts VB. Ces deux cas sont traités différemment et sont présentés ci-après.

D'une part, les fonctions prédéfinies dans Panorama E2 permettent de manipuler les valeurs des variables booléennes (mise à 0, mise à 1, inversion du 0 vers 1, inversion du 1 à 0, etc.). Toutes ces fonctions ont été implémentées par un automate et un ensemble de fonctions en C dans UPPAAL (figure 5.11). L'automate est composé d'un seul nœud et contient autant d'arcs qu'il y a de fonctions (figure 5.11-A). Chaque arc permet un appel de la fonction C associée et contient aussi une synchronisation (réception de message) qui permet à l'automate

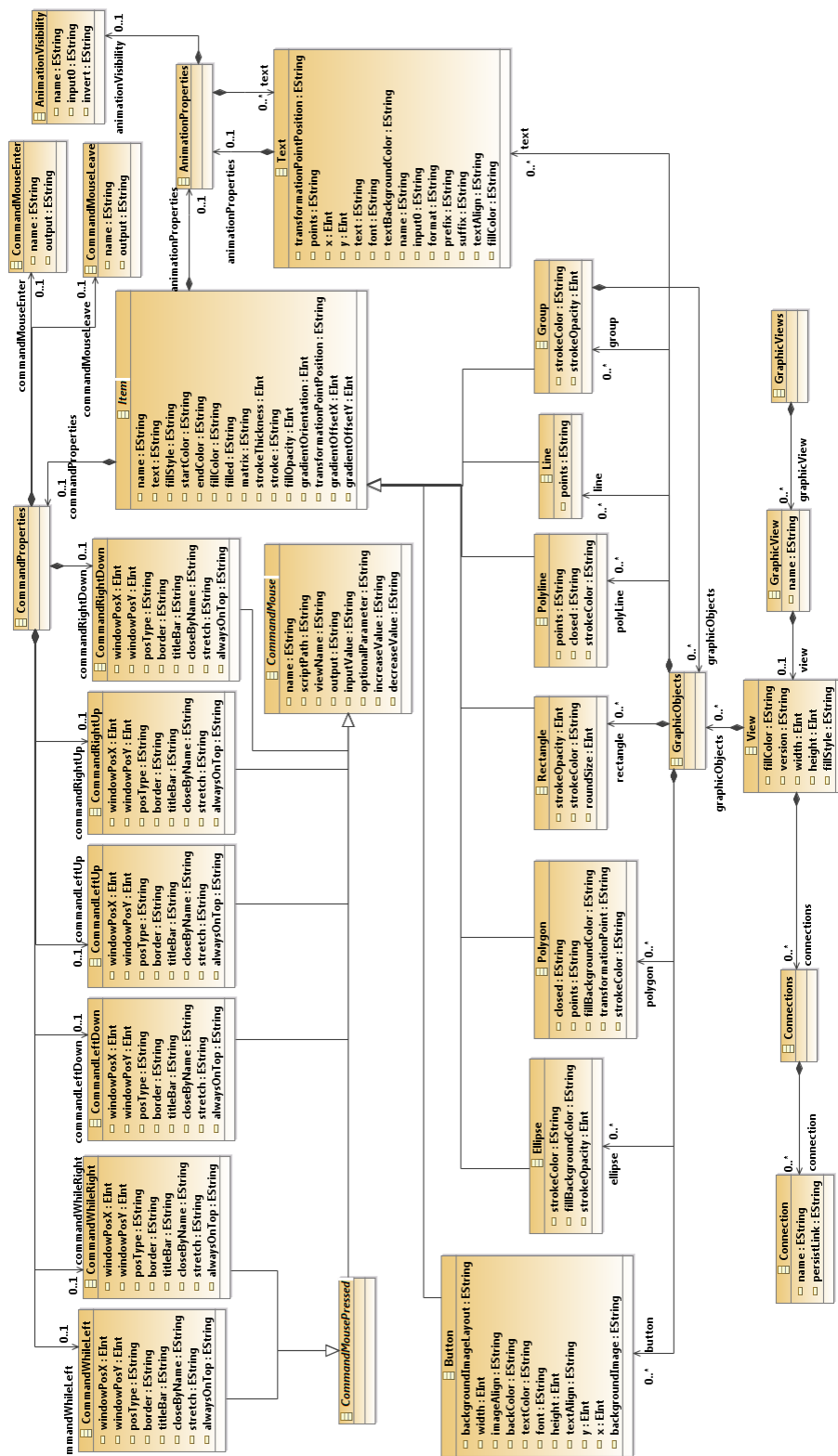


FIGURE 5.10 – Extrait du Méta-modèle Panorama E2

de se synchroniser avec les objets de commande. À la réception du message (*inversion ?*), par exemple, l'automate invoque la fonction souhaitée (*inversion*) en lui indiquant la variable à inverser (*inversion_in*) dans les paramètres (*inversion_out=Inversion(inversion_in)*). Le code C (figure 5.11-B) de la fonction est alors exécuté, permettant ainsi l'inversion de la variable booléenne *inversion_in*.

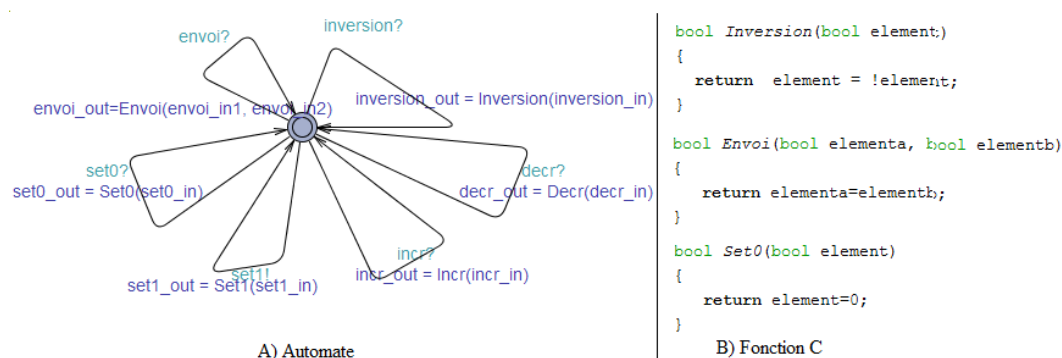


FIGURE 5.11 – Formalisation des fonctions prédéfinies dans Panorama E2 en : A) Automate temporel ; B) Fonctions C

D'autre part, les scripts VB sont préférables quand la commande est moins triviale. En revanche, ces scripts posent un problème pour la génération automatique des modèles UPPAAL à partir de l'IHM de supervision. L'automate UPPAAL généré doit évidemment prendre en compte le contenu des scripts afin d'intégrer les commandes envoyées au programme de commande.

La figure 5.12 résume le processus adopté pour l'obtention de code en langage C à partir de code VB.

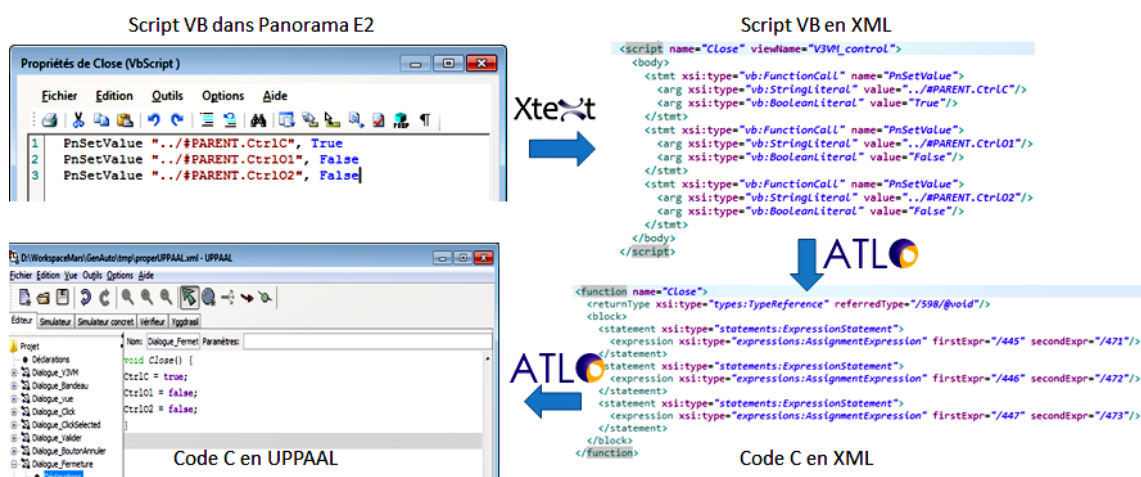


FIGURE 5.12 – Transformation des codes VB en C

Nous avons utilisé Xtext pour générer des modèles XML à partir des codes VB embarqués dans les IHM de supervision. Il s'agit ici d'écrire une grammaire VB simplifiée sur Xtext qui permet de transformer les scripts VB en modèles XML. Initialement, une grammaire qui engendre les codes VB a été proposée (en annexe). Ensuite, nous avons utilisé les fonctionnalités offertes par Xtext pour générer un méta-modèle et des modèles XML à partir d'un code VB. Le modèle XML est ensuite utilisé dans une transformation de modèle pour générer des fonctions C dans UPPAAL.

5.3.4 Opération de transformation des programmes LD en AT'

Dans cette section, nous présentons en détails la quatrième opération de notre flot (figure 5.13).

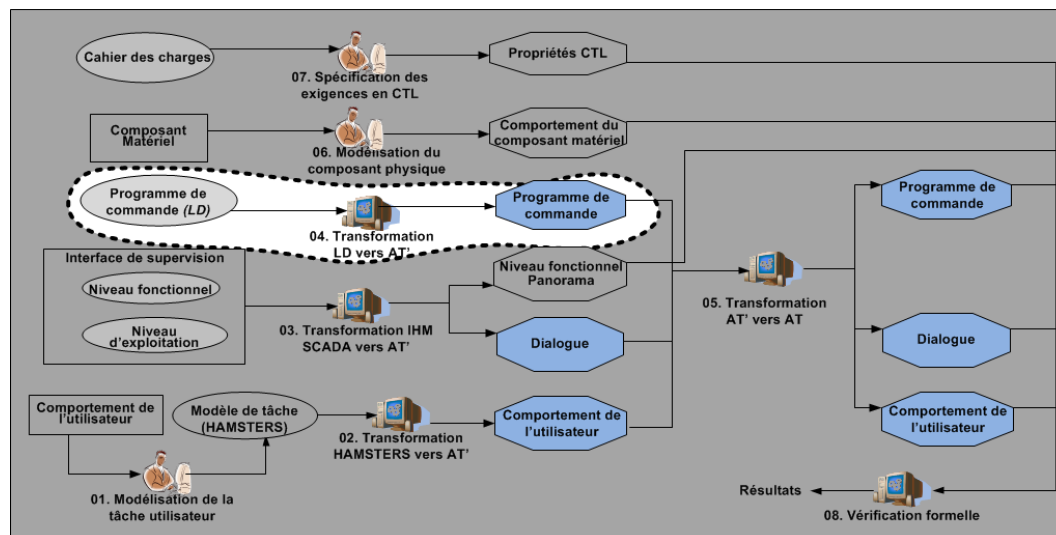


FIGURE 5.13 – Transformation des programmes LD en AT'

Le but de cette opération est de transformer les programmes de commande écrits en LD, en automates temporisés. Cette transformation impose des contraintes sur le modèle d'entrée LD. La représentation XML de celui-ci doit être conforme à la norme industrielle IEC-61131. Cependant, les programmes LD des composants standards dans Anaxagore sont saisis sous Straton, qui ne respecte pas exactement la norme IEC-61131. Le logiciel Beremiz permet quant à lui de saisir un modèle LD conforme à la norme IEC-61131. Par conséquent, la transformation des programmes de commande LD en automates temporisés est en deux étapes. La première étape consiste à transformer le modèle Straton en modèle Beremiz. La seconde permet de transformer le modèle Beremiz en automates temporisés. Cette approche permet, d'une part, de normaliser les modèles Straton et, d'autre part, de favoriser la réutilisation des transformations. En effet, dans le cas où un autre outil de programmation API est utilisé, il n'y a que la transformation permettant de transformer les modèles de cet outil vers Beremiz à écrire.

La transformation de Beremiz vers les automates temporisés peut réutilisée sans modification.

Dans les sections suivantes, nous présentons les deux méta-modèles Straton et Beremiz, ainsi que les différentes règles de transformation des modèles Straton vers des modèles Beremiz.

5.3.4.1 Méta-modèle Straton

La figure 5.14 illustre un extrait d'un modèle LD en Straton.

```
<LDrung dx="0" dy="4" sx="7" sy="2" label="">
<LDdiv dx="0" dy="4" sx="2" sy="2">
  <LDbranch dx="1" dy="4" sx="2" sy="1">
    <LDcontact kind="I" symbol="Ctrl0" dx="1" dy="4" />
    <LDcontact kind="D" symbol="Fdc0" dx="2" dy="4" />
  </LDbranch>
  <LDbranch dx="1" dy="5" sx="2" sy="1">
    <LDcontact kind="D" symbol="Ctrl0" dx="1" dy="5" />
    <LDcontact kind="D" symbol="FdcC" dx="2" dy="5" />
  </LDbranch>
</LDdiv>
<LDbox dx="3" dy="4" sx="3" sy="2" kind="FB" type="TON" inst="TON2">
  <LDboxinput pin="0" symbol="Tempo" />
  <LDboxoutput pin="0" symbol="" />
</LDbox>
<LDcoil kind="D" symbol="StFCmd" dx="6" dy="4" />
</LDrung>
```

FIGURE 5.14 – Extrait d'un modèle Straton (fichier XML)

Le méta-modèle Straton (figure 5.15) d'un programme LD est essentiellement composé d'éléments simples (SimpleElements) et d'éléments complexes (ComplexElements). Les éléments simples sont : les contacts, les lignes (Line) et les bobines (Coil). Les éléments complexes sont essentiellement des échelons (Rung), des divisions (ou point de séparation, Div), des blocs (Box) et des branches (Branch). Un Rung englobe un ensemble de Contact et un ensemble de Line. Il peut contenir au plus un Coil, et/ou au plus une Div. Les divisions contiennent une ou plusieurs branches (Branch). Chaque branche est composée d'au plus d'une bobine, d'un ensemble de contacts, d'un ensemble de lignes et d'un ensemble de Box. Ces dernières représentent les blocs fonctionnels (TON, TOF, etc.).

5.3.4.2 Méta-modèle Beremiz

Le méta-modèle Beremiz⁹ (LD sur la figure 5.16) présente une structure plus linéaire que les modèles Straton :

9. Le méta-modèle Beremiz a été généré automatiquement à partir d'un schéma XML fournit par l'organisation PLCOpen : <http://www.plcopen.org>

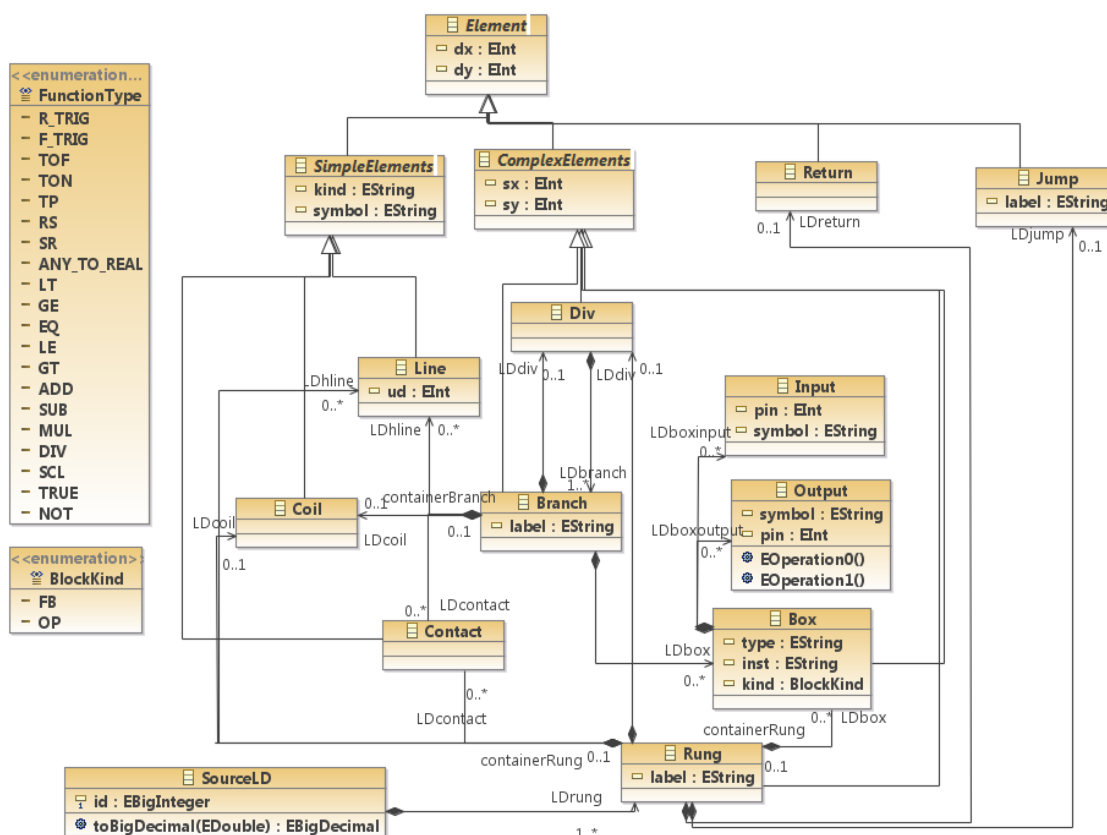


FIGURE 5.15 – Méta-modèle Straton

- Les Rung n'apparaissent pas explicitement dans le modèle. À la place, un rail d'alimentation (*LeftPowerRail*) est utilisé. Celui-ci possède des pins en sortie (*ConnectionPointOut*), auxquels peuvent se rattacher les différents éléments du diagramme LD (contacts, coils et blocs fonctionnels).
- Les éléments possèdent un attribut *localId* qui leur sert naturellement d'identifiant unique. Cet attribut est essentiel pour assurer les connexions entre les différents éléments.
- Les éléments principaux (contacts, coils et blocs fonctionnels) possèdent dans leur arborescence un attribut *ConnectionPointIn*. Cet attribut possède lui-même une ou plusieurs *Connection*, qui comme tous les autres éléments est identifié par un *refLocalId*. La valeur de cet attribut correspond à l'*id* de l'élément qui précède l'élément courant. On peut donc dire que la lecture d'un Rung sur un modèle Beremiz se fait de droite à gauche. On part du dernier élément (généralement un Coil) pour retrouver l'élément qui le précède (via l'attribut *refLocalId*), et ce successivement jusqu'à arriver au rail d'alimentation (*LeftPowerRail*).

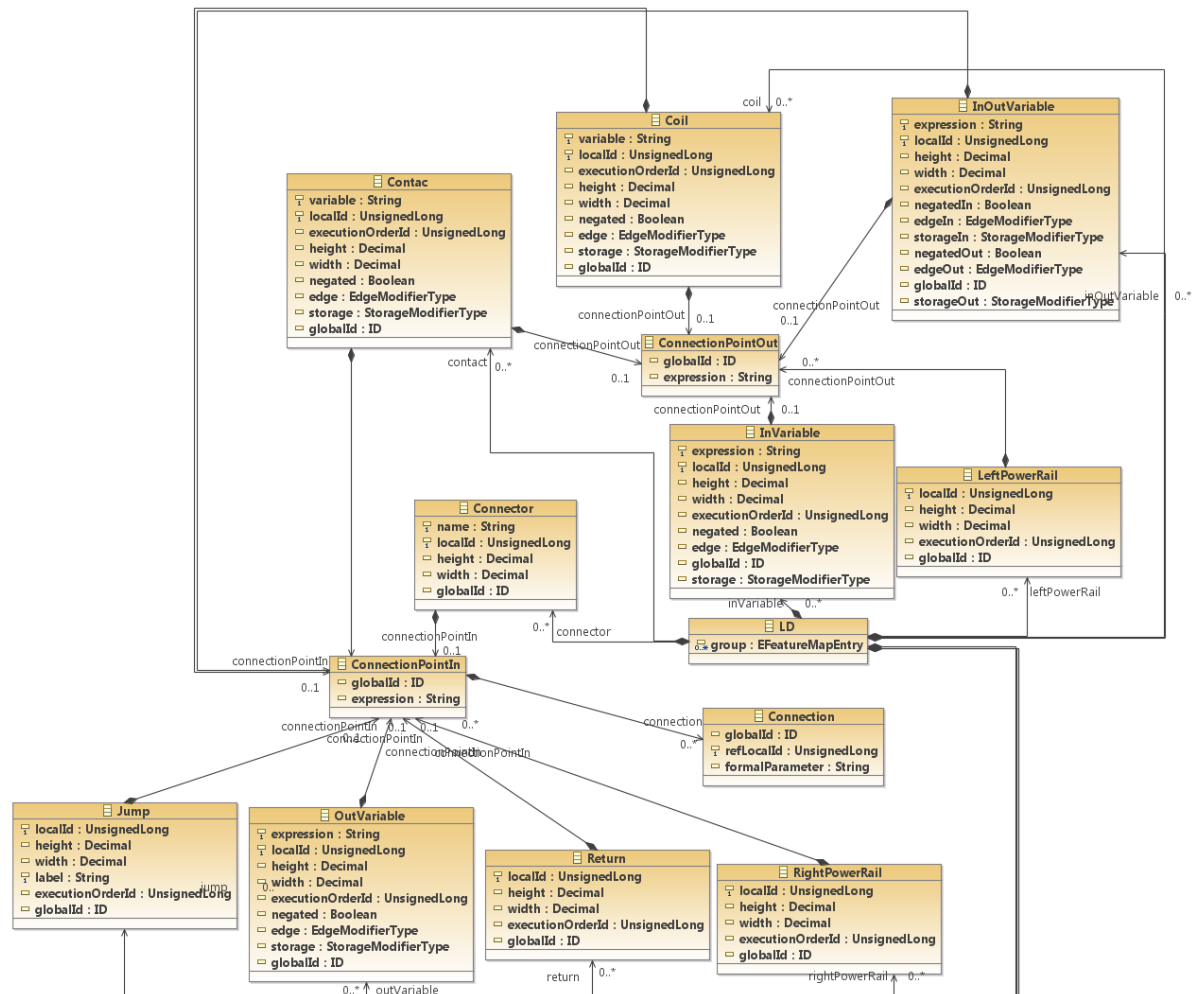


FIGURE 5.16 – Méta-modèle Beremiz

5.3.4.3 Transformation des programmes LD de Straton vers Beremiz

Cette transformation est relativement simple dans son principe. Les principales différences concernent les systèmes de coordonnées utilisés et la façon dont les éléments du diagramme LD sont reliés entre eux. Le tableau suivant (tableau 5.1) synthétise les principales différences entre les deux modèles.

La principale difficulté consiste à identifier et lier les éléments entre eux (lignes 4 et 5 du tableau précédent). Les règles de transformations sont les suivantes :

- La racine Straton (SourceLd) est transformée en racine Beremiz (LD).
- Les Contact et les Coil des modèles Straton sont transformés respectivement en Contact et Coil en Beremiz.

	Critère	Straton	Beremiz
1	Respect norme IEC	Non	Oui
2	Structuration du modèle	Hiéarchique	Linéaire
3	Séparateurs	Rungs	LeftPowerRail + Connection
4	Identification des éléments	Position de la balise XML	Attribut ID
5	Liaisons entre éléments	Position de la balise XML	Attribut refLocalID
6	Variables entrées/sorties	Attribut "IN"/"OUT"	Balise parent inVars/outVars

Tableau 5.1 – Tableau récapitulatif des différences entre Straton et Beremiz

- Les Box en Straton sont transformés en Bloc en Beremiz.
- Les Branch directement reliées à des contacts ou blocs fonctionnels sont transformées en Connexion.
- La gestion des refLocalId est réalisée par un algorithme qui permet, à partir des positions des éléments dans les balises XML de Straton, d’attribuer des identifiants localId aux éléments dans Beremiz. Ces identifiants sont stockés dans une liste qui est utilisée pour relier les éléments dans Beremiz.

Un extrait du modèle Beremiz généré est illustré ci-dessous.

```

<LD>
  <contact variable="Fdc02" localId="3" negated="false">
    <position x="100" y="200"/>
    <connectionPointIn> <connection refLocalId="0"/> </connectionPointIn>
  </contact>
  <coil variable="St02" localId="4" negated="false">
    <position x="700" y="200"/>
    <connectionPointIn> <connection refLocalId="3"/> </connectionPointIn>
    <connectionPointOut/>
  </coil>
  <leftPowerRail localId="0"/>
</LD>

```

5.3.4.4 Transformation Beremiz vers AT'

La transformation de modèles Beremiz en automates temporisés intermédiaires suit les règles suivantes (tableau 5.2) :

LD2Template. La racine du modèle Beremiz est transformée en Template contenant déjà les trois états non immédiats (*Lecture_Entrées*, *Traitements*, *Écriture_Sorties*), pour décrire l’exécution cyclique des APIs (Ligne 1, colonne 3 du tableau 5.2). Ce modèle générique est toujours produit quel que soit le programme LD utilisé. Un ensemble de localités et d’arcs sont ensuite générés à partir des éléments Ladder selon les règles ci-après.

Nom	Element Ladder	Equivalent Uppaal
Programme		
Variable d'entrée		
Variable de sortie		
Contacts en série		
Branches parallèles		
Blocs fonctionnels		
Enchaînement		

Tableau 5.2 – Règles de transformations des éléments Ladder en UPPAAL

Variable2Expression. L'arc reliant l'état *Lecture_Entrées* et l'état *Traitements* est complété par des assignations permettant de créer des copies de variables d'entrées. Ce sont ces copies qui sont utilisées dans le reste du calcul. Le but est de garder la même valeur de la variable durant tout un cycle. Les changements ne sont pris en compte que dans le cycle prochain. Les sorties sont aussi transformées en expressions logiques et positionnées sur l'arc reliant l'état *Écriture_Sorties* et l'état *Lecture_Entrée*.

BlocF2MessageEtatArc. Chaque bloc fonctionnel est converti en deux états instantanés et trois arcs (Ligne 2, colonne 3 du tableau 5.2). Pour chaque bloc fonctionnel, un message pour assurer la communication entre l'automate temporisé du bloc fonctionnel et l'automate principal est créé. Ce message est positionné sur l'arc qui relie les deux états instantanés (exemple (*TON!*)). Le deuxième arc relie le premier état instantané à l'élément qui le précède. Sur cet arc, sont positionnées les expressions logiques déduites à partir des entrées du bloc. Le troisième arc relie le deuxième état à l'élément qui le suit et porte les sorties du bloc, transformées en expressions logiques.

Bobine2Localité. Chaque bobine est transformée en localité instantanée.

Contact2Expression. Chaque contact est transformé en expression logique ET (&& dans UPPAAL). Le premier membre de cette expression correspond à la variable associée au contact courant. Le deuxième membre référence l'élément précédent :

- Si cet élément est un contact, celui-ci est lui-même transformé en expression ET. On peut de cette façon transformer des contacts mis en série en une expression de plusieurs ET logiques.
- Si cet élément est un bloc, on récupère la sortie de celui-ci.
- Les contacts liés au rail d'alimentation (qui n'ont donc pas d'élément précédent) sont transformés en expression simple ne contenant que la variable associée au contact.

Connexions2Expression. Chaque élément en parallèle est transformé en expression logique OU (|| en UPPAAL). Cette règle est conçue pour être appelée récursivement (L.34 sur la figure 5.17) puisque les expressions logiques sont considérées comme binaires dans ce modèle. Elle est appelée pour la première fois lorsque l'on trouve sur le modèle Beremiz, un élément dont les entrées sont multiples (le plus souvent cet élément est un *Coil* ou un bloc fonctionnel TON/TOF). Cette règle prend en argument une séquence de *Connection* (L.1).

Le premier membre de l'expression OU est créé comme suit :

- Si l'élément est un contact (L.6), on le référence directement (L.7) puisque les contacts sont directement transformés en expressions (cf. la règle *Contact2Expression*).
- Si l'élément est un bloc fonctionnel (L.9-14), on distingue deux cas :
 1. Le bloc est un bloc comparateur : on récupère alors le bloc en question (L.10-11), car ces blocs sont transformés en expressions logiques.
 2. Le bloc n'est pas un bloc comparateur : on récupère alors les variables de sortie (L.12-13). On ne récupère pas le bloc comme dans le cas précédent puisque les blocs autres que comparateurs ne créent pas d'expressions mais une localité UPPAAL.

Concernant le deuxième membre de l'expression OU, on considère deux cas selon la longueur de la liste de *Connection* en entrée :

- Si la liste est de longueur égale à 2 (i.e. il existe deux branches (L.19)), le deuxième membre est traité comme le premier membre (L.20-31).
- Si la liste est de longueur strictement supérieure à 2 (L.32), le second membre est lui-même une expression logique OU. On définit ainsi récursivement (L.33) des expressions logiques OU selon le nombre de branches.

```

1 rule OExpressions(i: Sequence(iec!Connection)) {
2 using { previous: OclAny = i -> first().referred();}
3 to
4 o: guppaal!LogicalExpression (
5   operator <- #OR,
6   firstExpr <- if previous.oclIsKindOf(iec!ContactType) then
7     previous
8   else
9     if previous.oclIsKindOf(iec!BlockType) then
10      if thisModule.comp_operator -> includes(previous.typeName) then
11        previous
12      else
13        previous.outputVariables
14      endif
15    else
16      OclUndefined
17    endif
18  endif,
19  secondExpr <- if i.size() = 2 then
20    let r: OclAny = i -> at(2).referred() in
21    if r.oclIsKindOf(iec!ContactType) then r
22  else
23    if r.oclIsKindOf(iec!BlockType) then
24      if thisModule.comp_operator -> includes(r.typeName) then
25        r
26      else
27        r.outputVariables
28      endif
29    else
30      OclUndefined.debug('NO 2ND EXPR FOUND')
31    endif
32  endif
33  else
34    thisModule.OExpressions(i -> excluding(i -> first()))
35  endif )
36 do { o; }

```

FIGURE 5.17 – Règle ATL de transformation des éléments en parallèles

5.3.5 Opération de transformation de AT' en AT

Cette section a pour but de détailler les étapes regroupées dans l'opération de transformation des automates temporisés intermédiaires en modèles UPPAAL exploitables (figure 5.18).

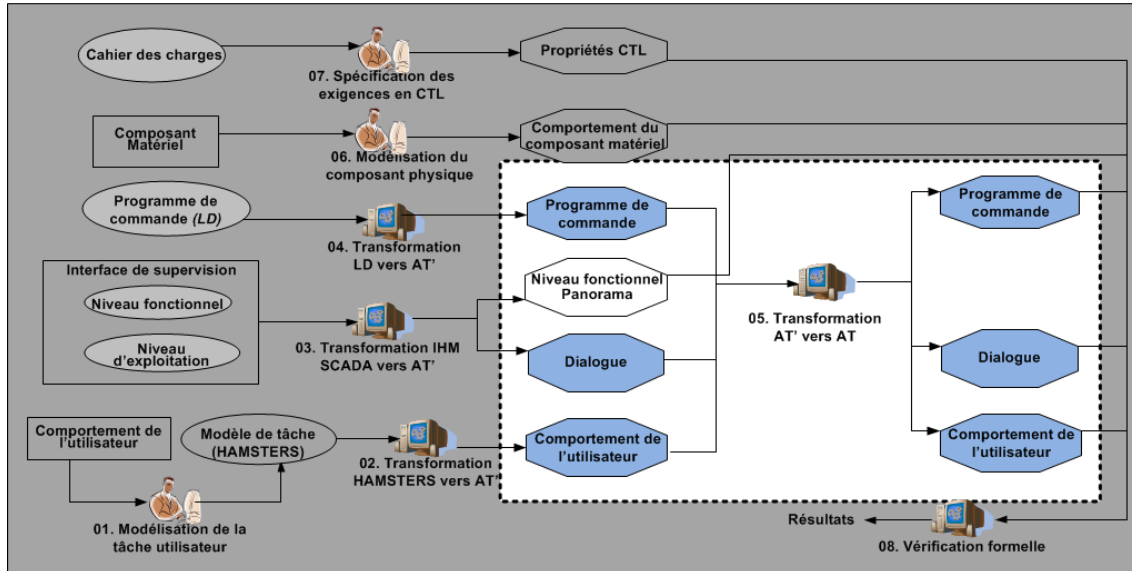


FIGURE 5.18 – Transformation de AT' en AT

Cette étape fait suite à la contrainte de la présence du texte brut dans les modèles UPPAAL. Cette transformation est relativement simple : comme les modèles d'entrée et de sortie représentent la même chose, chaque élément du modèle d'entrée a son unique équivalent dans le modèle de sortie.

Cependant, une transformation ATL ne permet pas la génération directe de texte XML (i.e texte entre balises ouvrantes et fermantes). Le modèle obtenu suite à la transformation précédente n'est donc toujours pas exploitable. La solution consiste donc à transformer les modèles AT' obtenus en modèles XML conformes au méta-modèle XML. Pour illustrer ces propos, voici un extrait du modèle AT' généré par ATL.

```
1 | <label value="cycle=0" kind="invariant"/>
```

Alors que l'on veut obtenir un modèle ayant l'allure suivante :

```
1 | <label kind="invariant">cycle=0</label>
```

5.3.5.1 Méta-modèle XML

Le méta-modèle XML est très simple (figure 5.19). Il est constitué essentiellement de nœuds (Node). Ces derniers sont soit des éléments (Element), des attributs (Attribut) ou du texte (Text). Les éléments se décomposent en d'autres éléments.

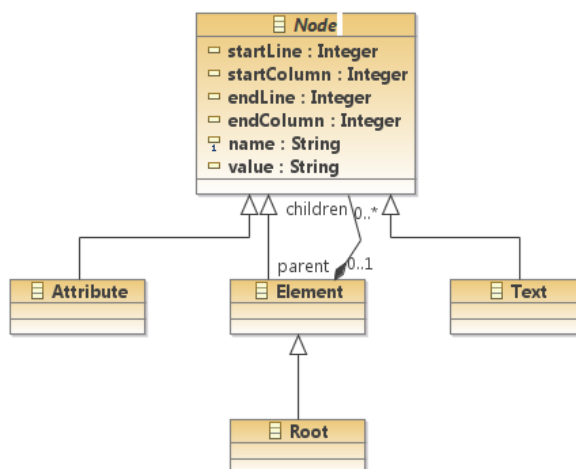


FIGURE 5.19 – Méta-modèle XML

5.3.5.2 Transformation des modèles AT' en XML

Chaque élément du modèle AT' est converti en élément XML générique qui lui correspond (Root pour l'élément racine, Element pour les autres éléments du modèle, Attribute pour les attributs de chaque élément, Text pour les attributs qui doivent être changés en texte). L'intérêt de cette transformation est qu'elle permet la distinction entre les attributs et le texte, ce qui n'est pas le cas avec ATL qui les considère au même titre.

La figure 5.20 illustre un extrait du modèle XML obtenu après la transformation des AT' vers XML. On remarque la création du nœud Text (L.9) ayant le texte "cycle=0" dans l'attribut value.

```

1 | <Root name="nta">
2 |   ...
3 |   <children xsi:type="Element" name="location">
4 |     <children xsi:type="Attribute" name="id" value="write_output"/>
5 |     <children xsi:type="Attribute" name="x" value="340"/>
6 |     <children xsi:type="Attribute" name="y" value="380"/>
7 |     <children xsi:type="Element" name="label">
8 |       <children xsi:type="Attribute" name="kind" value="invariant"/>
9 |       <children xsi:type="Text" value="cycle=0"/>
10 |     </children>
11 |   </children>
12 |   ...
13 | </Root>
  
```

FIGURE 5.20 – Extrait d'un modèle XML

5.3.5.3 Transformation modèle XML vers modèle UPPAAL définitif

Cette transformation est de type Modèle vers Texte (Model To Text) et elle est composée des règles suivantes :

- Chaque nœud de type `Element` est remplacé par une balise portant son nom (toujours sous forme de texte `String`).
- Chaque nœud de type `Attribute` de nom `X` et de valeur `Y` est remplacé par le `String` `'X="Y"'`.
- Chaque nœud de type `Text` contenu dans un élément parent `X` (`X` est donc forcément de type `Element`) et ayant une valeur `Y` est remplacé par `Y` ajouté entre les balises ouvrantes et fermantes de `X`. Le résultat final est donc `'<X>Y</X>'`
- Chaque nœud enfant d'un autre nœud est traité par récursivité en suivant les 3 règles précédentes.

On obtient finalement un fichier XML qui est bien lisible par UPPAAL.

5.3.6 Opération de modélisation du composant physique en AT

L'opération de modélisation du comportement du composant physique est réalisée manuellement par un analyste (figure 5.21). Le comportement du composant physique est également modélisé par un ensemble d'automates temporisés qui modélisent le comportement des actionneurs, des capteurs et du composant physique lui-même. La communication entre l'automate temporisé du composant physique et l'automate du programme de commande, est assurée grâce aux variables booléennes partagées.

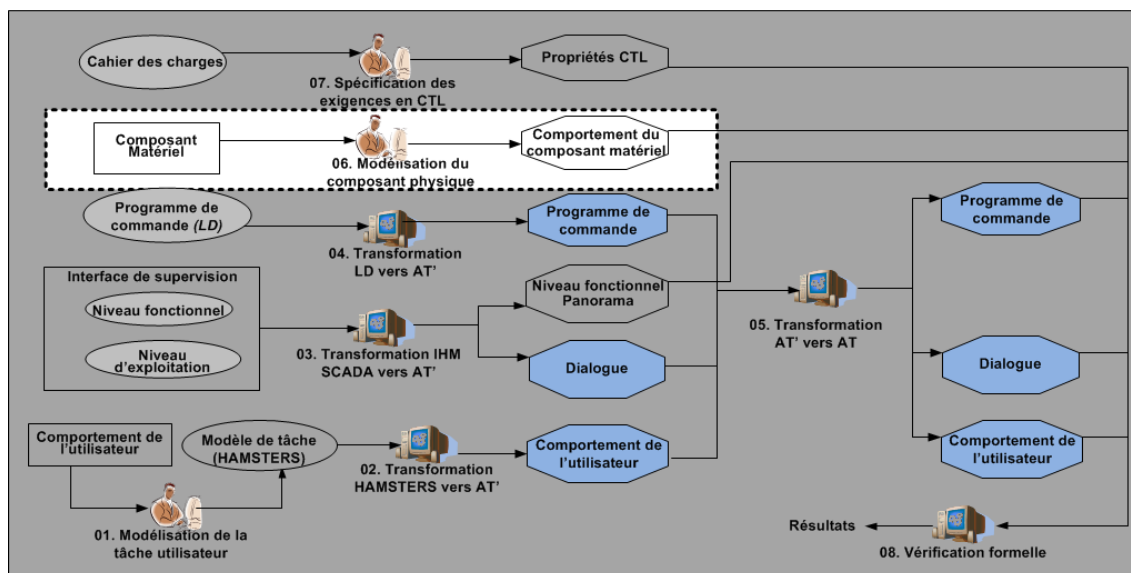


FIGURE 5.21 – Opération de modélisation du composant physique en AT

5.3.7 Opération de spécification des exigences en CTL

L'opération de spécification des exigences consiste à écrire en CTL les propriétés que le composant standard doit satisfaire (figure 5.22). Cette opération manuelle est réalisée par l'analyste ayant des connaissances en notation CTL. Souvent, ce sont des propriétés de sûreté, de vivacité et de non blocage qui sont vérifiées.

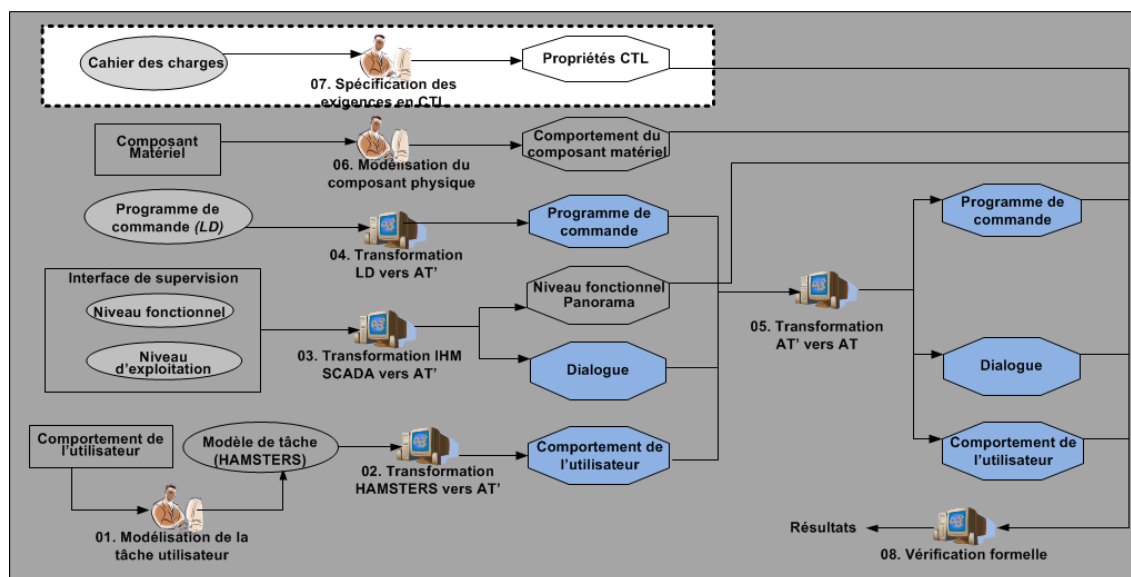


FIGURE 5.22 – Opération de spécification des exigences en CTL

5.3.8 Opération de vérification formelle

Les automates temporisés issus des opérations précédentes sont combinés avec les propriétés CTL, de la septième opération, pour conduire la vérification formelle (figure 5.23) du composant standard. Le *Model-Checker* UPPAAL est utilisé pour vérifier la satisfaction des propriétés CTL sur les automates temporisés.

L'interprétation des résultats de la vérification est à la charge de l'analyste. Ce dernier a pour rôle de déterminer la cause de l'erreur (si un contre-exemple est retourné) et de proposer des corrections. Les solutions portées aux programmes de commande et des IHM de supervision des composants standards sont alors communiquées aux ingénieurs pour qu'elles soient implémentées. Ensuite, notre approche est appliquée afin de générer des automates temporisés correspondant aux nouveaux codes. Une deuxième phase de vérification est alors exécutée. Ce processus itératif est appliqué jusqu'à ce que toutes les exigences soient satisfaites.

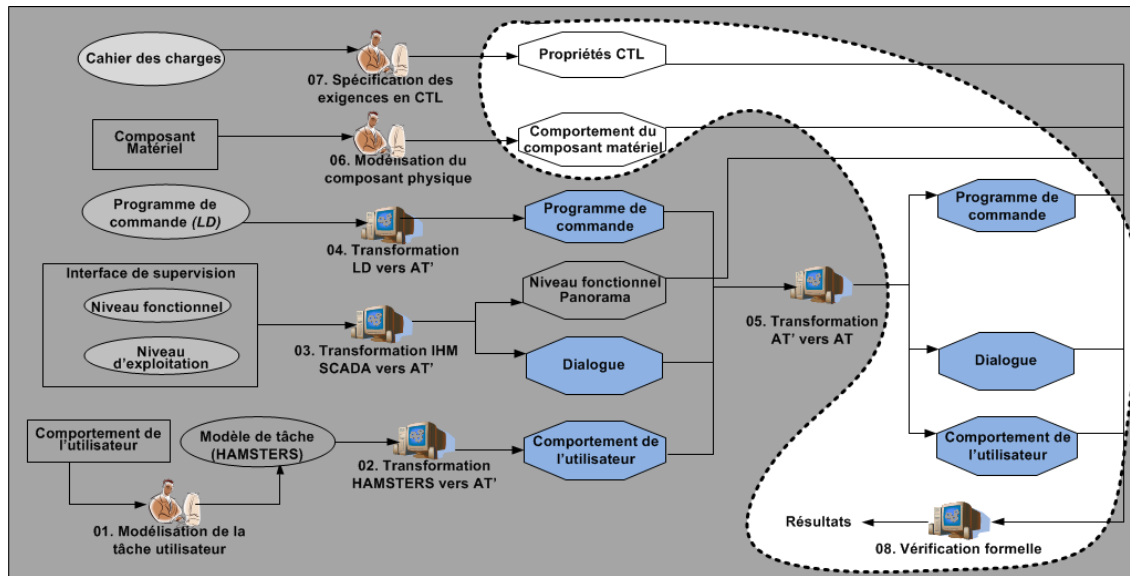


FIGURE 5.23 – Vérification formelle par UPPAAL

5.3.9 Bilan

Nous avons présenté et détaillé dans les sections précédentes, un flot de vérification des composants standards composé de huit opérations : cinq opérations automatiques et trois opérations manuelles. Les opérations automatiques sont implémentées par des transformations de modèles et permettent essentiellement de transformer les différentes vues de conception en automates temporisés. Les opérations de construction de modèles de tâches, de modélisation du système physique et de la spécification des exigences CTL, sont manuelles et nécessitent donc l'intervention du concepteur ou de l'analyste.

En complément de ces travaux, nous avons proposé un flot de vérification des diagrammes P&ID que nous abordons dans la section suivante.

5.4 Flot de vérification formelle des diagrammes P&ID

La transformation des diagrammes P&ID en Alloy est implémentée sous la forme d'un flot composé de plusieurs opérations (figure 5.24).

La première opération est l'opération de formalisation de la norme ANSI/ISA-5.1 en Alloy sous forme de style architectural (cf. section 4.4.3 du chapitre 4). Le but de cette opération est de poser un cadre formel pour la norme ANSI/ISA-5.1 et, ainsi, permettre la vérification de tout diagramme P&ID écrit dans cette norme. La deuxième opération est l'opération de construction. Lors de cette opération manuelle, le concepteur métier construit un diagramme P&ID, sous Visio, à partir des recommandations du cahier des charges et d'une bibliothèque de composants standards. Cette opération s'inscrit dans une approche ascendante. L'opération

manuelle "Spécification des exigences" est réalisée par l'analyste et a pour objectif d'écrire les exigences mentionnées dans le cahier des charges, sous forme d'assertions (Assert) en Alloy. Dans la quatrième opération, le diagramme P&ID est transformé en modèle générique "Le synoptique" (défini dans [Bignon 2012]). Ce dernier est une représentation abstraite de l'architecture physique du système sous la forme d'un ensemble de symboles (composants) et de connexions (connecteurs). Le modèle synoptique est utilisé comme modèle source, dans la cinquième opération ("Transformation P&ID à Alloy"), pour générer automatiquement un modèle Alloy cible. Ce dernier, fusionné avec le style architectural défini dans la première opération, ainsi que les exigences résultantes de l'opération de "Spécification des exigences", sont utilisés comme entrées lors de l'opération de "Vérification" (figure 5.24) réalisée avec l'analyseur Alloy. Si un contre-exemple est retourné, ce dernier est utilisé dans la dernière opération "Visualisation des erreurs" pour afficher le contre-exemple sous une forme graphique sur les diagrammes P&ID. Le concepteur peut effectuer des itérations de corrections (opération "Corrections") et de vérifications jusqu'à ce que le diagramme P&ID devienne correct. Ces opérations sont détaillées dans les sections suivantes.

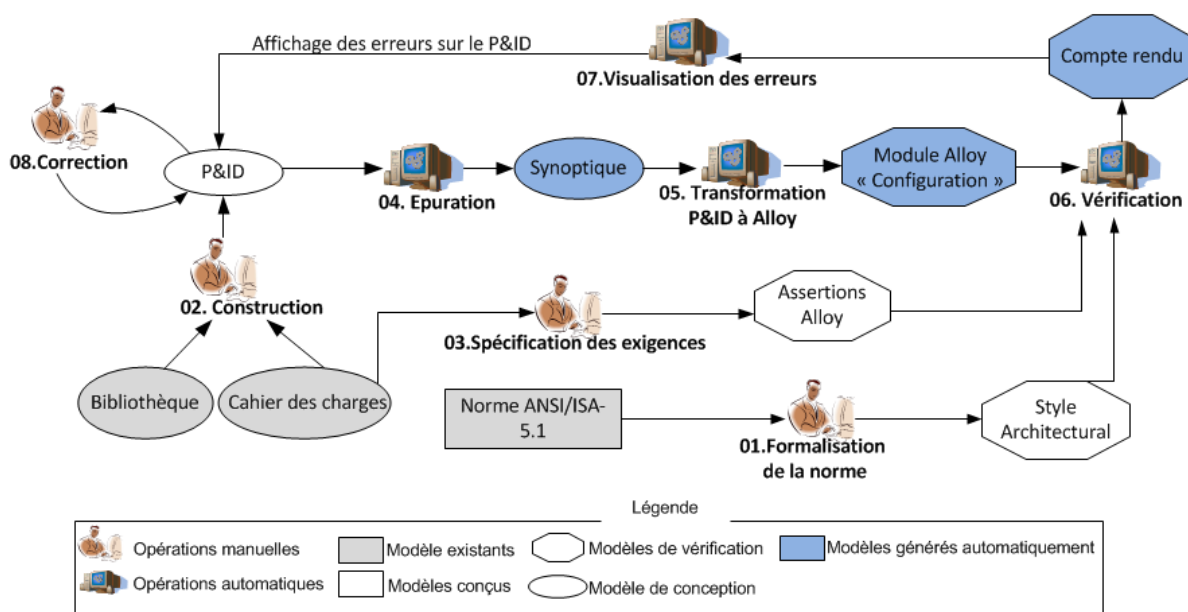


FIGURE 5.24 – Flot de vérification des diagrammes P&ID

5.4.1 Opération de formalisation de la norme ANSI/ISA-5.1

La figure 5.25 illustre l'opération manuelle de formalisation de la norme ANSI/ISA-5.1. Nous avons présenté dans le chapitre 4 une définition formelle de la norme ANSI/ISA-5.1. La définition de cette norme a été proposée sous la forme d'un style architectural. Ce dernier offre un ensemble d'entités et de règles pour la construction des architectures conformes au style.

Pour la définition d'un style architectural, il faut définir les composants, les connecteurs, les configurations ainsi que les règles de construction d'architecture [Kim et Garlan 2010]. Après une étude approfondie de la norme ANSI/ISA-5.1, nous avons défini une description formelle en Alloy pour le vocabulaire (les composants, les connecteurs et les configurations) et les règles (invariants) du style (cf. section 4.4.3 du chapitre 4).

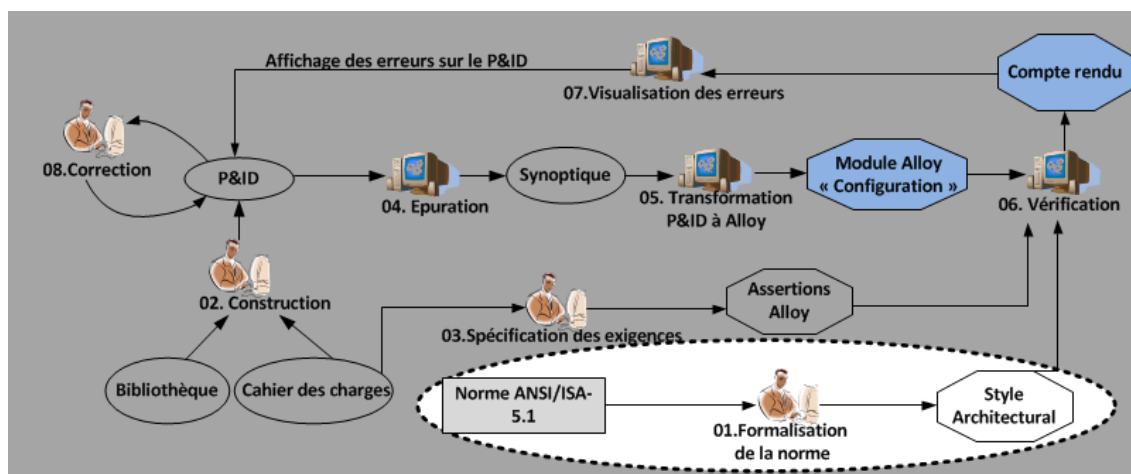


FIGURE 5.25 – Opération de formalisation de la norme ANSI/ISA

5.4.2 Opération de construction

L'opération de construction a été définie dans les travaux de Bignon [2012]. L'objectif de cette opération est l'obtention d'un diagramme P&ID à partir d'un cahier des charges et d'une agrégation (démarche ascendante) de composants standards (figure 5.26).

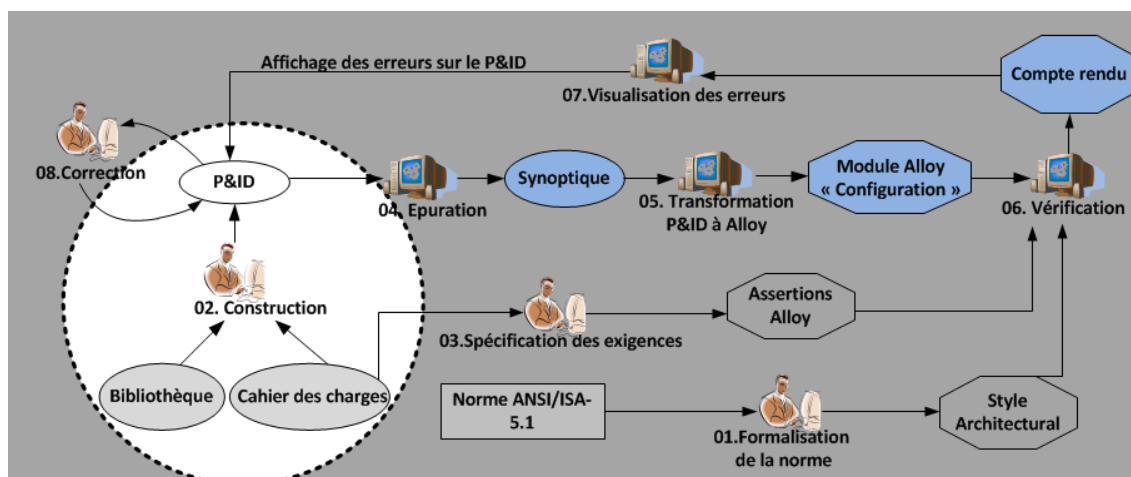


FIGURE 5.26 – Opération de construction

Le concepteur s'appuie sur le cahier des charges et la bibliothèque de composants standards pour construire un diagramme P&ID conforme à la norme ANSI/ISA-5.1. Le P&ID est un diagramme structuro-fonctionnel qui décrit l'architecture du procédé industriel d'un point de vue physique [Bignon 2012]. Ce qui signifie que le P&ID doit intégrer des exigences venant des normes, des exigences fonctionnelles et non-fonctionnelles.

La bibliothèque de composants standards a été implémentée dans l'outil Microsoft Visio [Bignon 2012]. Par conséquent, le diagramme P&ID est aussi saisi dans cet outil. Nous illustrons dans la figure 5.27 un exemple du P&ID saisi dans Visio (figure 5.27) à partir de la bibliothèque de composants standards (à gauche sur la figure 5.27).

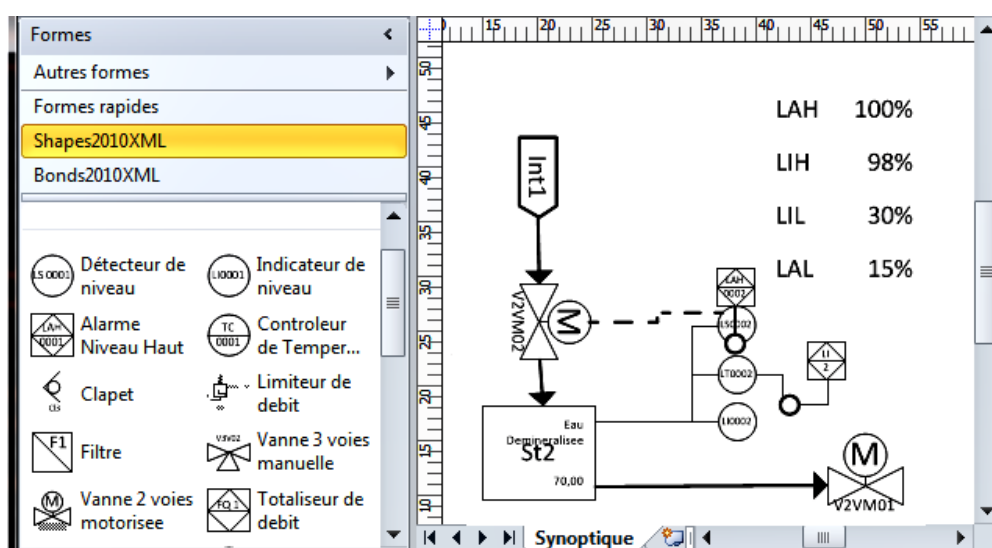


FIGURE 5.27 – Exemple d'un P&ID saisi sous Visio à partir d'une bibliothèque

Le P&ID de la figure 5.27 comprend trois composants process : une source d'eau (Int1), une vanne deux voies motorisées (V2VM02) et une soute (St2). Cette dernière est instrumentée par des capteurs et des indicateurs de niveau. Sur ce P&ID, plusieurs exigences ont été appliquées. Une exigence fonctionnelle provenant du cahier des charges serait par exemple de "permettre le stockage de l'eau provenant de la source (Int1) dans la soute (St2)". Pour satisfaire cette exigence, le concepteur a mis en séquence une source d'eau et une soute de stockage. Cette exigence est complétée par une autre exigence non-fonctionnelle portant sur la maintenance des composants. Cette exigence stipule que des vannes doivent être raccordées en amont et en aval de chaque soute. Pour assurer cette exigence, le concepteur a ajouté une vanne motorisée juste avant la soute. Des règles de style sont aussi appliquées dans cet exemple de P&ID. On voit bien que les composants process, comme la soute et la vanne, sont raccordés par des connecteurs type process (lien direct). Par contre, la connexion entre la soute et le capteur de niveau très haut est de type électrique (lien pointillé). Ces règles proviennent de la norme ANSI/ISA-5.1 et, par conséquent, elles figurent parmi les contraintes du style (invariant 1,

Listing 4 du chapitre 4).

5.4.3 Opération de spécification des exigences

Dans cette section, nous présentons et nous illustrons dans la figure 5.28, l'opération de "Spécification des exigences".

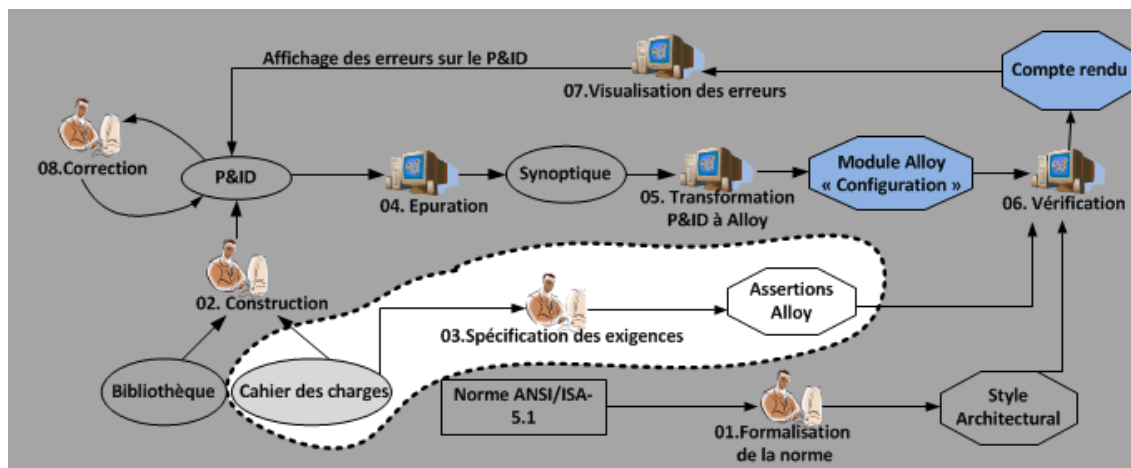


FIGURE 5.28 – Opération de spécification des exigences

Cette opération manuelle réalisée par l'analyste permet de spécifier en Alloy, les exigences que le diagramme P&ID doit satisfaire. Ces exigences, essentiellement des exigences fonctionnelles et non-fonctionnelles, sont extraites du cahier des charges. Elles sont ensuite écrites en Alloy sous forme d'assertions. Contrairement, aux règles du style architectural que tous les diagrammes P&ID doivent satisfaire, ces assertions sont spécifiques au système étudié dont l'architecture est décrite par un P&ID. Le Listing 1 présente un exemple d'une exigence non-fonctionnelle décrit par des prédicats en Alloy.

Listing 1

```

pred ComposantIsolé {
  let vannes = V2VM+V3VM | all s:TRCH+St+HP |
  some (ComposantIsoléAmont[s] & vannes) &&
  some(ComposantIsoléAval[s] & vannes)}

fun ComposantIsoléAmont[s:Composant]:Composant {
{c: V2VM+V3VM| some cl:C1 | c in s.ComponentConnecté ||
((cl in s.ComponentConnecté) && (c in cl.ComponentConnecté))}}

fun ComposantIsoléAval[s: Component]:Component {
{c: V2VM+V3VM | some cl:C1 | s in c.ComponentConnecté ||
((s in cl.ComponentConnecté) && (cl in c.ComponentConnecté))}}

```

Le predicat `ComposantIsolé` dans le Listing 1 permet de vérifier que les soutes (St) et les pompes (HP) sont toujours précédées (`ComposantIsoléAval`) et suivies (`ComposantIsoléAval`) par des vannes (V2VM ou V3VM). Cette exigence non-fonctionnelle permet de s'assurer que les gros composants soient bien isolés par des vannes à des fins de maintenance.

5.4.4 Opération d'épuration

L'objectif général de l'opération d'épuration est de n'extraire que les informations utiles du document Visio correspondant au P&ID. Cette opération est implémentée par une transformation de modèle. Elle permet de transformer automatiquement un modèle P&ID saisi dans Visio en un modèle "Synoptique". Ce dernier est une description abstraite du P&ID sous forme de composants et de connexions, indépendant de toutes plateformes (figure 5.29).

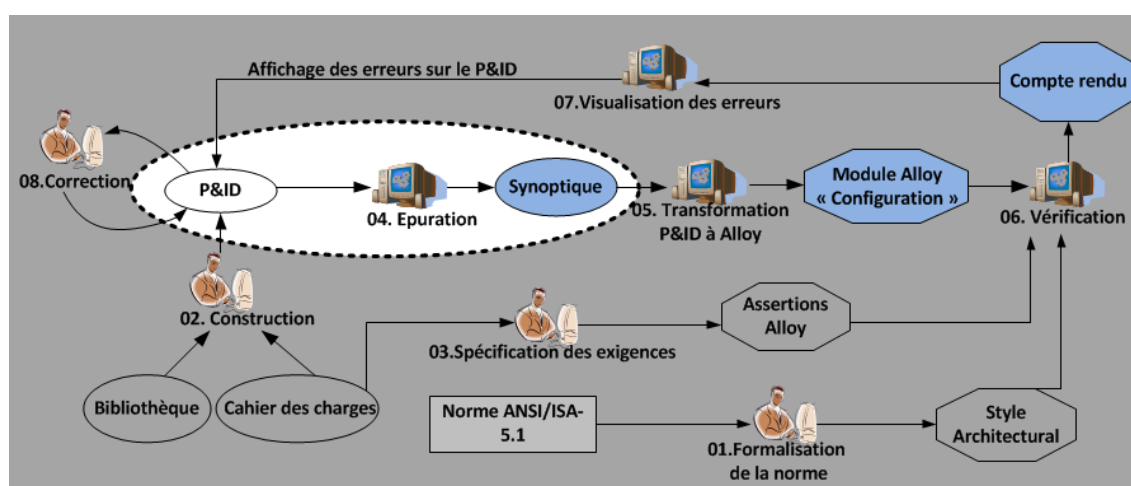


FIGURE 5.29 – Opération d'épuration

La figure 5.30 illustre un extrait du modèle synoptique généré à partir du modèle Visio correspondant au P&ID de la figure 5.27. Ce modèle (XML) est composé de symboles et de connexions. Les symboles et les connexions sont générés respectivement à partir des composants et des liens présents dans le P&ID. Le symbole avec le nom "V2VM" (Nom="V2VM" sur la figure) et l'identifiant (ID="V2VM02") correspond à la vanne motorisée V2M02 sur le P&ID. La connexion avec l'identifiant "P-13.349" (ID="P-13.349" sur la figure 5.30) a été générée à partir du lien qui relie la vanne V2VM02 à la soute St2. Cette connexion de type process comporte deux extrémités (extremities sur la figure 5.30). La balise (ID="V2VM02" ext="Ext1") signifie que la première extrémité de cette connexion est reliée à la vanne V2VM02. La deuxième extrémité (ext="Ext2") est reliée à la soute St2.

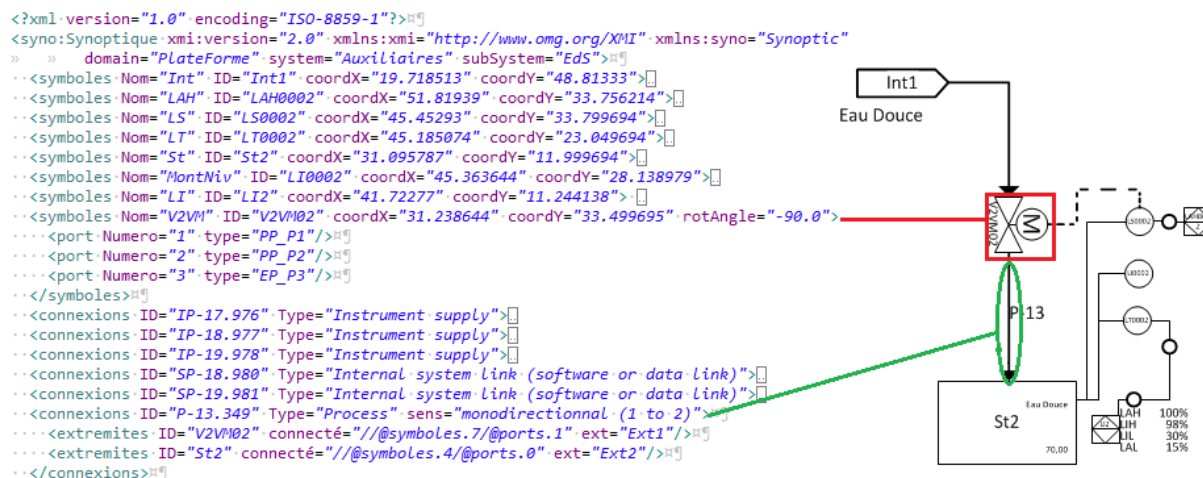


FIGURE 5.30 – Extrait du modèle synoptique correspondant au P&ID de la figure 5.27

5.4.5 Opération de transformation de P&ID en Alloy

Nous présentons, ici, l’opération automatique de la transformations des diagrammes P&ID en Alloy (figure 5.31).

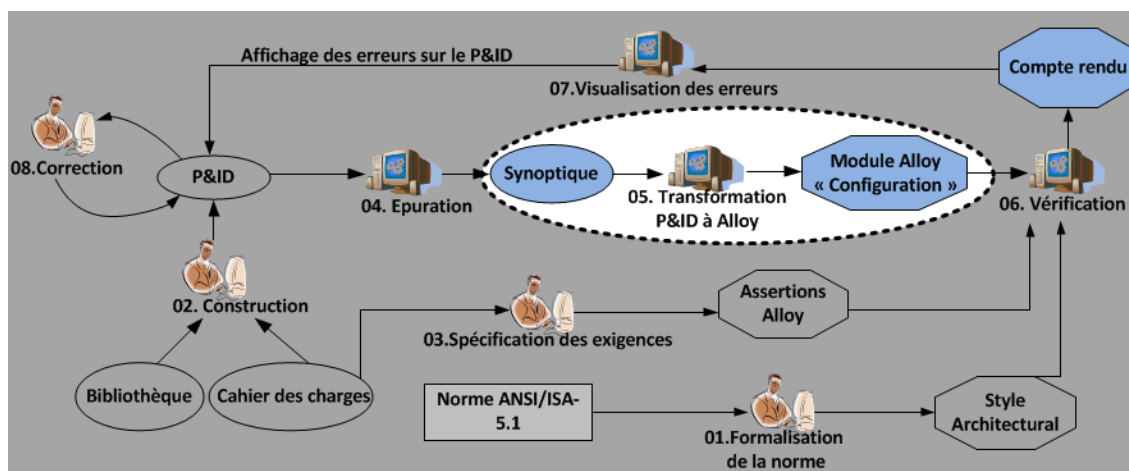


FIGURE 5.31 – Opération de transformation des diagrammes P&ID en Alloy

Cette opération est implémentée par une transformation de modèles. Nous présentons, ci-après, les méta-modèles, les différentes règles de transformation et le modèle généré.

5.4.5.1 Les méta-modèles

L’opération de transformation des diagrammes P&ID en Alloy prend en entrée un modèle synoptique conforme au méta-modèle Synoptique et génère un module Alloy conforme au

méta-modèle Alloy.

5.4.5.1.1 Le méta-modèle des diagrammes P&ID

La figure 5.32 illustre le méta-modèle Synoptique, tiré de la thèse de Bignon [2012]. Ce dernier est un ensemble de symboles et de connexions. Chaque symbole représente un composant de la norme ANSI/ISA-5.1. Il est caractérisé par un nom et un identifiant (ID). Le nom correspond au type du composant. Le ID est unique et permet d'identifier le composant par rapport aux autres composants.

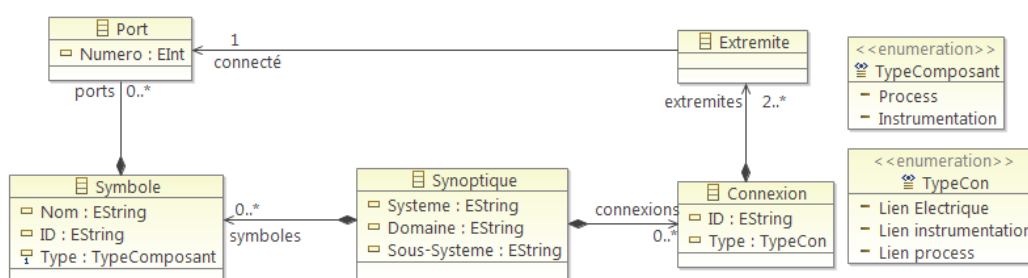


FIGURE 5.32 – Le méta-modèle des diagrammes P&ID [Bignon 2012]

Le symbole contient des ports qui représentent les points d'interaction du symbole avec son environnement. Les connexions offertes par la norme ANSI/ISA-5.1 sont de plusieurs types (e.g : les liens process, les liens d'instrumentation, les liens électriques ou les liens logiciels) et elles assurent l'interaction entre les composants. Chaque connexion comprend au maximum deux extrémités. Chaque extrémité est connectée à un port.

5.4.5.1.2 Le méta-modèle d'Alloy

Le méta-modèle d'Alloy [Garis et al. 2012] est organisé en modules (figure 5.33). Un module Alloy est composé d'une entête (Header sur la figure 5.33), d'un ensemble de `import` et de paragraphes (Paragraph). Ces derniers regroupent des signatures (Sig), des faits (Fact), des prédicats (Pred), des fonctions (Fun), des assertions (Assert) et des commandes (Check et run).

5.4.5.2 Les transformations de modèles

Pour la vérification des diagrammes P&ID, nous avons utilisé deux modules Alloy. Le premier module, nommé `Style`, spécifie le style architectural initialement défini. Le deuxième, nommé `Configuration_Générée` est généré automatiquement à partir d'un diagramme P&ID.

À partir d'un modèle synoptique, nous générons un module `Configuration_Générée` qui importe le module `Style`. Le module `Configuration_Générée` comprend aussi des para-

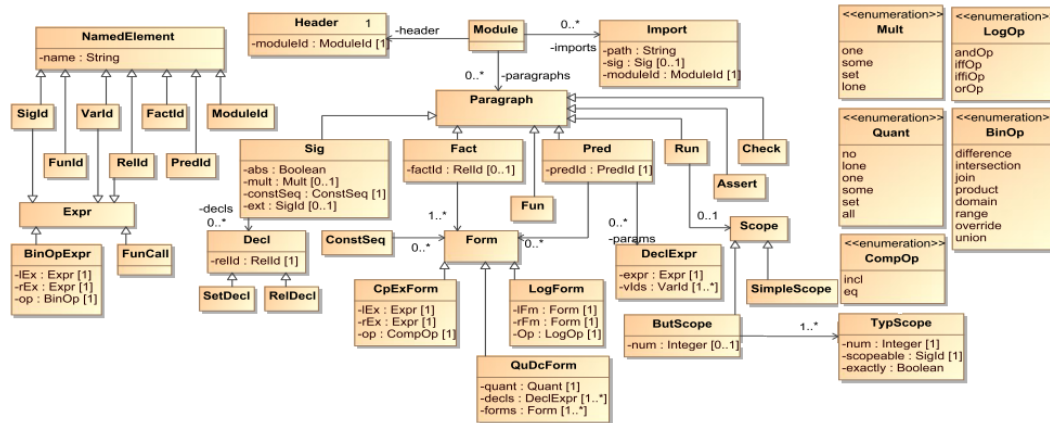


FIGURE 5.33 – Le métamodèle Alloy [Garis et al. 2012]

graphes de type sig, facts, run et check. Les règles de transformation utilisées pour transformer les diagrammes P&ID en Alloy sont décrites ci-après.

Synoptique2Module. Le nœud Synoptique est transformé en module Alloy nommé `Configuration_Générée`. Ce module importe le module "Style" (`import Style as style` dans le Listing 2) qui correspond au style architectural défini dans la section 4.4.3 du chapitre 4. Le module contient également une signature nommée `EdS` (Listing 2). Cette signature hérite de la signature `Configuration` (Listing 3 dans le chapitre 4). Les attributs composants correspondent à l'union de tous les symboles transformés en signatures (`Int1+V2VM02+St2+...` sur le Listing 2). L'attribut `connecteurs` contient l'union de tous les connecteurs générés à partir des connexions.

Symbole2Signature. Chaque symbole correspond à un composant qui hérite d'une signature déjà définie dans le style. L'attribut `Nom` du symbole est utilisé pour définir la signature mère de la signature générée. L'attribut `ID` est utilisé pour nommer la signature générée. Par exemple, le symbole `Nom="V2VM" ID="V2VM02"` est transformé en `one sig V2VM02 extends V2VM`. Les ports prédéfinis dans chaque composant (exemple `p1`, `p2`, `p3` dans `V2VM`) correspondent aux signatures générées à partir de chaque port. Dans le Listing 2, nous illustrons une instance du composant `V2VM`. Cette instance `V2VM02` hérite de la signature `V2VM`. La signature `V2VM02_PORT_1`, générées à partir d'un port, correspond au port `p1`.

Port2Signature. Chaque port dans le modèle synoptique est transformé en une signature qui hérite du type de port correspondant. Dans le Listing 2, nous pouvons voir que la signature générée à partir du port `V2VM02_PORT_1` de type `process`, hérite de la signature `PP`. La signature `V2VM02_PORT_3` de type `électrique` hérite de la signature `EP`. Les ports non connectés ne sont pas générés.

Connexion2Signature. Chaque connexion est transformée en une signature qui hérite de la signature qui correspond à son type. Par exemple, les connexions `Process` dans la figure 5.32 sont transformées en signatures qui héritent la signature `LP` (Listing 2 dans le chapitre4). L'attribut `roles` de la signature `Connecteur` correspond à l'union de toutes ses extrémités transformées en `Role`. Par exemple, dans le Listing 2, la connexion `process LP_1` est composée des `roles` : `LP_1_1` et `LP_1_2`.

Extrémité2Signature. Chaque extrémité est transformée en un `role` (`extends Role`). Elle est composée de deux attributs : `connecteur` et `connecté` (Listing 2). L'attribut `connecteur` définit le connecteur qui contient l'extrémité. La relation `connecté` détermine le port auquel le rôle est connecté. Dans le Listing 2, par exemple, le rôle `LP_1_1` est connecté au port `p2` du composant `V2VM02` (`connecté = V2VM02.p2` sur le Listing 2).

Listing 2

```

module Configuration_Générée
open Style as style

-Connecteurs
one sig LP_1 extends PL{}{
    roles= LP_1_1+LP_1_2}
one sig LP_2 extends SL{}{
    roles =LP_2_1+LP_2_2}

-Ports
one sig V2VM02_PORT_1 extends PP{}{
    composant = V2VM02}
one sig V2VM02_PORT_2 extends PP{}{
    composant = V2VM02}
one sig V2VM02_PORT_3 extends EP{}{
    composant = V2VM02}

-Configuration
one sig EdS extends Configuration{}{
    composants = Int1+V2VM02+St2+..
    connecteurs = LP_1+..}

-Composants
one sig Int1 extends Interface{}{
    p1 = Int1_PORT_1}
one sig V2VM02 extends V2VM{}{
    p1 = V2VM02_PORT_1
    p2 = V2VM02_PORT_2
    p3 = V2VM02_PORT_3}

-Roles
one sig LP_1_1 extends Role{}{
    connecteur = LP_1
    connecté = V2VM02.p2}
one sig LP_1_2 extends Role{}{
    connecteur = LP_1
    connecté = St2.p1}

```

5.4.6 Opération de vérification

Le module Alloy résultant de l'opération de transformation du P&ID en Alloy, ainsi que les assertions résultantes de l'opération de spécification des exigences, sont utilisés comme entrée pour l'analyseur Alloy. La vérification est réalisée par la commande `Check`. La vérification a été réalisée sur une machine 2.7 Ghz i5 contenant 8 GO de mémoire. Nous avons utilisé la version 4.2_20150222 de l'analyseur Alloy avec le solveur *SAT4J*.

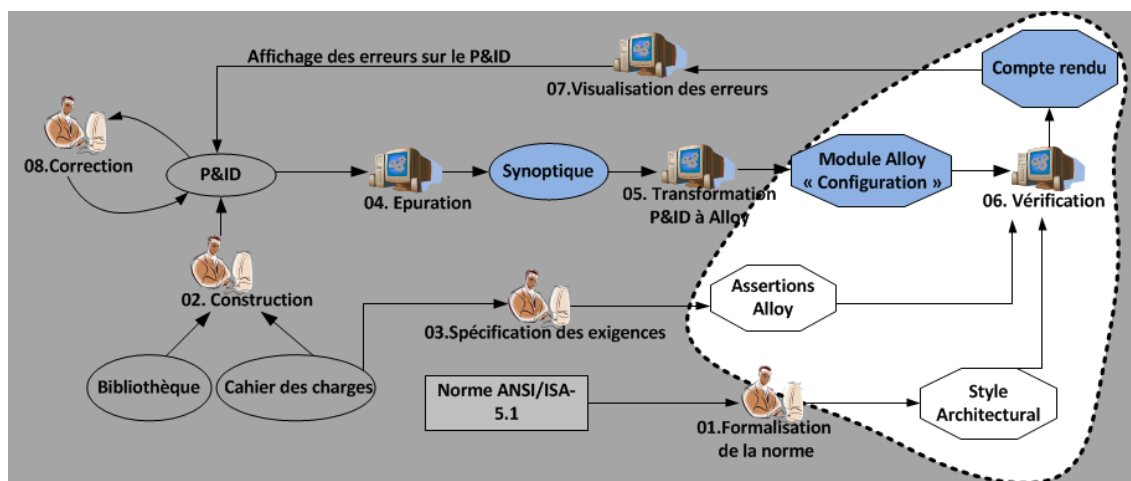


FIGURE 5.34 – Opération de vérification formelle

5.4.7 Opération de visualisation des erreurs

Dans cette section, nous présentons et détaillons l'opération de visualisation des erreurs (figure 5.35).

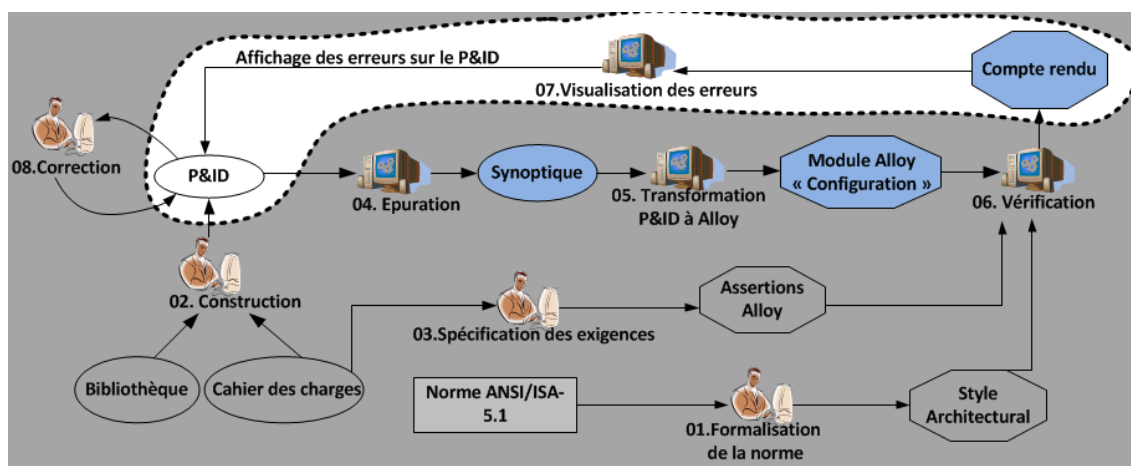


FIGURE 5.35 – Opération de visualisation des erreurs sur le P&ID

L'opération de visualisation, comme définie dans le chapitre 4, implémente toutes les fonctionnalités permettant à un concepteur de vérifier des propriétés sur un diagramme P&ID saisi dans Visio et d'afficher, dans le cas de violation d'une propriété, le contre-exemple retourné par l'analyseur Alloy sous forme graphique (sur le P&ID). La visualisation des contre-exemples est réalisée par trois étapes (figure 5.36) : la transformation des données, la transformation de visualisation et enfin la transformation du visuel graphique.

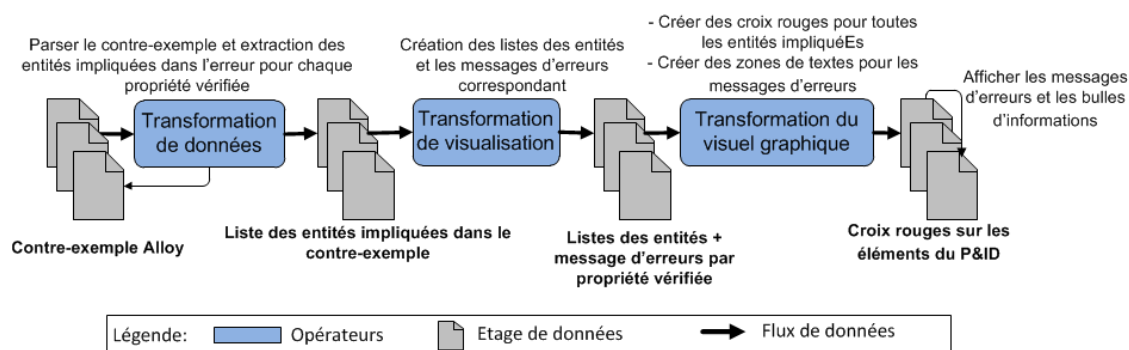


FIGURE 5.36 – Flot de visualisation des contre-exemples Alloy sur les diagrammes P&ID

5.4.7.1 Transformation de données

Le module de transformation est développé avec le langage Java et implémenté sur l'IDE¹⁰ Eclipse. Il est constitué globalement d'un ensemble de classes qui importent dans leur code la librairie "Alloy.jar" (le code de l'analyseur Alloy). Nous avons choisi le langage java car l'analyseur Alloy est développé en Java. Il était plus facile pour nous de réutiliser les méthodes de son code source.

Dans son ensemble, notre module contient la définition des objets manipulés mais aussi les méthodes nécessaires à la compilation d'un modèle Alloy et à l'évaluation de ses propriétés. En effet, le fonctionnement peut être assimilé à un grand module qui reçoit en entrée le nom d'une propriété, réalise un couplage pour déterminer la formule Alloy de la propriété reçue, fait appel aux méthodes de la librairie alloy.jar et évalue la propriété traduite (Listing 3).

Listing 3

```
try{
  TransformationDonnees = new Controleur(args[0]);
  TransformationDonnees.LectureCommande();

  switch (chaine){
    case "Composant Isolé":
      TransformationDonnees.Evaluation("ComposantIsolé");
      TransformationDonnees.Parser();
      break;
      ...
    default:
      System.out.println("Aucune commande trouvée");
  }
}
```

Lorsque par exemple, la propriété Composant Isolé (propriété qui vérifie que les composants sont isolés par des vannes) est sélectionnée, son nom est mis dans la chaîne de type

10. IDE : Integrated Development Environment

String. Ensuite, l'exécution de la boucle case permet de choisir la formule Alloy correspondante, d'évaluer et de traiter (Parser()) les résultats. Cette méthode permet de transformer les évaluations de chaque propriété en une liste qui relie chaque propriété aux entités (composants, connecteur,...) qui la violent.

5.4.7.2 Transformation de visualisation

La liste résultante de la première transformation (transformation de données) est, dans cette étape, complétée et transformée en modèle (fichier XML, Listing 4) conforme au méta-modèle de la figure 5.37.

Listing 4

```
<?xml version="1.0" encoding="ASCII"?>
<Visualisation:Visualisation xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:Visalisation="Visualisation"
  xsi:schemaLocation="Visualisation ../Metamodels/Propriete.ecore">
  <propriétés Nom="Composant Isolé">
    <entités id="St2" name="St"/>
    <entités id="TRCH1" name="TRCH"/>
    <message Texte="Chaque composant doit être isolé par une vanne"/>
  </propriétés>
</Visualisation:Visualisation>
```

Nous avons défini ce méta-modèle pour garantir l'indépendance de la liste de toute plateforme graphique. Le méta-modèle de visualisation est composé de l'ensemble des propriétés vérifiées. Chaque propriété est identifiée par un nom et contient à son tour des entités qui sont soit des symboles, soit des connexions. La propriété comporte aussi un message textuel qui sera affiché sur le P&ID. Cette étape de transformation est implémentée par le langage Java.

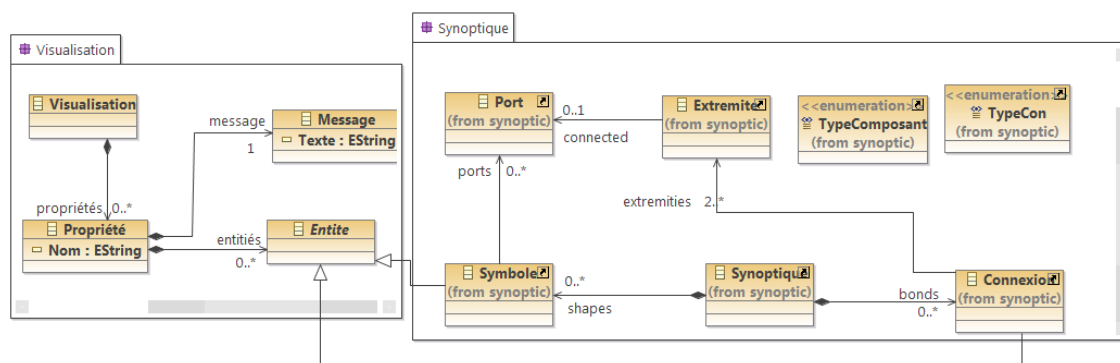


FIGURE 5.37 – Méta-modèle pour la visualisation des erreurs

5.4.7.3 Transformation du visuel graphique

Cette couche définit toutes les formes et les fenêtres de notre interface ainsi que les événements liés aux composants de celle-ci. Elle permet aussi de transformer le fichier XML de l'étape précédente en objets graphiques. Des croix rouges sont créées et positionnées sur les entités qui violent la propriété. Les messages sont transformés en zones de textes et affichés si l'utilisateur clique sur les croix rouges.

La figure 4.8 du chapitre 4 illustre l'outil informatique permettant la visualisation des contre-exemples Alloy.

5.4.8 Opération de correction

L'opération manuelle de correction (figure 5.38) permet au concepteur de porter des corrections, d'une manière itérative, sur P&ID après une visualisation des erreurs (opération précédente). En effet, après l'affichage et la compréhension d'une erreur, le concepteur peut modifier son P&ID et le re-vérifier jusqu'à ce que ce dernier soit correct.

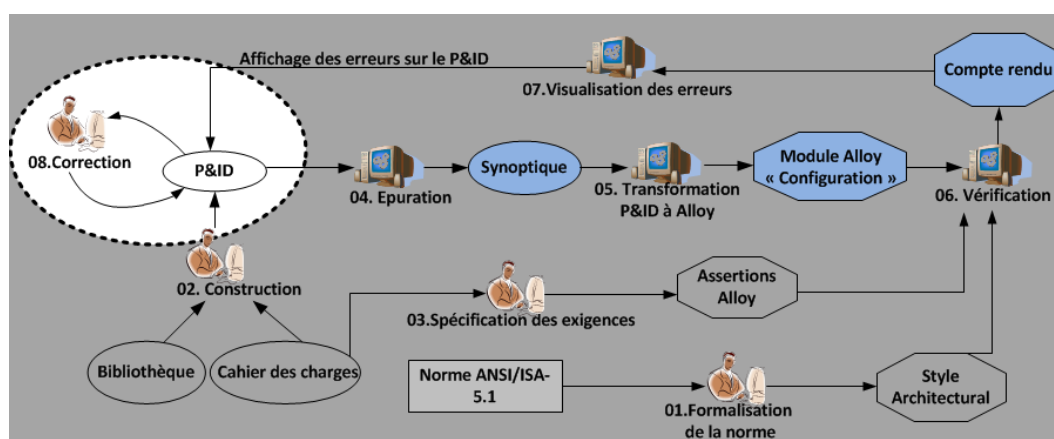


FIGURE 5.38 – Opération de correction

5.4.9 Bilan

En résumé, nous avons développé un outil informatique pour la vérification formelle des diagrammes P&ID et la visualisation des erreurs. Les principales opérations qui interviennent lors d'un processus de vérification et de visualisation peuvent être regroupées de la manière suivante :

1. Sélection de la propriété à vérifier.
2. Transformation des diagrammes P&ID en Alloy.
3. Exécution de la vérification formelle (lecture du modèle Alloy, vérification des propriétés)

4. Extraction du contre-exemple généré (lancement d'évaluateur, suppression des informations inutiles)
5. Affichage (positionnement des croix rouges sur le modèle, info-bulle, messages...)

5.5 Conclusion

Dans ce chapitre, nous avons présenté deux flots de vérification semi-automatisés. Ces flots basés sur l'IDM ont pour objectif général l'introduction de la vérification formelle dans une démarche de conception mixte.

Le premier flot, constitué de huit opérations, a pour but la vérification d'une chaîne de contrôle-commande élémentaire. Il propose essentiellement la génération automatique des automates temporisés à partir des vues de la chaîne élémentaire.

Le deuxième flot a pour but la vérification et la visualisation des erreurs des diagrammes P&ID. Il intègre aussi huit opérations automatiques et manuelles. Les opérations automatiques permettent, d'une part, la génération automatique des modèles Alloy à partir d'un diagramme P&ID, et d'autre part, la visualisation du contre-exemple retourné par l'analyseur Alloy.

Nous nous sommes basés sur le projet Anaxagore pour l'implémentation des deux flots de vérification. Nous présentons dans le chapitre 6, la validation de nos approches sur la bibliothèque de composants standards d'Anaxagore et le diagramme P&ID d'un système de gestion de fluide.

6

Application à des cas industriels

Résumé : Nous présentons dans ce chapitre l'application de nos deux approches de vérification sur un cas d'étude industriel. L'enjeu à travers ce chapitre est double car il permet d'un côté, d'illustrer nos contributions sur un cas concret et, d'un autre côté, d'évaluer la capacité et l'applicabilité de nos approches pour la vérification d'un cas d'étude industriel de taille importante.

Sommaire

6.1	Introduction	150
6.2	Vérification formelle de V2VM : Étude de cas	150
6.2.1	Opération de modélisation de la tâche utilisateur	150
6.2.2	Opération de transformation de HAMSTERS en AT	151
6.2.3	Opération de transformation des IHM SCADA en AT	152
6.2.4	Opération de transformation des programmes LD en AT	154
6.2.5	Opération de modélisation du composant physique en AT	155
6.2.6	Opération de spécification des exigences en CTL	157
6.2.7	Vérification formelle de V2VM	157
6.2.8	Validité	159
6.2.9	Bilan de la vérification formelle de la V2VM	160
6.3	Vérification formelle du P&ID du système EdS : Etude de cas	160
6.3.1	Opération de construction	160
6.3.2	Opération de spécification des exigences	160
6.3.3	Opération d'épuration et de transformation du P&ID en Alloy	164
6.3.4	Opération de vérification	164
6.3.5	Opération de visualisation des erreurs	165
6.3.6	Validité	165
6.3.7	Bilan de la vérification formelle du P&ID du système EdS	166
6.4	Conclusion	167

6.1 Introduction

Le projet Anaxagore adopte une démarche de conception mixte pour la génération automatique des programmes de commande et des interfaces de supervision à partir d'une bibliothèque de composants standards et d'un modèle métier (P&ID). La démarche Anaxagore a été appliquée et évaluée sur un cas d'étude industriel du domaine maritime. Le cas d'étude est plus précisément un système de production, stockage, distribution d'eau douce sanitaire, embarqué dans un navire (nommé EdS dans la suite de ce manuscrit). Nous nous basons sur ce cas d'étude pour évaluer nos deux approches de vérification formelle dans une démarche de conception mixte.

Le P&ID du système EdS a été construit à partir de la bibliothèque de composants standards proposée par Anaxagore. Ensuite, à partir de ce P&ID, une interface de supervision et un programme de commande ont été générés automatiquement pour le système EdS [Bignon 2012].

Ce chapitre est divisé en deux sections. La première section illustre la démarche de vérification formelle d'une chaîne de contrôle-commande élémentaire proposée sur un composant standard de la bibliothèque d'Anaxagore. La deuxième section présente la vérification formelle du diagramme P&ID (modèle métier) du système EdS utilisé pour la génération des applicatifs (interface de supervision et programme de commande).

6.2 Vérification formelle de V2VM : Étude de cas

Dans cette section, nous présentons la validation de notre approche de vérification de la chaîne de contrôle-commande complète sur la vanne deux voies motorisée V2VM. Cet composant a déjà été présenté dans la section 3.2 du chapitre 3.

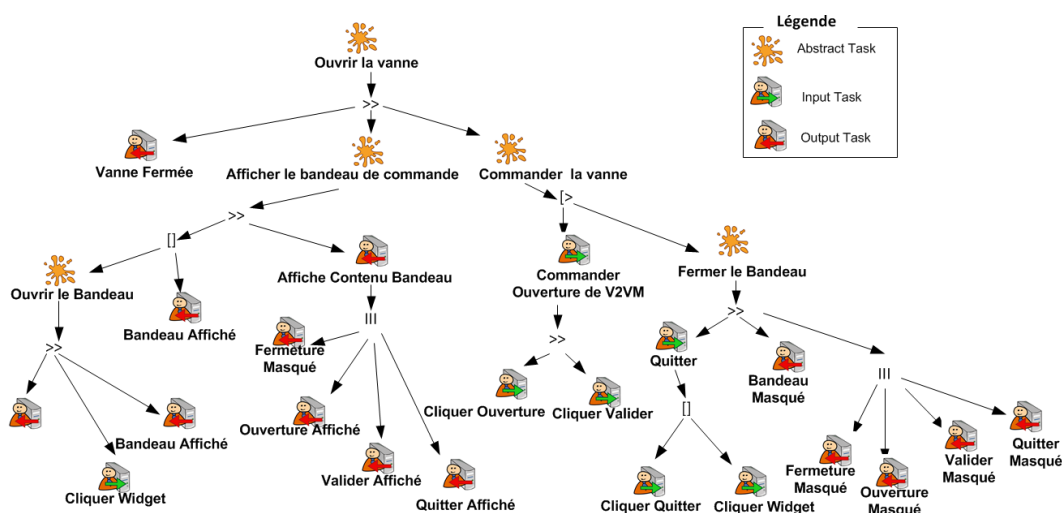
6.2.1 Opération de modélisation de la tâche utilisateur

Pour la V2VM, l'utilisateur peut effectuer trois tâches principales : *Surveiller la vanne*, *Ouvrir la vanne et Fermer la vanne*. La première tâche consiste à surveiller les différents états de la vanne (*Fermée*, *Ouverte*, *En défaut de commande*, *En défaut matériel*) à travers les widgets de l'interface de supervision. Les deux autres tâches consistent à contrôler l'ouverture ou la fermeture de la vanne par requête de l'utilisateur à partir de l'IHM.

La tâche *Ouvrir la vanne* (tâche abstraite) est composée de trois sous-tâches séquentielles : *Vanne Fermée*, *Afficher le bandeau de commande* et *Commander la vanne*. La tâche *Vanne Fermée* (tâche système) signifie que pour ouvrir la vanne, cette dernière doit être affichée *Fermée* à l'utilisateur. Ensuite, l'utilisateur doit faire *Afficher le bandeau de commande*, afin de commander la vanne.

Deux cas se présentent : soit le bandeau de commande est déjà affiché, représenté par la sous-tâche *Bandeau Affiché* ; ou bien le bandeau n'est pas affiché, représenté par la sous-tâche

Ouvrir le bandeau. Cette dernière tâche est composée de 3 sous-tâches séquentielles (*Bandeau Masqué*, *Cliquer Widget*, *Bandeau Affiché*), ce qui veut dire qu'initialement le bandeau de commande est masqué. L'utilisateur doit alors cliquer sur le widget de V2VM (tâche utilisateur) pour afficher le bandeau. Une fois le bandeau affiché, les boutons (*Ouverture*, *Valider*, *Quitter*), contenus dans le bandeau, doivent être affichés simultanément et le bouton *Fermeture* doit être masqué. Cette tâche est représentée sur la figure 6.1 par *Affiche Contenu Bandeau*, elle est composée de 4 sous-tâches réalisées simultanément. L'utilisateur pourra ensuite commander l'ouverture de la vanne (*Cliquer Ouverture*), puis, valider (*Cliquer Valider*) ou bien, faire disparaître le bandeau en cliquant sur le bouton *Quitter* (*Cliquer Quitter*) ou bien sur le widget de V2VM (*Cliquer widget*). À la fin de cette tâche, le bandeau est masqué (*Bandeau Masqué*) ainsi que les boutons *Ouverture*, *Fermeture*, *Valider* et *Quitter*. Cette tâche fait disparaître le bandeau et en même temps désactive, par l'opérateur $[>$, la tâche *Commander l'ouverture de la vanne*.

FIGURE 6.1 – La tâche *Ouvrir la vanne* en HAMSTERS

6.2.2 Opération de transformation de HAMSTERS en AT

La figure 6.2 illustre l'automate temporisé représentant la tâche *Ouvrir la vanne* généré automatiquement à partir du modèle de tâches (figure 6.1).

Dans l'objectif de commander l'ouverture de la vanne, l'utilisateur doit recevoir, à partir de l'état initial, le message (*VanneFermee_Affichée ?*) lui indiquant que la vanne est fermée. Ensuite, il doit afficher le bandeau de commande. Ce dernier peut être déjà affiché (réception du message *Bandeau_Affiché ?*) ou bien fermé (réception du message *Bandeau_Masqué ?*). Dans ce dernier cas, l'utilisateur doit cliquer sur le widget de la vanne (envoi du message *Cliquer_Widget !*) pour afficher le bandeau (réception du message *Bandeau_Affiché ?*). Les boutons ouverture, valider et quitter contenus dans le bandeau sont alors affichés simultanément

(opérateur de concurrence). Le bouton de fermeture doit être masqué car la vanne est fermée (*Fermeture_Masque?*). À ce stade (état *Commander_Ouverture_Vanne*), l'utilisateur peut commander l'ouverture de la vanne en cliquant sur le bouton ouverture (*Cliquer_Ouverture!*), puis sur valider (*Cliquer_Valider!*) pour valider son choix. Il peut ensuite fermer le bandeau en cliquant soit sur le bouton quitter (*Cliquer_Quitter!*) soit sur le widget de la vanne (*Cliquer_Widget!*). Le bandeau et les quatre boutons sont alors masqués.

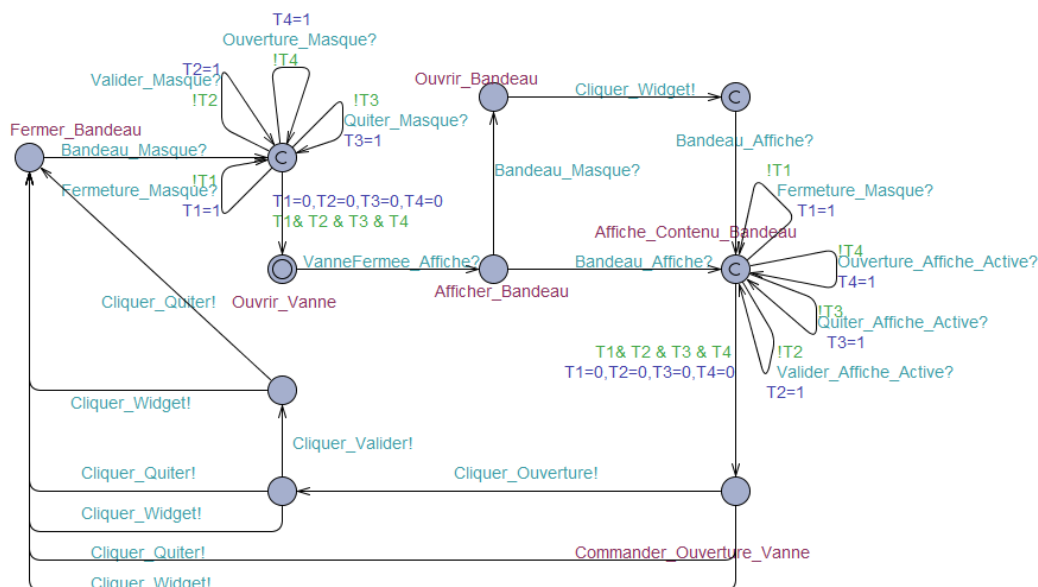


FIGURE 6.2 – Automate temporel généré à partir du modèle HAMSTERS de la figure 6.1

Remarque. Le comportement de l'utilisateur est modélisé par :

- 2 Automates temporels similaires correspondant aux tâches "Ouvrir la vanne" (figure 6.2) et "Fermer la vanne" (en annexe C) ;
- 1 Automate temporel modélisant la tâche "Superviser la vanne" (en annexe C).

6.2.3 Opération de transformation des IHM SCADA en AT

L'interface de supervision de la V2VM est implémentée dans le logiciel Panorama E2 (figure 6.3).

La vanne est constituée des vues V2VM et V2VM_Control qui correspondent respectivement au widget de V2VM et au bandeau de commande. La vanne est dotée aussi de deux alarmes (V2VM_AL_StFCmd et V2VM_AL_StFMat) qui se déclenchent respectivement dans le cas de défaut de commande et de défaut de matériel.

Le bandeau de commande (V2VM_Control sur la figure 6.3) est composé de plusieurs *Items* : les boutons ouverture, fermeture, valider et quitter, une zone de texte et une image.

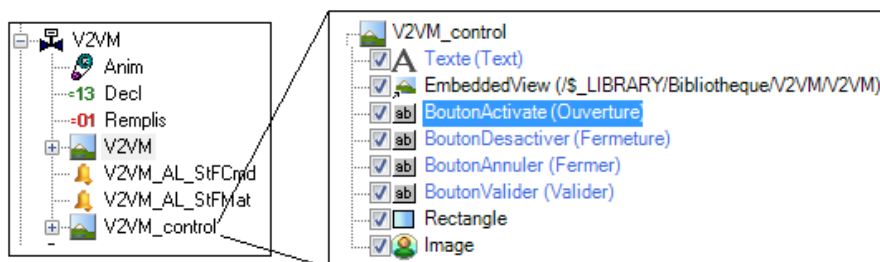


FIGURE 6.3 – Implémentation de la V2VM dans Anaxagore

Chaque *Item* englobe des propriétés de visibilité et peut aussi contenir des propriétés de commande. Ces propriétés sont utilisées pour paramétrer les automates génériques que nous avons proposés pour modéliser les objets de commande et les objets informationnels (section 3.4.3.2.1 dans le chapitre 3).

Nous obtenons pour le bouton ouverture, par exemple, l'automate temporisé de la figure 6.4.

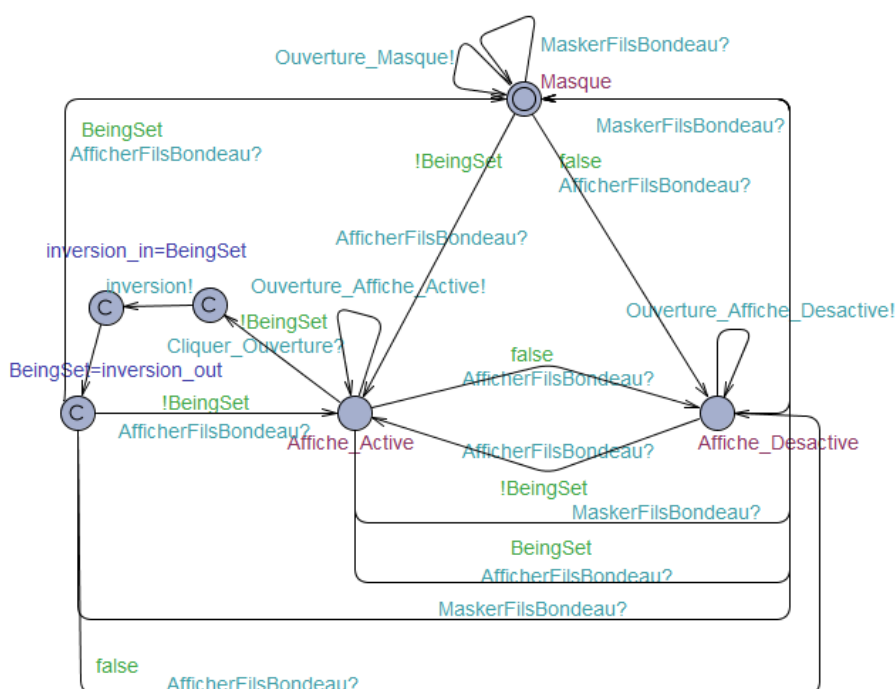


FIGURE 6.4 – Automate temporisé généré à partir du bouton Ouverture

Nous remarquons que le bouton est affiché une fois que la variable *BeginSet* est égale 0, elle représente sa condition d'activation. Si l'utilisateur clique sur le bouton (*Cliquier_Ouverture!* dans le modèle de tâche), cette information est interceptée (*Cliquier_Ouverture?*) par l'objet de

commande relié au bouton ouverture. L'objet de commande appelle alors la fonction prédéfinie *Inversion*. Cette dernière permet d'inverser la variable *BeingSet* à 1, ce qui signifie que la condition d'activation du bouton ouverture devient fausse, et par conséquent, celui-ci atteint l'état *Masqué*.

Remarque. Le module de dialogue de V2VM est composé de :

- 7 objets informationnels dont 4 modélisant les états *Fermée*, *Ouverte*, *En défaut de commande*, *En défaut matériel* de la vanne, 2 modélisant les alarmes (*Défaut de commande*, *Défaut matériel*) et 1 pour la vue du bandeau de commande ;
- 6 objets de commande, dont 4 modélisant les boutons (*Ouverture*, *Fermeture*, *Quitter*, *Valider*), et deux autres modélisant le widget V2VM.

6.2.4 Opération de transformation des programmes LD en AT

Le programme LD de la V2VM est illustré dans la figure 6.5.

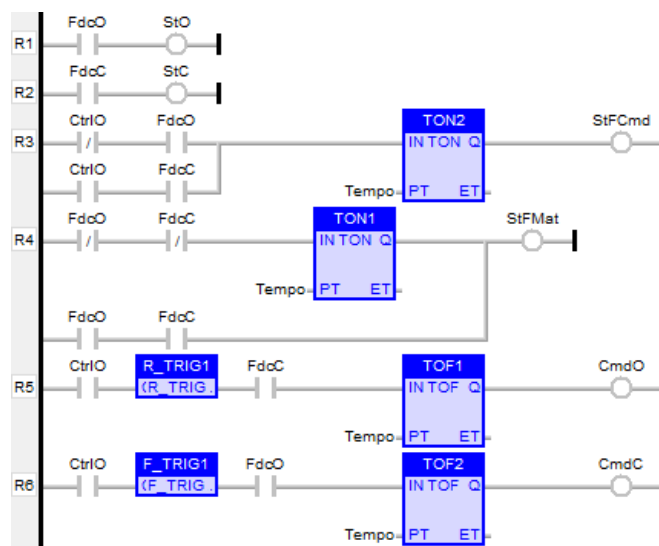


FIGURE 6.5 – Programme LD de la V2VM

Il est constitué de six Rungs. Les deux premiers rungs permettent la transmission de l'état (FdcO et FdcC) de la vanne vers l'interface de supervision (StO et StC) pour que cet état soit affiché à l'utilisateur. Les rungs R3 et R4 se chargent de détecter respectivement le défaut de commande (StFCmd) et le défaut de matériel (StFMat). Un défaut de commande est déclenché si la vanne reste ouverte (FdcO dans la première branche) ou fermée (FdcC dans la deuxième branche) durant un délai dépassant la temporisation (Tempo) après une demande de fermeture (!CtrlO) respectivement d'ouverture (CtrlO). Les rungs R5 et R6 permettent de déclencher les ordres d'ouverture (CmdO) et de fermeture (CmdC) à envoyer vers la vanne physique. Un ordre d'ouverture (CmdO) est émis après un front montant de la variable CtrlO (provenant de

l'interface de supervision) et une position fermée de la vanne (FdcC). Cet ordre est maintenu pendant une certaine temporisation (TOF).

Après application des règles de transformation sur le programme de commande de V2VM, nous obtenons l'automate temporisé présenté sur la figure 6.6. À l'état initial, le contrôleur est dans l'état *Lecture Entrées*, dans lequel il lit des entrées (CtrlO, FdcC, FdcO) provenant respectivement de l'interface de supervision, du capteur de fin de course-fermeture et du capteur de fin de course-ouverture (figure 6.6), puis il met à jour les entrées des temporisateurs (TON1 et TON2) et des blocs (RTRIG et FTRIG). Ensuite, il atteint l'état *Traitements*, où il commence le calcul des sorties. Les différents calculs sont représentés par des localités instantanées, ce qui signifie que ce sont des états immédiats. Il invoque successivement TON1, par l'envoi du message (*TON1!*), TON2 (*TON2!*), RTRIG (*RTRIG!*), TOF1 (*TOF1!*), FTRIG (*FTRIG!*) et TOF2 (*TOF2!*). Ensuite, le contrôleur atteint l'état *Ecriture Sorties* qui signifie que toutes les sorties sont calculées. Dans cet état, il met à jour les sorties calculées (StO, StC, StFMat, StFCmd) et (CmdO, CmdC) qu'il partage respectivement avec l'interface de supervision et la vanne physique. L'horloge *cycle* est remise à 0 (*cycle=0*) pour le prochain cycle.

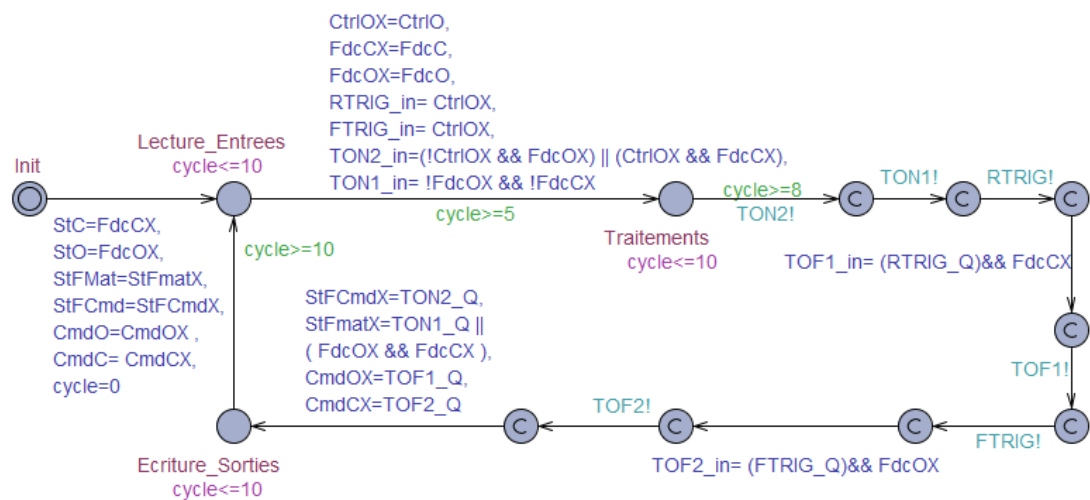


FIGURE 6.6 – Automate temporisé généré automatiquement à partir du programme LD de la figure 6.5

6.2.5 Opération de modélisation du composant physique en AT

La partie physique représente le comportement des deux capteurs fin de course-fermeture (figure 6.7-a) et fin de course-ouverture (figure 6.7-b), ainsi que celui de V2VM (figure 6.7-c). Le capteur fin de course-ouverture met la variable booléenne FdcO à 1 si la vanne est ouverte, sinon à 0. Le capteur de fin de course-fermeture met la variable booléenne FdcC à 1 si la vanne est fermée, sinon à 0.

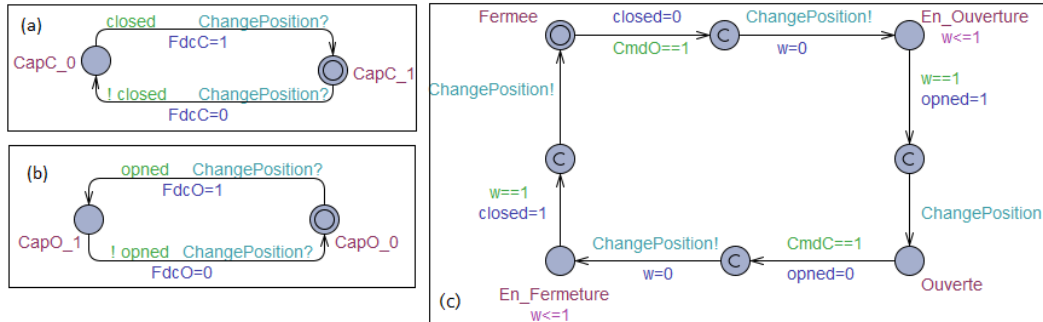


FIGURE 6.7 – Modélisation du composant matériel : (a) Capteur fin de course-fermeture ; (b) Capteur fin de course-ouverture ; (c) La vanne

L'automate de la vanne est composé de 4 états (*Fermée*, *Ouverte*, *En Ouverture*, *En Fermeture*). À l'état initial, la vanne est *Fermée*, les capteurs de fin de course-ouverture et fin de course-fermeture sont respectivement à l'état *CapO_0* et *CapC_1*, ce qui signifie que les variables booléennes *FdcO* et *FdcC* sont respectivement à 0 et à 1. Si la vanne reçoit la commande d'ouverture ($\text{CmdO}=1$) du programme de commande, elle commence à s'ouvrir, elle met la variable *closed* à 0 et signale aux capteurs qu'elle change d'état, en envoyant le message broadcast (*ChangePosition!*). Quand elle atteint l'état *En ouverture*, le capteur de fin de course-fermeture passe de l'état *CapC_1* à l'état *CapC_0* (la vanne est entre-ouverte et fermée). L'état *En ouverture* de la vanne est doté d'une horloge w . On suppose que la vanne met une seconde pour s'ouvrir. Quand la seconde est passée, la vanne quitte l'état *En ouverture* et met la variable *opened* à 1. Ensuite, elle envoie le signal (*ChangePosition!*) aux capteurs, ce qui permet au capteur de fin de course-ouverture de passer à l'état *CapO_1* ($\text{FdcO}=1$) et à la vanne d'atteindre l'état *Ouverte*. Un processus similaire est suivi pour la fermeture de la vanne.

6.2.5.1 Scénario d'ouverture d'une vanne

Initialement, la vanne est fermée. La tâche *Vanne Fermée* de la figure 6.1 est représentée par la réception du message (*VanneFermée?*) dans la figure 6.2, ce qui veut dire que pour l'utilisateur, la vanne s'affiche fermée. Ensuite, pour commander la vanne, le bandeau de commande doit être affiché. Si l'automate temporisé modélisant le comportement de l'utilisateur reçoit le message (*BandeauAffiché?*), cela signifie que le bandeau de commande est affiché. L'utilisateur peut alors commander la vanne, sinon, l'utilisateur reçoit le message (*BandeauMasqué?*), signifiant que le bandeau est masqué. Dans ce cas, l'utilisateur doit afficher le bandeau, en cliquant sur le widget de V2VM, représenté par l'envoi du message (*CliquerV2VM!*). Ainsi, l'objet relié au widget de V2VM dans le module du dialogue reçoit le message (*CliquerV2VM?*) qui inverse la condition d'activation du bandeau de 0 à 1. Cela permet au bandeau de commande (objet informationnel) qui était à l'état *Masqué* d'atteindre l'état *Affiché*. Dans

cet état, il envoie le message (*BandeauAffiché !*) à l'utilisateur, qui signifie que le bandeau vient d'être affiché. Une fois le bandeau affiché, il envoie à tous ses fils (*boutons Ouverture, Fermeture, Valider, Quitter*) le message (*VueMereAffichée !*). Chaque bouton teste alors sa condition d'activation et de désactivation pour atteindre l'un des états *Masqué, Affiché Activé* ou *Affiché Désactivé*. Alors, l'utilisateur (figure 6.2) reçoit simultanément les messages (*OuvertureAffichéActivé ?*), (*FermetureMasqué ?*), (*ValiderAffichéActivé ?*) et (*QuitterAffichéActivé ?*), représentés par l'état *Fils BandeauAffichés*. Cet état est modélisé par une localité instantanée afin de modéliser la concurrence. Ensuite, l'utilisateur clique sur le bouton *Ouverture* (envoi du message (*CliquerOuverture !*)) puis il valide son choix (il clique sur le bouton *Valider*). L'objet de commande lié au bouton *Valider* met la variable *CtrlO* à 1. Cette information, qui veut dire que l'utilisateur demande l'ouverture de la vanne, est émise vers l'automate temporisé modélisant le programme de commande. Ce dernier (figure 6.6) exécute un cycle et met, à la fin du cycle, la variable *CmdO* à 1 (envoi de la commande d'ouverture vers la vanne physique). La vanne physique reçoit la commande et passe alors de la position *Fermée* à la position *Ouverte*. Cette information (*FdcO=1* et *FdcC=0*) est transmise au programme de commande qui met à jour les variables *StO=1* et *StC=0*, puis, il les envoie à la supervision. Quand l'interface de supervision reçoit cette variable (condition d'activation de l'objet informationnel lié au widget *Vanne Ouverte*), elle affiche la vanne en état *Ouverte* à l'utilisateur.

6.2.6 Opération de spécification des exigences en CTL

Au niveau interface, nous proposons de vérifier des propriétés de visibilité pour garantir l'utilisabilité de l'IHM :

P1. $A \square \text{not } (OuvertureAffichéActivé \ \&\& \ FermetureAffichéActivé)$. Les deux boutons *Ouverture* et *Fermeture* ne doivent jamais être affichés en même temps.

Une propriété de sûreté est à vérifier au niveau du programme de commande :

P2. $A \square \text{not } (CmdO==1 \ \&\& \ CmdC==1)$. Le programme de commande ne peut jamais envoyer en simultané une commande d'ouverture (*CmdO*) et de fermeture (*CmdC*) à la vanne.

Une propriété qualitative à vérifier sur tout le composant est le non blocage :

P3. $A \square \text{not } \text{deadlock}$.

6.2.7 Vérification formelle de V2VM

Le modèle AT complet de V2VM avec ses vues (modèle du comportement de l'utilisateur, modèle de l'interface, modèle de la commande et modèle de la partie opérative) est composé de 107 états, 246 transitions, 48 variables, 6 horloges et 52 canaux de communication (tableau 6.1).

Nous avons effectué la vérification sur une machine avec un microprocesseur i5 de fréquence 2,7 Ghz et 8 GO de mémoire. Nous avons utilisé la version académique UPPAAL-4.1.19.

	V2VM
États	107
Transitions	246
Variables	48
Horloges	6
Canaux de communication	52
Temps de vérification de P1	169 secondes
Temps de vérification de P2	0.87 secondes
Temps de vérification de P3	0.028 secondes
Temps total de vérification	169.898 secondes (2.83 minutes)

Tableau 6.1 – Composition du modèle complet de la V2VM

Les résultats de la vérification montrent que, seule la propriété **P1** est vérifiée formellement sous UPPAAL.

La propriété **P2** n'est pas vérifiée. Nous avons simulé la trace d'exécution du contre-exemple retourné par UPPAAL sur le programme de commande avec le logiciel Straton et nous avons obtenu le même résultat (figure 6.8). Nous remarquons sur la figure 6.8 que les deux sorties CmdO et CmdC sont égales toutes les deux à 1. L'envoi simultané de ces ordres peut endommager la vanne physique ou la bloquer.

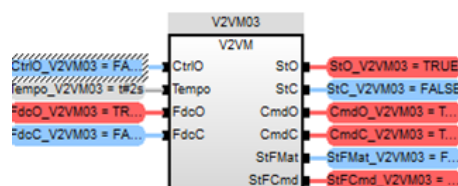


FIGURE 6.8 – Simulation du contre-exemple dans Straton

Plusieurs itérations de correction ont été réalisées et à chaque fois qu'un contre-exemple est retourné, nous obtenons les mêmes résultats en simulant le programme de commande lui-même. Nous avons proposé de corriger les rungs 5 et 6 du programme LD de la vanne (figure 6.5) respectivement en, $CmdO = TOF1_Q \ \&\& \ !FdcO \ \&\& \ !CmdC$ et en $CmdC = TOF2_Q \ \&\& \ !FdcC \ \&\& \ !CmdO$. Après cette modification, la propriété **P2** est vérifiée.

La propriété **P3** n'est pas vérifiée sur la version initiale du composant. Le programme se bloque et ne permet plus à l'utilisateur d'accomplir la tâche *Ouvrir la vanne*. Le contre-exemple indique qu'initialement, la vanne est *Fermée* ($StC=1$) et la variable booléenne *BeingSet*, qui gère respectivement la condition d'activation et de désactivation des boutons *Ouverture* et *Fermeture*, est égale à 0. L'utilisateur affiche le bandeau de commande, le bouton *Ouverture* est affiché et le bouton *Fermeture* est masqué. L'utilisateur demande l'ouverture de la vanne, en cliquant sur le bouton *Ouverture*. Cette action met la variable *BeingSet* à 1. Les boutons *Ouverture* et *Fermeture* deviennent respectivement masqué et affiché. L'utilisateur clique sur *Quitter*

(le bandeau est masqué) sans valider son choix (la commande n'est pas envoyée au programme de commande). L'utilisateur pense qu'il a ouvert la vanne, mais la vanne est toujours affichée fermée ($StC=1$). L'utilisateur ouvre le bandeau de commande pour ouvrir la vanne, mais le bouton *Ouverture* est masqué. L'utilisateur est bloqué devant l'interface.

La solution consiste à relier les conditions d'activation et de désactivation des boutons *Ouverture* et *Fermeture* aux variables StO et StC . De plus, quand l'utilisateur clique sur *Ouverture*, le bouton doit être grisé en attendant la réponse du système. Le bouton *Fermeture* s'affichera après que l'utilisateur ait obtenu une réponse (vanne affichée ouverte, ou bien en défaut).

6.2.8 Validité

Nous avons présenté dans cette section une étude de cas. Nous discutons, ci-après, les précautions que nous avons prises pour garantir la qualité de notre étude. Selon Runeson et Höst [2009], les menaces de validité d'un cas d'étude sont :

La validité du construit ou du concept consiste à s'assurer que l'étude réalisée mesure bien ce qui était prévu de mesurer. Au niveau modélisation formelle, cette validité peut être affectée si les modèles formels ne reproduisent pas les comportements des vues du composant. Pour minimiser cette menace, la construction des modèles formels a été réalisée sur plusieurs étapes. Dans chaque étape, nous avons comparé les traces d'exécution avec les résultats de la simulation des codes des différentes vues. Nous sommes confiants sur le fait que les modèles formels produisent des comportements similaires aux comportements des vues, car la simulation des traces d'exécution du contre-exemple retourné pour les propriétés non vérifiées donne les mêmes résultats (figure 6.8).

La validité interne porte sur la cohérence des conclusions tirées sur les causes et effets entre les résultats. Ce type de validité concerne davantage les études quantitatives pour lesquelles les relations de causalités sont calculables. Notre étude est qualitative donc moins concernée par ce type de validité.

La validité externe porte sur le pouvoir de généralisation des conclusions aux autres populations, domaines, etc. Nous abordons cette validité selon trois axes : la modélisation, la génération automatique et la vérification formelle. Au niveau de la modélisation formelle, notre proposition de modélisation des interfaces de supervision SCADA peut être généralisée à d'autres types d'interface de supervision. Cependant, elle reste spécifique aux interfaces de supervision industrielles. Au niveau de la génération automatique, les transformations des modèles Panorama, Straton et HAMSTERS en automates temporisés sont spécifiques à ces outils, et par conséquent, sont difficilement généralisables à d'autres outils. Cependant, pour la transformation des programmes LD, nous avons proposé une étape intermédiaire permettant de normaliser les modèles et les rendre conformes à la norme IEC 61131-3, ce qui augmente leur généralisabilité.

La fiabilité de l'étude consiste à la capacité de reproduction et de réplique des résultats par d'autres chercheurs. Pour augmenter la fiabilité de notre étude, les transformations de mo-

dèles et les modèles formels sont mis sur le réseau de l'entreprise pour une future utilisation ou une réplique. Nous avons décrit en détails, dans le chapitre 5, les transformations de modèles et les modèles formels pour faciliter leurs réutilisations. Le modèle formel complet de la vanne est mis en annexe C dans ce manuscrit.

6.2.9 Bilan de la vérification formelle de la V2VM

Nous avons présenté dans cette section la validation de notre approche de vérification formelle d'une chaîne de contrôle-commande élémentaire. Cette validation consiste en l'application de notre approche sur un cas représentatif tiré de la bibliothèque Anaxagore. En effet, l'approche a été validée sur le cas d'une vanne deux voies motorisée.

Les résultats de notre *systematic mapping* (chapitre 2) couplés avec ceux déduits des entretiens avec les experts, nous ont permis de définir les propriétés à vérifier sur une chaîne de contrôle-commande élémentaire. Sur la V2VM, un ensemble de propriétés a été écrit en CTL et vérifié par l'outil UPPAAL. La vérification par *Model-Checking* a permis de détecter des erreurs de conception dans les programmes de commande et les IHM de supervision des composants standards. Des solutions ont été alors proposées afin de corriger ces erreurs.

Les résultats de cette validation sont encourageants et montrent que le *Model-Checking* peut être utilisé dans un contexte industriel afin de détecter les erreurs de conception.

6.3 Vérification formelle du P&ID du système EdS : Etude de cas

Dans cette section, nous présentons l'application de notre approche de vérification formelle des diagrammes P&ID sur le P&ID du système EdS. Pour assurer ces fonctionnalités, le système EdS comporte plusieurs composants comme des vannes et des pompes qui assurent la circulation de l'eau, des soutes pour le stockage, des clapets anti-retour, des capteurs, etc. Ces composants sont reliés les uns aux autres soit par des tuyaux (liens process), soit par des liens électriques ou logiciels.

6.3.1 Opération de construction

La figure 6.9 présente un extrait du diagramme P&ID du système EdS. Il regroupe 34 instances reliées par 59 connexions.

Ce P&ID doit respecter les contraintes du style architectural définies précédemment et un ensemble d'exigences qui proviennent du cahier des charges, des normes, des règles métier, etc. Nous suivons le protocole proposé par [Wohlin et al. \[2012\]](#) pour réaliser une étude exploratoire de l'étude de cas afin d'extraire les exigences auxquelles ce système doit répondre.

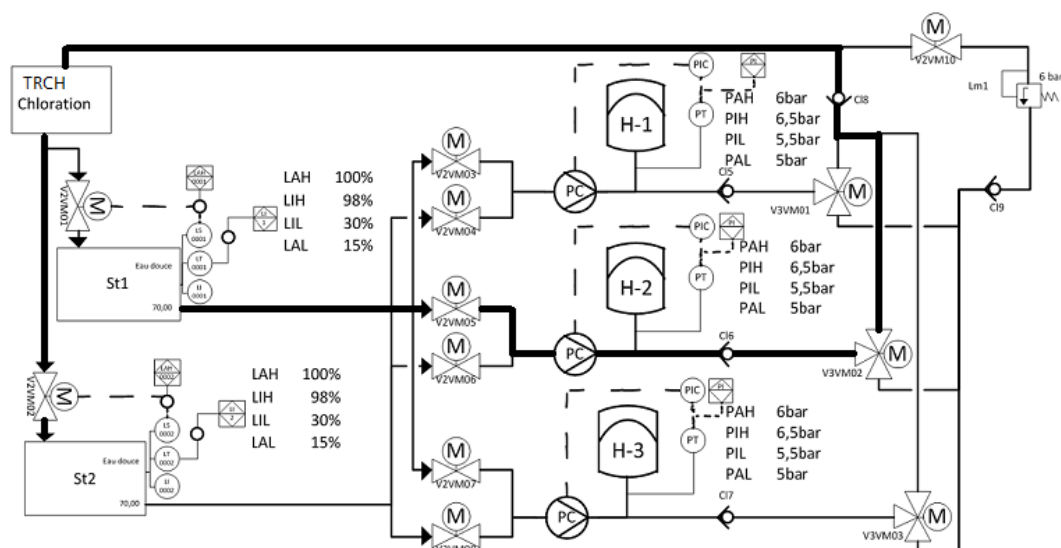


FIGURE 6.9 – Un extrait du diagramme P&ID du système EdS

6.3.2 Opération de spécification des exigences

Le but de cette opération est d'extraire et de définir les différentes exigences que le P&ID du système EdS doit satisfaire.

6.3.2.1 Elicitation des exigences

Afin d'extraire et de classer les différentes exigences, nous avons combiné plusieurs méthodes (indépendantes et directes). Premièrement, nous avons récupéré la documentation technique du projet (le cahier des charges, les spécifications, les patterns...). Après une analyse de cette documentation, nous nous sommes aperçus que le système doit aussi respecter un ensemble de normes et de standards mentionnés dans le cahier des charges. Nous avons aussi récupéré ces normes et standards que nous avons étudiés afin d'extraire de nouvelles exigences et compléter l'ensemble initial des exigences. Nous avons constaté que, en plus de ces exigences, des règles métier sont prises en compte par les concepteurs lors de la spécification des diagrammes P&ID. Pour capturer les connaissances des concepteurs, nous avons réalisé des entretiens semi-directifs (méthode directe) avec cinq experts métiers (ingénieurs) ayant tous une expérience significative (10 ans en moyenne) dans l'élaboration des diagrammes P&ID.

Les questions de l'entretien (annexe B) ont été validées par un spécialiste. Les entretiens, de 1h30 à 1h45, ont été enregistrés sous forme audio puis transcrits pour l'extraction des données. L'extraction des données a été, ensuite, réalisée et validée par trois professeurs et deux experts industriels.

Nous présentons dans les sections suivantes, les différentes catégories qui ont émergé de notre étude exploratoire.

6.3.2.2 Définition des exigences de conception

Selon l'INCOSE¹ [Haskins 2010], les exigences peuvent être fonctionnelles, non-fonctionnelles et également des contraintes d'architecture ou de performances. Les exigences fonctionnelles sont des contraintes liées aux différentes fonctions du système. Elles représentent "*Ce que le système doit faire*". Les exigences non-fonctionnelles, nommées "ilities", sont des contraintes qui décrivent la qualité opérationnelle que le système doit avoir [Haskins 2010]. Les contraintes d'architecture sont toutes les contraintes liées à l'architecture ou à la conception. Les exigences de performances sont des mesures quantitatives qui caractérisent les fonctionnalités du système. Dans cette étude, nous proposons uniquement la vérification des exigences fonctionnelles, non-fonctionnelles et celles liées à l'architecture. Les exigences de performances ne sont pas vérifiées sur le P&ID, car ce dernier reste un schéma abstrait qui ne fournit pas des informations quantitatives liées aux performances du système.

6.3.2.2.1 Les contraintes d'architecture

Les contraintes d'architecture regroupent les standards appliqués, les exigences physiques, les exigences de coût et des langages de programmation utilisés. Par exemple, dans notre cas d'application, ces exigences impliquent la compatibilité du P&ID avec le style architectural et la présence d'une certaine instrumentation sur des composants spécifiques selon les différentes normes appliquées.

6.3.2.2.2 Les exigences fonctionnelles

Le système EdS doit assurer sept fonctions : le transfert, le traitement, la distribution, la distribution à partir du quai, la production, l'embarquement et le débarquement.

La fonction de transfert, par exemple, doit assurer le transfert d'un volume V d'eau, d'une soute A vers une autre soute B, en passant obligatoirement par l'un des groupes hydrophores (H1, H2, H3). Cette exigence fonctionnelle est représentée sur le P&ID (figure 6.9) par un chemin (en gras) entre la soute A et la soute B en passant par l'un des hydrophores. Pour cela, le concepteur a mis en séquence les composants suivants : la soute St1, le groupe hydrophore (pompe) H1, le module de chloration TRCH, utilisé pour traiter l'eau en chlore, et enfin la soute St2. Ces composants sont séparés par des vannes deux voies motorisées (V2VM02, V2VM05) et une vanne trois voies motorisée (V3VM02) pour faciliter la maintenance. Des clapets anti-retour (CI6, CI8) sont aussi utilisés pour empêcher la circulation de fluide dans le sens contraire.

Pour vérifier que le diagramme P&ID respecte les différentes fonctions, nous avons utilisé les fonctions et les prédicats du Listing 1. La fonction `ComposantConnecté` retourne un tuple de composants qui sont liés par un même connecteur. Le prédicat `ExistChemin` retourne vrai si le composant destination `dest` peut être atteint à partir d'un composant source `sc`, en passant

1. INCOSE : International Council on Systems Engineering

obligatoirement par les composants pas. Le prédicat `pred Transfert` détermine si un chemin existe entre une soude (St) de départ et une soude d'arrivée passant par les pompes (HP).

Listing 1

```

fun ComposantConnecté:Composant->Composant {
  {s1,s2: Composant | ( disj[s1,s2] && not ((s1.ports).(s1.actions) .
  ~(connected).RoleConnecté.(connecté).(s2.actions) =none )}}

fun RoleConnecté: Role->Role{
  {disj r1,r2: Role | connecteur[r1] = connecteur[r2]}}

pred ExistChemin[sc:Composant , pas:set Composant , dest:Composant ]{
  (pas in sc.^(ComposantConnecté)) &&
  (dest in pas.^(ComposantConnecté))}

pred Transfert [sc:Composant , pas:set Composant , dest:Composant ]{
  ExistChemin[sc, pas, dest] && (sc in St) &&
  (dest in St) && (pas in HP)}

```

6.3.2.2.3 Les exigences non-fonctionnelles

Nous présentons ci-dessous des exemples de ces exigences appliquées au système EdS. Ils représentent les qualités des fonctions offertes telles que :

- Efficacité globale : par exemple, la pression dans la tuyauterie ne doit pas dépasser 6 bars ou la production de 1000 litres d'eau douce par jour.
- Fiabilité : ces exigences déterminent les redondances matérielles nécessaires (par exemple 3 pompes) et le temps de fonctionnement des composants.
- Maintenabilité : chaque composant "non isolant" (pompe, Soude...) doit être précédé et suivi par une vanne pour faciliter la maintenance.
- Sécurité : comme la présence des clapets anti-retour pour interdire la circulation de flux dans des sens contradictoires (menant à une collision des flux). Il ne faut jamais un clapet anti-retour en amont d'une pompe.
- Facilité d'utilisation : comme la présence de l'instrumentation (capteurs) afin de gérer les alarmes dans les salles machine sur les interfaces de supervision industrielles.
- Evolutivité : facilité d'extension par d'autres composants

Dans le Listing 2, nous modélisons l'exigence de maintenabilité (`pred ComposantIsolé`). Tous les composants (TRCH+St+HP) de type TRCH (traitement en chlore) ou St (soude) ou HP (pompe) doivent être précédés (`ComposantIsoléAmont`) et suivis (`ComposantIsoléAval`) par des vannes (V2VM+V3VM) de façon à les isoler facilement.

Listing 2


```

pred ComposantIsolé {
  let vannes = V2VM+V3VM | all s:TRCH+St+HP |
  some (ComposantIsoléAmont[s] & vannes) &&
  some(ComposantIsoléAval[s] & vannes)}

fun ComposantIsoléAmont[s:Composant]:Composant{
{c: V2VM+V3VM| some cl:C1 | c in s.ComponentConnecté ||
((cl in s.ComposantConnecté) && (c in cl.ComponentConnecté))}}

fun ComposantIsoléAval[s: Component]:Component{
{c: V2VM+V3VM | some cl:C1 | s in c.ComponentConnecté ||
((s in cl.ComponentConnecté) && (cl in c.ComponentConnecté))}}

```

6.3.3 Opération d'épuration et de transformation du P&ID en Alloy

Le P&ID du système EdS est saisi dans l'outil Microsoft Visio. Ce diagramme est ensuite traité et épuré lors de l'opération d'épuration. À la fin de cette opération, on obtient un fichier *Synoptique* (XML en figure 5.30, chapitre 5) qui contient uniquement les données pertinentes concernant le schéma P&ID.

Le modèle synoptique est utilisé pour générer des modèles Alloy. La transformation du diagramme P&ID a été réalisée en 8 secondes. Le module EdS qui représente la moitié d'un système auxiliaire complexe de grande taille dans un navire [Bignon et al. 2013], est composé de 176 signatures (sig).

6.3.4 Opération de vérification

La vérification a été effectuée sur une machine avec un microprocesseur i5 de fréquence 2,7 Ghz et 8 GO de mémoire. Nous avons effectué l'analyse par l'outil d'Alloy de la version 4.2_2015-02-22 avec un solveur SAT4J.

Le module correspondant au style architectural est composé de 46 signatures, 7 prédicats, 6 fonctions et 1 fait (fact). Ce module est importé dans le module correspondant au système EdS généré lors de la précédente opération et est complété par une assertion et une commande Check (Listing 3). Le modèle complet est alors composé de 222 signatures, 7 prédicats, 6 fonctions, 1 fait, 1 assertion et 1 commande. Ce nombre est dû à la nature industrielle de notre étude de cas (238 paragraphes au total).

Sur ce modèle, nous vérifions la correction, la complétude, la cohérence et la compatibilité du diagramme avec le style architectural. La correction porte sur le respect des exigences fonctionnelles et non-fonctionnelles. Ces exigences ont été modélisées par des assertions. L'assertion `assert PID_verification` (Listing 3), par exemple, vérifie que chaque composant est bien isolé (`ComposantIsolé`) par des vannes et que le diagramme assure la fonction de transfert (`Transfert[St2, HP, St1]`) ente les soutes St1 et St2. Nous vérifions ensuite l'exis-

tence ou non de contre-exemples en utilisant la commande `check PID_verification for 1`.

Listing 3

```
assert PID_verification {
  ComposantIsolé && Transfert[St1,HP,St2]}

check PID_verification for 1
```

Les résultats montrent que le P&ID du système EdS est complet, cohérent et compatible avec le style. La vérification de la correction a pris 1,13 minutes et n'a pas renvoyé de contre-exemple. Ce résultat signifie que le modèle répond à ses exigences fonctionnelles et non-fonctionnelles.

6.3.5 Opération de visualisation des erreurs

La vérification du diagramme P&ID du système EdS n'a pas retourné de contre-exemple. Donc, aucune visualisation de contre-exemple n'a été affichée sur le P&ID.

Pour évaluer notre outil de visualisation des erreurs, nous avons provoqué une erreur sur le P&ID en supprimant la vanne V2VM01, qui est en amont de la soute St1. Lors de la vérification formelle de ce P&ID, un contre-exemple est généré par l'outil Alloy. Ce contre-exemple est ensuite visualisé sur le P&ID comme le montre la figure 6.10. On voit bien que les composants St1 et TRCH sont décorés par des croix rouges illustrant le fait qu'ils sont mal isolés. La soute St1 n'est pas isolée en amont par une vanne et le composant TRCH n'est pas isolé en aval (connexion avec St1) par une vanne.

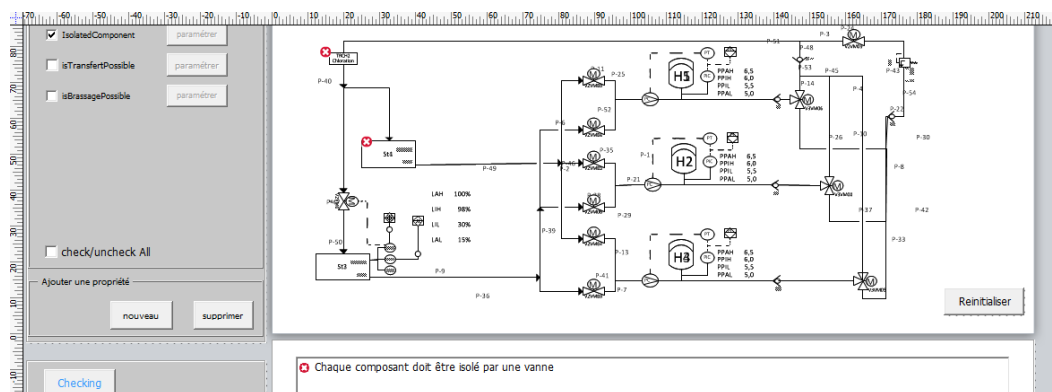


FIGURE 6.10 – Visualisation du contre-exemple

6.3.6 Validité

Pour discuter de la validité de notre étude, nous reprenons la classification proposée par Runeson et Höst [2009].

La validité du construit reflète le fait que l'expérimentation ou l'étude réalisée doit traiter les problèmes identifiés. Dans notre étude, cette menace peut concerner l'interprétation des questions des entretiens par les experts. Pour réduire cette menace, nous avons travaillé sur la terminologie technique avec les experts qui ont participé aux entretiens.

La validité interne concerne la définition des relations causales dans les expérimentations Runeson et Höst [2009]. Notre étude est exploratoire, donc moins sensible à ce type de menace. Une autre menace potentielle concerne le nombre réduit (cinq) d'experts participants aux entretiens. Pour contrer cette menace, nous avons utilisé des entretiens semi-directifs pour extraire le maximum de données et les compléter par des données archivées constituées de documentation et de normes.

La validité externe se réfère à la généralisation des résultats de l'étude. Nous avons proposé un style architectural basé sur la norme ANSI/ISA-5.1. Par conséquent, il peut être utilisé pour vérifier tout P&ID basé sur cette norme. Une autre menace potentielle est que les entretiens et l'extraction de données ont été réalisés par les mêmes chercheurs. Pour réduire le risque de biais, les questions de l'entretien ont été examinées et corrigées par un expert indépendant. La transcription et l'extraction des données ont également été examinées par ce même expert.

La fiabilité aborde la possibilité que d'autres chercheurs puissent reproduire l'étude. Pour faciliter cette réplication, nous avons transcrit les entretiens et avons placé la documentation complète du projet sur le réseau de l'entreprise.

6.3.7 Bilan de la vérification formelle du P&ID du système EdS

Nous avons appliqué notre approche de vérification formelle des diagrammes P&ID sur le système d'eau douce sanitaire, considéré comme la moitié d'un système auxiliaire complexe d'un navire. Le P&ID de ce système comporte 34 instances et 59 connexions (tableau 6.2). Ce P&ID est transformé automatiquement en un module Alloy assez volumineux (tableau 6.2).

	P&ID	Style A	EdS Alloy	Total
Instances	34	-	-	34
Connexions	59	-	-	59
Objets (instances+ connexions)	93	-	-	93
Signatures	-	46	176	222
Prédicats (Pred)	-	7	0	7
Fonctions (Fun)	-	6	0	6
Faits (Fact)	-	1	0	1
Assertions (Assert)	-	0	1	1
Commande (Check)	-	0	1	1
Paragraphes	-	60	178	238
Temps de Génération automatique	-	-	8 secondes	8 secondes
Temps de vérification	-	-	-	1,13 minutes

Tableau 6.2 – Récapitulatif de la vérification formelle du P&ID du système EdS

Afin de définir les exigences que ce P&ID doit satisfaire, nous avons procédé à une étude exploratoire par plusieurs méthodes d'extraction de données. Nous avons ensuite spécifié ces exigences en fonctions, prédicats et assertions Alloy. La vérification formelle par l'analyseur Alloy du P&ID du système EdS a montré que ce P&ID est complet, cohérent, compatible avec le style architectural de la norme ANSI/ISA-5.1 et aussi qu'il respecte ses exigences fonctionnelles et non-fonctionnelles.

La génération automatique des modèles Alloy à partir des diagrammes P&ID offre, d'une part, la possibilité d'avoir des modèles formels sans manipulation directe des notations mathématiques et, d'autre part, un gain de temps et une garantie sur la qualité des diagrammes P&ID utilisés pour la génération des applicatifs. En effet, la génération des modèles Alloy pour le système EdS a été réalisée en 8 secondes (tableau 6.2) et la vérification formelle a pris 1.13 minutes pour un modèle contenant 238 paragraphes.

Nous rappelons que l'approche proposée pour la vérification formelle des diagrammes P&ID permet aussi une visualisation graphique compréhensible par les concepteurs des contre-exemples retournés par l'outil Alloy.

6.4 Conclusion

L'approche de la vérification formelle des composants standards a été appliquée sur un composant concret des systèmes de gestion de fluide qui est la vanne deux voies motorisée. A partir des vues de commande et des vue de supervision de cette vanne, nous avons généré automatiquement des automates temporisés. Ces derniers sont couplés avec des automates temporisés, qui modélisent l'environnement composé de l'opérateur et du composant physique. Le comportement de l'opérateur a été décrit par un modèle de tâche, ensuite il a été transformé en automates temporisés. Sur ces automates, des propriétés CTL ont été vérifiées. Les résultats montrent la robustesse des méthodes formelles dans la détection des erreurs de conception.

Pour les 29 composants standards de la bibliothèque Anaxagore, la génération automatique des modèles formels a permis l'obtention de 17 modèles UPPAAL, modélisant les vues de commande et, 22 modèles UPAAAL modélisant les vues de supervision (voir tableau 6.3). l'ensemble des 39 modèles UPPAAL a été obtenu en 11 secondes (tableau 6.3), ce qui est un gain de temps considérable.

	Vue de commande	Vue de supervision	Total
Modèle UPPAAL	17	22	39
Temps de génération	5 secondes	6 secondes	11 secondes

Tableau 6.3 – Temps de génération des modèles UPPAAL pour les composants de la bibliothèque Anaxagore

Par ailleurs, la vérification formelle de tous les composants standards est en cours. Les opérations manuelles telles que l'élaboration des modèles de tâches, la spécification des pro-

priétés CTL, la modélisation du comportement des composants physiques sont envisagées très prochainement.

Au niveau des diagrammes P&ID, les résultats obtenus restent très encourageants. Dans un futur proche, des expérimentations avec des utilisateurs et des validations sur plusieurs P&ID sont prévues.

7

Conclusion

Résumé : Dans ce dernier chapitre, nous rappelons et résumons les principales contributions de cette thèse qui portent essentiellement sur l'introduction de la vérification formelle dans une démarche de conception mixte. Nous complétons le chapitre par un ensemble de perspectives et d'ouvertures scientifiques qui émergent de ce travail de thèse.

Sommaire

7.1 Introduction	169
7.2 Rappel des contributions	170
7.2.1 Intégration de la vérification formelle dans une démarche de conception	170
7.2.2 Aide à l'obtention des modèles formels	171
7.2.3 Application à un cas d'étude concret	172
7.3 Perspectives	172
7.3.1 Extensions des flots proposés	172
7.3.2 Aide à la spécification des exigences	173
7.3.3 Vérification formelle du système global	174

7.1 Introduction

Plusieurs approches et démarches ont été proposées dans la littérature pour maîtriser la conception des systèmes complexes. Bien qu'offrant déjà une organisation des différentes étapes de réalisation des systèmes complexes, le bon fonctionnement de ces systèmes n'est que partiellement garanti. En effet, dans le monde industriel, les phases de vérification sont réalisées par des tests empiriques qui, en plus d'être réalisés assez tard (le système est souvent déjà implémenté) sont loin d'être exhaustifs. Cette pratique augmente les risques d'erreurs et donc de dysfonctionnements, qui ont un impact non négligeable sur les coûts matériels et parfois humains.

Afin de résoudre ces problèmes et de garantir la qualité des systèmes de contrôle-commande, nous proposons deux approches permettant l'intégration des boucles de vérification dans une

démarche de conception mixte. Ces vérifications sont basées sur des méthodes formelles qui, d'une part, sont exhaustives et d'autre part, contrairement aux tests, peuvent être appliquées dès les premières phases de conception, permettant ainsi de détecter au plus tôt les erreurs.

7.2 Rappel des contributions

Nous avons proposé et présenté dans cette thèse, deux approches pour l'intégration de la vérification formelle dans une démarche de conception mixte dans le domaine du contrôle-commande. La première approche propose la vérification d'une chaîne de contrôle-commande élémentaire. La deuxième propose la vérification d'un modèle d'architecture construit manuellement qui sert de point d'entrée à la génération du contrôle-commande d'un système complexe. Chaque approche est implémentée sous forme d'un flot de vérification comprenant plusieurs opérations automatiques et manuelles. La validité de ces deux approches a été évaluée au niveau de leur application à des cas industriels concrets. Ces points sont abordés plus en détails dans les sections suivantes.

7.2.1 Intégration de la vérification formelle dans une démarche de conception

Pour l'intégration de la vérification formelle dans une démarche mixte, basée sur une bibliothèque de composants standards et un modèle métier, il paraît logique de vérifier la bibliothèque de composants standards et le modèle métier. Notre première contribution, dans cette thèse, est la définition des propriétés à vérifier sur chaque entité. Pour cela, un état de l'art a été réalisé par une méthode cadrée et organisée, basée sur les techniques de la *Systematic Mapping*. Après l'application d'un protocole bien défini, nous avons retenu 81 articles pertinents pour nos recherches. L'étude et l'extraction de données à partir de ces articles, nous a permis d'établir un état de l'art sur la vérification formelle, les langages de spécification et aussi les propriétés généralement vérifiées formellement. Cette étude nous a aussi permis de proposer une nouvelle classification des langages formels. Elle a aussi guidé nos choix de méthodes, de langages formels pour les approches proposées.

La vérification formelle des composants standards concerne **la vérification formelle d'une chaîne de contrôle-commande élémentaire complète**. Nous rappelons qu'un composant de contrôle-commande est, dans notre cas, composé d'un programme de commande écrit dans l'un des cinq langages de la norme IEC 61131-3 et aussi d'une IHM de supervision. Cette approche exploite la technique du *Model-Checking* pour vérifier conjointement le programme de commande et l'interface de supervision. Pour cela, nous avons modélisé en automates temporisés le programme de commande et l'interface de supervision. L'environnement, composé de l'utilisateur et du composant physique, a été aussi modélisé en automates temporisés. En effet, le comportement de l'utilisateur a été formalisé sous la forme d'un modèle de tâches, à partir duquel, nous avons généré des automates temporisés. Sur les automates temporisés modélisant le comportement de l'utilisateur, l'IHM de supervision, le programme de commande et le com-

portement physique du composant, nous avons vérifié des propriétés de sûreté et de vivacité par *Model-Checking* sous UPPAAL. Si la propriété n'est pas vérifiée, un contre-exemple est alors retourné par le *Model-Checker*. Nous avons utilisé cette fonctionnalité pour analyser et corriger les programmes de contrôle-commande de la bibliothèque de l'outil Anaxagore.

En complément de l'approche de vérification des composants standards, nous avons proposé une approche pour la vérification des modèles de conception, utilisés dans une approche mixte pour générer des modèles plus détaillés, voire même des codes exécutables. Les modèles de conception expriment souvent l'architecture du système à développer selon un certain point de vue (physique, comportemental, structurel). L'architecture des systèmes de gestion de fluide est généralement modélisée par un diagramme P&ID basé sur la norme ANSI/ISA-5.1. Pour la **vérification formelle des diagrammes P&ID**, une approche formelle basée aussi sur le *Model-Checking* a été proposée. Dans une première étape, nous nous sommes inspirés des travaux sur la vérification des architectures logicielles pour définir un **style architectural en Alloy pour la norme ANSI/ISA-5.1**. Ce style permet de définir les composants et les connecteurs utilisés pour la description des architectures tout en respectant un ensemble de contraintes qui caractérisent le style. Ensuite, une formalisation en Alloy des diagrammes P&ID a été proposée. Des propriétés de complétude, de cohérence, de compatibilité du P&ID avec le style ainsi que le respect des exigences des cahiers des charges, sont écrites en Alloy et vérifiées par l'analyseur Alloy. Si un contre-exemple est retourné, nous avons proposé une approche et un outil pour **assister les concepteurs dans la compréhension et la détection des erreurs**. Ainsi nous facilitons l'intégration de la vérification formelle aux concepteurs du monde industriel. Cependant, nos approches de vérification formelle proposées s'appuient sur des modèles formels qui sont difficiles à obtenir.

7.2.2 Aide à l'obtention des modèles formels

Pour faciliter l'introduction de la vérification formelle dans un contexte industriel, nous avons proposé **deux flots semi-automatisés** permettant la génération automatique des modèles formels. Les concepts de l'IDM ont été utilisés pour l'implémentation des opérations automatiques sous forme de transformations de modèles.

Au niveau de la bibliothèque, nous avons proposé des transformations de modèles pour générer des automates temporisés à partir des programmes LD, des modèles de tâches HAMSTERS et des interfaces de supervision implémentées dans l'outil Panorama E2. Pour chaque transformation, des méta-modèles et aussi des règles de transformation ont été définies.

Au niveau des diagrammes P&ID, des transformations de modèles permettant la transformation des diagrammes P&ID en modèles Alloy ont été proposées. Les modèles Alloy générés sont combinés avec le style architectural défini initialement pour conduire la vérification formelle. Dans le cas où l'outil de vérification retourne un contre-exemple témoignant d'une violation de propriété, une opération automatique de notre flot de vérification comprenant trois couches de traitements, permet une visualisation graphique de l'erreur au niveau du P&ID. Cet

outil permet d'assister les concepteurs dans la visualisation graphique, la compréhension, la détection et la correction des erreurs de conception.

7.2.3 Application à des cas d'étude concrets

Nos approches outillées ont été **appliquées à des cas industriels concrets**. Nous avons appliqué notre approche de vérification de composants standards sur la vanne deux voies motorisée, utilisée dans la conception des systèmes de gestion de fluide. Notre approche a permis la génération automatique des modèles formels correspondant au programme de commande et à l'IHM de supervision de la vanne. La vérification formelle a révélé plusieurs erreurs de conception, auxquelles nous avons proposées des corrections. Les résultats de cette validation sont encourageants et permettent de confirmer la capacité des méthodes formelles à détecter des erreurs de conception.

L'approche de vérification des diagrammes P&ID a été appliquée sur un système industriel complexe. Ce dernier est un système embarqué dans les navires qui a en charge la production, la distribution et le stockage de l'eau douce. Le P&ID de ce système a été saisi avec l'outil Visio, et ensuite transformé automatiquement en modèle Alloy. Nous avons conduit aussi des entretiens semi-directifs avec des experts métier pour définir les propriétés à vérifier sur le P&ID. Les résultats de la vérification montrent, d'une part, que le P&ID du système EdS intègre et respecte les exigences définies et, d'autre part, l'applicabilité de notre approche sur des cas industriels concrets. Le style architectural défini pour la norme ANSI/ISA-5.1 peut être utilisé pour la vérification de tous les diagrammes P&ID conformement à cette norme.

7.3 Perspectives

Des perspectives et ouvertures ont émergé à l'issue de cette thèse. Ces perspectives portent sur l'extension de nos deux flots de vérification formelle par des opérations automatiques qui remplaceront les opérations manuelles. Une deuxième ouverture importante de ce travail de thèse est la vérification formelle du système global généré par une approche mixte. L'un des problèmes d'une telle perspective est de faire face à l'explosion combinatoire induite par la taille des modèles du système global. Dans cette thèse, les choix de travailler sur les composants standards qui composent le système et le modèle P&ID ont été guidés par la conscience que de tels modèles doivent être formellement vérifiés pour permettre d'avoir des systèmes corrects. Cette vérification est un verrou très difficile à traiter à l'échelle industrielle.

7.3.1 Extensions des flots proposés

Au niveau du flot de vérification formelle des composants standards, deux extensions sont possibles. Elles permettront de visualiser les contre-exemples UPPAAL sous une forme accessible aux concepteurs et de normaliser la génération des automates temporisés à partir des codes des IHM de supervision industrielles.

D'une part, la **visualisation des contre-exemples UPPAAL** sur les programmes de commande et les interfaces de supervision facilitera la compréhension et la détection des erreurs. Une approche similaire a été proposée et intégrée dans le flot de vérification des diagrammes P&ID. Cependant, à la différence des diagrammes P&ID qui sont statiques, les programmes de commande et les IHM de supervision, eux, sont dynamiques. La visualisation des contre-exemples, doit tenir compte de ce critère très important. De plus, de nombreuses variables entrent en jeu dans les interfaces de supervision et les programmes de commande. Il sera question alors d'étudier les besoins informationnels des concepteurs pour sélectionner les données les plus pertinentes à afficher et la manière de les afficher aux concepteurs.

D'autre part, la **normalisation** de la transformation des IHM de supervision implémentées sous le logiciel Panorama E2, en automates temporisés peut être envisagée. Cette transformation reste dépendante de cet outil. Il sera donc préférable d'utiliser un modèle intermédiaire indépendant de toutes plateformes et qui pourra engendrer n'importe quelle IHM de supervision, afin d'étendre notre approche à d'autres IHM saisies dans d'autres outils.

Enfin, des expérimentations peuvent être envisagées pour évaluer **l'utilité et l'utilisabilité** des outils proposés.

7.3.2 Aide à la spécification des exigences

Les flots de vérification proposés dans cette thèse, offrent la possibilité aux concepteurs de vérifier un certain nombre de propriétés spécifiées manuellement. La **spécification** formelle des propriétés nécessite des connaissances en langages formels. La mise à disposition de moyens pour assister et guider le concepteur métier lors de la phase de spécification des exigences s'avère donc indispensable.

Une solution consiste à écrire les exigences dans un langage structuré et proche du langage naturel (ex. le français ou l'anglais). Afin d'y parvenir, une démarche pour la spécification de propriétés sous forme de patterns peut être envisagée. Un pattern offre une solution réutilisable à un problème récurrent donné, dans un contexte défini [Arnaud 2008].

La spécification par **pattern** permet de tirer les avantages des langages formels et des langages naturels. Ces patterns sont proposés à l'utilisateur "*non spécialiste des méthodes formelles*" en langage naturel réduit. Cela permet à ce dernier de spécifier les exigences sans manipuler les formules mathématiques complexes des langages formels. Les exigences saisies sont transformées en langage formel pour être vérifiées. En résumé, l'utilisation des patterns pour la spécification des exigences a les avantages de [Palomares et al. 2014] : 1) garantir la conformité des exigences ; 2) garantir leur complétude ; 3) limiter les ambiguïtés et 4) réduire le temps de spécification.

Pour l'intégration d'une méthode de spécification assistée pour la spécification des exigences, en utilisant les patterns, il faudrait **définir les patterns** qui englobent toutes les propriétés vérifiables sur le modèle. Une fois la liste des patterns définie, il faudra proposer une **grammaire** en langage naturel qui permet de définir la forme textuelle de chaque pattern (mots-

clés, paramètres). Ensuite, pour chaque pattern, il faudrait générer une formulation en langage formel à partir d'une formulation textuelle.

7.3.3 Vérification formelle du système global

Nous assurons, avec les deux approches de vérification formelle proposées dans cette thèse, que les applicatifs générés ne contiennent pas d'erreurs qui proviennent de la bibliothèque et du P&ID. Cependant, ces applicatifs peuvent contenir d'autres types d'erreurs. Ces erreurs peuvent provenir des transformations de modèles, dans le cas d'une génération automatique, ou bien d'erreurs de spécification fonctionnelle.

Pour garantir la qualité des applicatifs générés, plusieurs solutions sont envisageables. Les techniques de vérification formelle peuvent aussi être utilisées à ce niveau de conception. Cependant, les applicatifs sont généralement des modèles très détaillés. Pour cette raison, la vérification formelle par *Model-Checking* peut s'avérer impossible face au problème d'explosion combinatoire. Un moyen de résoudre ce problème pourrait être l'utilisation des techniques basées sur l'abstraction ou les contrats [McMillan 1999, Quinton et Graf 2008, Cofer et al. 2012]. L'utilisation de la **simulation** [Prat et al. 2015; 2016] couplée avec la vérification formelle peut être également une solution pour vérifier le système global en évitant le problème de l'explosion combinatoire.

Pour la vérification du système complet, la **spécification des exigences fonctionnelles** est indispensable. Une obtention automatique de ces exigences permettrait de faciliter l'intégration de la vérification formelle au niveau système. Les techniques de programmation par démonstration semblent faciliter l'obtention de ces spécifications automatiquement [Goubali et al. 2016].

Les approches proposées dans cette thèse permettent d'assister les experts informaticiens, automaticiens et mécaniciens dans la conception des systèmes sociotechniques. Ces derniers sont connus par leur pluridisciplinarité et leur forte interaction avec l'humain. L'extension de nos approches à d'**autres modèles métier** permettant la prise en charge des critères psychologiques, comme l'analyse du travail cognitif (**CWA** pour Cognitive Work Analysis en anglais) [Vicente 1999] et les **interfaces écologiques** [Rechard 2015], serait une ouverture très intéressante.

Bibliographie

- [AlloyAnalyzer] : *The Alloy analyzer*. – URL <http://alloy.mit.edu/>
- [Kmad] : *KMAD*. – URL <http://lisi-forge.ensma.fr/forge/projects/kmade/>
- [Start] : *The StArt tool*. – URL http://lapes.dc.ufscar.br/tools/start_tool
- [IEEE-1233 1998] IEEE Guide for Developing System Requirements Specifications. In : *IEEE Std 1233, 1998 Edition* (1998), Dec, p. 1–36
- [Aboussoror 2013] ABOUSSOROR, El A. : *Méthodes de diagnostic avancées dans la validation formelle des modèles*, Université de Toulouse, Université Toulouse III-Paul Sabatier, Ph.D. thesis, 2013
- [Abowd et al. 1995] ABOWD, Gregory D. ; WANG, Hung-Ming ; MONK, Andrew F. : A formal technique for automated dialogue development. In : *Proceedings of the 1st conference on Designing interactive systems : processes, practices, methods, & techniques* ACM (event), 1995, p. 219–226
- [Abrial 2010] ABRIAL, Jean-Raymond : *Modeling in Event-B : system and software engineering*. Cambridge University Press, 2010
- [Abrial et Abrial 2005] ABRIAL, Jean-Raymond ; ABRIAL, Jean-Raymond : *The B-book : assigning programs to meanings*. Cambridge University Press, 2005
- [Aït-Ameur et al. 2005] AÏT-AMEUR, Y ; AÏT-SADOUNE, I ; BARON, M : Modélisation et Validation formelles d’IHM : LOT 1 (LISI/ENSMA). In : *Délivrable pour le projet RNRT-VERBATIM* (2005), p. 73
- [Aït-Ameur et Baron 2006] AÏT-AMEUR, Yamine ; BARON, Mickael : Formal and experimental validation approaches in HCI systems design based on a shared event B model. In : *International Journal on Software Tools for Technology Transfer* 8 (2006), Nr. 6, p. 547–563. – ISSN 1433-2779
- [Aït-Ameur et al. 1999] AÏT-AMEUR, Yamine ; GIRARD, Patrick ; JAMBON, Francis : Using the B formal approach for incremental specification design of interactive systems. In : *Engineering for Human-Computer Interaction*. Springer, 1999, p. 91–109
- [Alagar et Periyasamy 2011] ALAGAR, Vangalur S. ; PERIYASAMY, Kasilingam : *Specification of software systems*. Springer Science & Business Media, 2011
- [Alanche et al. 1986] ALANCHE, P ; LHOSTE, P ; MOREL, G ; ROESCH, M ; SALIM, M ; SALVI, Ph : Application de la modélisation de la Partie opérative à la structuration de la commande. In : *Journées AFCET, Montpellier, France* (1986)

- [Alavizaedh et al. 2007] ALAVIZAEDH, S F. ; NEKOO, Alireza H. et al. : ReUML : A UML profile for modeling and verification of reactive systems. In : *International Conference on Software Engineering Advances (ICSEA 2007)* IEEE (event), 2007, p. 50–50
- [Alemán et Álvarez 2000] ALEMÁN, José Luis F. ; ÁLVAREZ, Ambrosio T. : Can intuition become rigorous ? Foundations for UML model verification tools. In : *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on* IEEE (event), 2000, p. 344–355
- [Alenljung et Lennartson 2009] ALENLJUNG, Tord ; LENNARTSON, Bengt : Formal verification of PLC controlled systems using sensor graphs. In : *IEEE International Conference on Automation Science and Engineering* IEEE (event), 2009, p. 164–170
- [Alford 1978] ALFORD, Mack W. : Software Requirements Engineering Methodology (SREM) at the age of two. In : *Computer Software and Applications Conference, 1978. COMPSAC'78. The IEEE Computer Society's Second International* IEEE (event), 1978, p. 332–339
- [Allen 1997] ALLEN, Robert : *A Formal Approach to Software Architecture*, Carnegie Mellon, School of Computer Science, Ph.D. thesis, January 1997. – Issued as CMU Technical Report CMU-CS-97-144.
- [Almeida et al. 2011] ALMEIDA, José B. ; FRADE, Maria J. ; PINTO, Jorge S. ; SOUSA, Simao M. de : An overview of formal methods tools and techniques. In : *Rigorous Software Development*. Springer, 2011, p. 15–44
- [Alspaugh et al. 1992] ALSPAUGH, Thomas A. ; FAULK, Stuart R. ; BRITTON, Kathryn H. ; PARKER, R A. ; PARNAS, David L. : Software Requirements for the A-7E Aircraft. / DTIC Document. 1992. – Research Report
- [Alur et Dill 1990] ALUR, Rajeev ; DILL, David : Automata for modeling real-time systems. In : *Automata, languages and programming*. Springer, 1990, p. 322–335
- [Amorim et al. 2005] AMORIM, Leonardo ; MACIEL, P ; NOGUEIRA, M ; BARRETO, R ; TAVARES, Eduardo : A methodology for mapping live sequence chart to coloured Petri net. In : *2005 IEEE International Conference on Systems, Man and Cybernetics Volume 4* IEEE (event), 2005, p. 2999–3004
- [Apvrille et al. 2004] APVRILLE, Ludovic ; COURTIAT, J-P ; LOHR, Christophe ; SAQUISANNES, Pierre de : TURTLE : A real-time UML profile supported by a formal validation toolkit. In : *IEEE transactions on Software Engineering* 30 (2004), Nr. 7, p. 473–487
- [Archer et Heitmeyer 1997] ARCHER, Myla ; HEITMEYER, Constance : Verifying hybrid systems modeled as timed automata : A case study. In : *International Workshop on Hybrid and Real-Time Systems* Springer (event), 1997, p. 171–185

- [Aredo et Owe 2005] AREDO, Demissie B. ; OWE, Olaf : Model-based verification in the development of dependable systems. In : *International Conference on Information Technology : Coding and Computing (ITCC'05)-Volume II* Volume 2 IEEE (event), 2005, p. 327–334
- [Arnaud 2008] ARNAUD, Nicolas : *Fiabiliser la réutilisation des patrons par une approche orientée complétude, variabilité et généricité des spécifications*, Université Joseph-Fourier-Grenoble I, Ph.D. thesis, 2008
- [Astesiano et al. 2002] ASTESIANO, Egidio ; BIDOIT, Michel ; KIRCHNER, Hélène ; KRIEGBRÜCKNER, Bernd ; MOSSES, Peter D. ; SANNELLA, Donald ; TARLECKI, Andrzej : CASL : the common algebraic specification language. In : *Theoretical Computer Science* 286 (2002), Nr. 2, p. 153–196
- [Astesiano et Wirsing 1987] ASTESIANO, Egidio ; WIRSING, Martin : An introduction ASL. In : *The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation* North-Holland Publishing Co. (event), 1987, p. 343–365
- [Baier et al. 2008] BAIER, Christel ; KATOEN, Joost-Pieter ; LARSEN, Kim G. : *Principles of model checking*. MIT press, 2008
- [Bajovs et al. 2013] BAJOVŠ, Andrejs ; NIKIFOROVA, Oksana ; SEJANS, Janis : Code Generation from UML Model : State of the Art and Practical Implications. In : *Applied Computer Systems* 14 (2013), Nr. 1, p. 9–18
- [Bajwa 2014] BAJWA, Imran S. : *A natural language processing approach to generate SBVR and OCL*, University of Birmingham, Ph.D. thesis, 2014
- [Bajwa et al. 2010] BAJWA, Imran S. ; BORDBAR, Behzad ; LEE, Mark G. : OCL constraints generation from natural language specification. In : *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International* IEEE (event), 2010, p. 204–213
- [Bajwa et al. 2012] BAJWA, Imran S. ; BORDBAR, Behzad ; LEE, Mark G. ; ANASTASAKIS, Kyriakos : NL2 Alloy : A Tool to Generate Alloy from NL Constraints. In : *JDIM* 10 (2012), Nr. 6, p. 365–372
- [Barboni et al. 2010] BARBONI, Eric ; LADRY, Jean-François ; NAVARRE, David ; PALANQUE, Philippe ; WINCKLER, Marco : Beyond modelling : an integrated environment supporting co-execution of tasks and systems models. In : *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems* ACM (event), 2010, p. 165–174
- [Barroca et McDermid 1992] BARROCA, Leonor M. ; MCDERMID, John A. : Formal methods : Use and relevance for the development of safety-critical systems. In : *The Computer Journal* 35 (1992), Nr. 6, p. 579–599

- [Basit-Ur-Rahim et al. 2014] BASIT-UR-RAHIM, Muhammad A. ; ARIF, Fahim ; AHMAD, Jamil : Formal verification of sequence diagram using DiVinE. In : *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on IEEE* (event), 2014, p. 1–6
- [Bastias et al. 2011] BASTIAS, Alberto ; BIHARY, Sidharth ; ROY, Suman : An automated analysis of errors for BPM processes modeled using an in-house Infosys tool. In : *2011 18th Asia-Pacific Software Engineering Conference IEEE* (event), 2011, p. 97–105
- [Behrmann et al. 2004] BEHRMANN, Gerd ; DAVID, Alexandre ; LARSEN, Kim G. : A tutorial on uppaal. In : *Formal methods for the design of real-time systems*. Springer, 2004, p. 200–236
- [Beizer 2003] BEIZER, Boris : *Software testing techniques*. Dreamtech Press, 2003
- [Benavides et al. 2010] BENAVIDES, David ; SEGURA, Sergio ; RUIZ-CORTÉS, Antonio : Automated analysis of feature models 20 years later : A literature review. In : *Information Systems* 35 (2010), Nr. 6, p. 615–636
- [Bender et al. 2008] BENDER, Darlam F. ; COMBEMALE, Benoît ; CRÉGUT, Xavier ; FARINES, Jean M. ; BERTHOMIEU, Bernard ; VERNADAT, François : Ladder metamodeling and PLC program validation through time Petri nets. In : *Model Driven Architecture–Foundations and Applications Springer* (event), 2008, p. 121–136
- [Bennion et Habli 2014] BENNION, Matthew ; HABLI, Ibrahim : A candid industrial evaluation of formal software verification using model checking. In : *Companion Proceedings of the 36th International Conference on Software Engineering ACM* (event), 2014, p. 175–184
- [Berry et Gonthier 1992] BERRY, Gérard ; GONTHIER, Georges : The Esterel synchronous programming language : Design, semantics, implementation. In : *Science of computer programming* 19 (1992), Nr. 2, p. 87–152
- [Bertot et Castéran 2013] BERTOT, Yves ; CASTÉРАН, Pierre : *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013
- [Bévan 2013] BÉVAN, Romain : *Approche composant pour la commande multi-versions des systèmes transistiques reconfigurables*, Lorient, Ph.D. thesis, 2013
- [Bézivin et Blanc 2002] BÉZIVIN, Jean ; BLANC, Xavier : MDA : Vers un important changement de paradigme en génie logiciel. In : *Développeur référence v2* 16 (2002), p. 15
- [Bézivin et Gerbé 2001] BÉZIVIN, Jean ; GERBÉ, Olivier : Towards a Precise Definition of the OMG/MDA Framework. In : *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2001 (ASE '01), p. 273–. – URL <http://dl.acm.org/citation.cfm?id=872023.872565>

- [Bignon 2012] BIGNON, Alain : *Génération conjointe de commandes et d'interfaces de supervision pour systèmes sociotechniques reconfigurables*, Université de Bretagne Sud, Ph.D. thesis, 2012
- [Bignon et al. 2010] BIGNON, Alain ; BERRUET, Pascal ; ROSSI, André : Joint generation of controls and interfaces for sociotechnical and reconfigurable systems. In : *IEEE International Conference on Systems Man and Cybernetics (SMC)* IEEE (event), 2010, p. 749–755
- [Bignon et al. 2013] BIGNON, Alain ; ROSSI, André ; BERRUET, Pascal : An integrated design flow for the joint generation of control and interfaces from a business model. In : *Computers in industry* 64 (2013), Nr. 6, p. 634–649
- [Bjørner et Havelund 2014] BJØRNER, Dines ; HAVELUND, Klaus : *40 Years of Formal Methods*. p. 42–61. In : JONES, Cliff (Editor) ; PIHLAJASAARI, Pekka (Editor) ; SUN, Jun (Editor) : *FM 2014 : Formal Methods : 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Cham : Springer International Publishing, 2014
- [Bloom et al. 1996] BLOOM, Bard ; CHENG, Allan ; DZOUZA, A : Verifying SOS specifications. In : *Proceedings of the Eleventh Annual Conference on Computer Assurance, COMPASS'96, Systems Integrity. Software Safety. Process Security*. IEEE (event), 1996, p. 117–127
- [Bobot et al. 2011] BOBOT, François ; FILLIÂTRE, Jean-Christophe ; MARCHÉ, Claude ; PASKEVICH, Andrei : Why3 : Shepherd your herd of provers. In : *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, 2011, p. 53–64
- [Boehm et al. 1981] BOEHM, Barry W. et al. : *Software engineering economics*. Volume 197. Prentice-hall Englewood Cliffs (NJ), 1981
- [Bolognesi et Brinksma 1987] BOLOGNESI, Tommaso ; BRINKSMA, Ed : Introduction to the ISO specification language LOTOS. In : *Computer Networks and ISDN systems* 14 (1987), Nr. 1, p. 25–59
- [Bolton et al. 2013] BOLTON, Matthew L. ; BASS, Ellen J. ; SIMINICEANU, Radu I. : Using formal verification to evaluate human-automation interaction : A review. In : *Systems, Man, and Cybernetics : Systems, IEEE Transactions on* 43 (2013), Nr. 3, p. 488–503
- [Bolton et al. 2011] BOLTON, Matthew L. ; SIMINICEANU, Radu ; BASS, Ellen J. et al. : A systematic approach to model checking human–automation interaction using task analytic models. In : *Systems, Man and Cybernetics, Part A : Systems and Humans, IEEE Transactions on* 41 (2011), Nr. 5, p. 961–976
- [Bonfè et Fantuzzi 2004] BONFE, Marcello ; FANTUZZI, Cesare : A practical approach to object-oriented modeling of logic control systems for industrial applications. In : *43rd IEEE Conference on Decision and Control, CDC*. Volume 1 IEEE (event), 2004, p. 980–985

- [Bowen et Hinchey 1995] BOWEN, Jonathan P. ; HINCHEY, Michael G. : Ten commandments of formal methods. In : *Computer* 28 (1995), Nr. 4, p. 56–63
- [Bowen et Hinchey 2006] BOWEN, Jonathan P. ; HINCHEY, Michael G. : Ten commandments of formal methods... ten years later. In : *Computer* 39 (2006), Nr. 1, p. 40–48
- [Brereton et al. 2007] BRERETON, Pearl ; KITCHENHAM, Barbara A. ; BUDGEN, David ; TURNER, Mark ; KHALIL, Mohamed : Lessons from applying the systematic literature review process within the software engineering domain. In : *Journal of systems and software* 80 (2007), Nr. 4, p. 571–583
- [Brink et al. 1998] BRINK, K. ; BUN, L.J.G. ; KATWIJK, Jan van ; SPELBERG, R.F.L. ; TOETENEL, W.J. : Automatic analysis of embedded systems specified in Astral. In : *Proceedings of the Thirty-First Hawaii International Conference on System Sciences* Volume 3 IEEE (event), 1998, p. 177–186
- [Brunel et al. 2014] BRUNEL, Julien ; RIOUX, Laurent ; PAUL, Stéphane ; FAUCOGNEY, Anthony ; VALLÉE, Frédérique : Formal safety and security assessment of an avionic architecture with alloy. In : *arXiv preprint arXiv :1405.1113* (2014)
- [Bucci et al. 1994] BUCCI, Giacomo ; CAMPANAI, Maurizio ; NESI, Paolo ; TRAVERSI, Marcello : An object-oriented dual language for specifying reactive systems. In : *Proceedings of the First International Conference on Requirements Engineering, 1994*. IEEE (event), 1994, p. 6–15
- [Cao et al. 2011] CAO, Yan ; XU, Tianhua ; TANG, Tao ; WANG, Haifeng ; ZHAO, Lin : Automatic generation and verification of interlocking tables based on domain specific language for computer based interlocking systems (DSL-CBI). In : *IEEE International Conference on Computer Science and Automation Engineering (CSAE)* Volume 2 IEEE (event), 2011, p. 511–515
- [Cavalcante et al. 2014] CAVALCANTE, Everton ; OQUENDO, Flavio ; BATISTA, Thais : Architecture-Based Code Generation : From π -ADL Architecture Descriptions to Implementations in the Go Language. In : *European Conference on Software Architecture* Springer (event), 2014, p. 130–145
- [Cha et Yoo 2012] CHA, Sungdeok ; YOO, Junbeom : A safety-focused verification using software fault trees. In : *Future Generation Computer Systems* 28 (2012), Nr. 8, p. 1272–1282
- [Chang et al. 1994] CHANG, Edward ; MANNA, Zohar ; PNUELI, Amir : Compositional verification of real-time systems. In : *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on* IEEE (event), 1994, p. 458–465

- [Charbonnier 1996] CHARBONNIER, François : *Commande supervisée des systèmes à événements discrets*, Ph.D. thesis, 1996
- [Chechik et Gannon 2001] CHECHIK, Marsha ; GANNON, John : Automatic analysis of consistency between requirements and designs. In : *IEEE Transactions on Software Engineering* 27 (2001), Nr. 7, p. 651–672
- [Chen et al. 2007] CHEN, Chunqing ; DONG, Jin S. ; SUN, Jun : Machine-assisted proof support for validation beyond Simulink. In : *International Conference on Formal Engineering Methods* Springer (event), 2007, p. 96–115
- [Chen et al. 2009] CHEN, Geng ; LUO, Lei ; GONG, Rong ; GUI, Shenglin : Dependability analysis for AADL models by PVS. In : *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing, DASC'09*. IEEE (event), 2009, p. 19–24
- [Chi 2000] CHI, Ed Huai-hsin : A taxonomy of visualization techniques using the data state reference model. In : *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on IEEE* (event), 2000, p. 69–75
- [Cimatti et Tonetta 2012] CIMATTI, Alessandro ; TONETTA, Stefano : A property-based proof system for contract-based design. In : *38th Euromicro Conference on Software Engineering and Advanced Applications* IEEE (event), 2012, p. 21–28
- [Clarke et Emerson 1981] CLARKE, Edmund M. ; EMERSON, E A. : Design and synthesis of synchronization skeletons using branching time temporal logic. In : *Workshop on Logic of Programs* Springer (event), 1981, p. 52–71
- [Clarke et Wing 1996] CLARKE, Edmund M. ; WING, Jeannette M. : Formal methods : State of the art and future directions. In : *ACM Computing Surveys (CSUR)* 28 (1996), Nr. 4, p. 626–643
- [Clavel et al. 2002] CLAVEL, Manuel ; DURÁN, Francisco ; EKER, Steven ; LINCOLN, Patrick ; MARTI-OLIET, Narciso ; MESEGUER, José ; QUESADA, José F : Maude : specification and programming in rewriting logic. In : *Theoretical Computer Science* 285 (2002), Nr. 2, p. 187–243
- [Cofer et al. 2012] COFER, Darren ; GACEK, Andrew ; MILLER, Steven ; WHALEN, Michael W. ; LAVALLEY, Brian ; SHA, Lui : Compositional verification of architectural models. In : *NASA Formal Methods Symposium* Springer (event), 2012, p. 126–140
- [Combemale 2008] COMBEMALE, Benoit : *Approche de métamodélisation pour la simulation et la vérification de modèle—Application à l'ingénierie des procédés*, Institut National Polytechnique de Toulouse-INPT, Ph.D. thesis, 2008

- [Cottet et Grolleau 2005] COTTET, Francis ; GROLLEAU, Emmanuel : *Systèmes temps réel de contrôle-commande : conception et implémentation*. Dunod, 2005
- [Coupât 2014] COUPÂT, Raphaël : *Méthodologie pour les études d'automatisation et la génération automatique de programmes Automates Programmables Industriels sûrs de fonctionnement. Application aux Equipements d'Alimentation des Lignes Électrifiées*, Reims, Ph.D. thesis, 2014
- [Da Silva Oliveira et al. 2011] DA SILVA OLIVEIRA, E.A. ; SILVA, L.D. da ; GORGONIO, K. ; PERKUSICH, A. ; MARTINS, A.F. : Obtaining formal models from Ladder diagrams. In : *9th IEEE International Conference on Industrial Informatics (INDIN)*, July 2011, p. 796–801
- [Dabaghchian et Azgomi 2015] DABAGHCHIAN, Maryam ; AZGOMI, Mohammad A. : Model checking the observational determinism security property using PROMELA and SPIN. In : *Formal Aspects of Computing 27* (2015), Nr. 5-6, p. 789–804
- [Daneels et Salter 1999] DANEELS, Axel ; SALTER, Wayne : What is SCADA. In : *International Conference on Accelerator and Large Experimental Physics Control Systems*, 1999, p. 339–343
- [De Smet et Rossi 2002] DE SMET, Olivier ; ROSSI, Olivier : Verification of a controller for a flexible manufacturing line written in Ladder Diagram via model-checking. In : *American Control Conference, 2002. Proceedings of the 2002 Volume 5 IEEE (event)*, 2002, p. 4147–4152
- [Debruyne et al. 2005] DEBRUYNE, Vincent ; SIMONOT-LION, Françoise ; TRINQUET, Yvon : EASTADL An architecture description language. In : *Architecture Description Languages*. Springer, 2005, p. 181–195
- [Dhaussy et al. 2009] DHAUSSY, Philippe ; PILLAIN, Pierre-Yves ; CREFF, Stephen ; RAJI, Amine ; LE TRAON, Yves ; BAUDRY, Benoit : Evaluating context descriptions and property definition patterns for software formal validation. In : *International Conference on Model Driven Engineering Languages and Systems* Springer (event), 2009, p. 438–452
- [Diaconescu et Futatsugi 1998] DIACONESCU, Razvan ; FUTATSUGI, Kokichi : CafeOBJ report : The language. In : *Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification 6* (1998)
- [Dickover et al. 1977] DICKOVER, Melvin E. ; MCGOWAN, Clement L. ; ROSS, Douglas T. : Software design using : SADT. In : *Proceedings of the 1977 annual conference ACM* (event), 1977, p. 125–133
- [Dong et al. 2001] DONG, Wei ; WANG, Ji ; QI, Xuan ; QI, Zhi-Chang : Model checking UML statecharts. In : *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific IEEE (event)*, 2001, p. 363–370

- [Dormoy 2008] DORMOY, Francois-Xavier : Scade 6 : a model based solution for safety critical software development. In : *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, 2008, p. 1–9
- [D'Souza et al. 2014] D'SOUZA, Meenakshi ; RAMESH, S ; SATPATHY, Manoranjan : Architectural semantics of AADL using Event-B. In : *International Conference on Contemporary Computing and Informatics (IC3I) IEEE (event)*, 2014, p. 92–97
- [Du et al. 2000] DU, Xiaoqun ; RAMAKRISHNAN, CR ; SMOLKA, Scott A. : Tabled resolution+ constraints : A recipe for model checking real-time systems. In : *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE IEEE (event)*, 2000, p. 175–184
- [Dumas et al. 2010] DUMAS, Xavier ; BONIOL, Frédéric ; DHAUSSY, Philippe ; BONNAFOUS, Eric : Context modelling and partial-order reduction : Application to SDL industrial embedded systems. In : *International Symposium on Industrial Embedded System (SIES) IEEE (event)*, 2010, p. 197–200
- [Efftinge et Völter 2006] EFFTINGE, Sven ; VÖLTER, Markus : oAW xText : A framework for textual DSLs. In : *Workshop on Modeling Symposium at Eclipse Summit Volume 32*, 2006, p. 118
- [EIA/ANSI-632 1999] EIA/ANSI-632 : *GEIA Standard Processes for Engineering a System EIA-632*. Government Electronics and Information Technology Association, 1999
- [Esteve et al. 2012] ESTEVE, Marie-Aude ; KATOEN, Joost-Pieter ; NGUYEN, Viet Y. ; POSTMA, Bart ; YUSHTEIN, Yuri : Formal correctness, safety, dependability, and performance analysis of a satellite. In : *Proceedings of the 34th International Conference on Software Engineering IEEE Press (event)*, 2012, p. 1022–1031
- [Evans et al. 2004] EVANS, Neil ; TREHARNE, Helen ; LALEAU, Regine ; FRAPPIER, Marc : How to verify dynamic properties of information systems. In : *Proceedings of the Second International Conference on Software Engineering and Formal Methods, SEFM 2004*. IEEE (event), 2004, p. 416–425
- [Falkman et al. 2011] FALKMAN, Petter ; HELANDER, Erik ; ANDERSSON, Mikael : Automatic generation : A way of ensuring PLC and HMI standards. In : *2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA) IEEE (event)*, 2011, p. 1–4
- [Fiorèse et Meinadier 2012] FIORÈSE, Serge ; MEINADIER, Jean-Pierre : *Découvrir et comprendre l'ingénierie système*. CEPADUES Editions, ISBN, 2012
- [Fleischhack et Grahlmann 1998] FLEISCHHACK, Hans ; GRAHLMANN, Bernd : Towards Compositional Verification of SDL Systems. In : *Proceedings of the Thirty-First Hawaii International Conference on System Sciences Volume 7 IEEE (event)*, 1998, p. 404–414

- [Furia et al. 2007] FURIA, Carlo A. ; ROSSI, Matteo ; MANDRIOLI, Dino ; MORZENTI, Angelo : Automated compositional proofs for real-time systems. In : *Theoretical computer science* 376 (2007), Nr. 3, p. 164–184
- [Futatsugi et al. 1985] FUTATSUGI, Kokichi ; GOGUEN, Joseph A. ; JOUANNAUD, Jean-Pierre ; MESEGUER, José : Principles of OBJ2. In : *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* ACM (event), 1985, p. 52–66
- [an Fuxman 2001] FUXMAN, Ariel D. an : *Formal analysis of early requirements specifications*, Citeseer, Ph.D. thesis, 2001
- [Fuxman et al. 2004] FUXMAN, Ariel ; LIU, Lin ; MYLOPOULOS, John ; PISTORE, Marco ; ROVERI, Marco ; TRAVERSO, Paolo : Specifying and analyzing early requirements in Tropos. In : *Requirements Engineering* 9 (2004), Nr. 2, p. 132–150
- [Fuxman et al. 2001] FUXMAN, Ariel ; PISTORE, Marco ; MYLOPOULOS, John ; TRAVERSO, Paolo : Model checking early requirements specifications in Tropos. In : *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on* IEEE (event), 2001, p. 174–181
- [Gammaitoni et Kelsen 2014] GAMMAITONI, Loïc ; KELSEN, Pierre : Domain-specific visualization of alloy instances. In : *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* Springer (event), 2014, p. 324–327
- [Gammaitoni et Kelsen 2015] GAMMAITONI, Loïc ; KELSEN, Pierre : F-alloy : An alloy based model transformation language. In : *International Conference on Theory and Practice of Model Transformations* Springer (event), 2015, p. 166–180
- [Gang et Zhiming 2003] GANG, Xu ; ZHIMING, Wu : A new method for FMS modeling and formal verification. In : *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference Volume 1* IEEE (event), 2003, p. 224–231
- [Gardey et al. 2005] GARDEY, Guillaume ; LIME, Didier ; MAGNIN, Morgan ; ROUX, Olivier (H.). : *Romeo : A Tool for Analyzing Time Petri Nets*. p. 418–423. In : ETESSAMI, Kousha (Editor) ; RAJAMANI, Sriram K. (Editor) : *Computer Aided Verification : 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – URL http://dx.doi.org/10.1007/11513988_41. – ISBN 978-3-540-31686-2
- [Garis et al. 2012] GARIS, Ana ; PAIVA, Ana C. ; CUNHA, Alcino ; RIESCO, Daniel : Specifying UML protocol state machines in Alloy. In : *Integrated Formal Methods* Springer (event), 2012, p. 312–326

- [Garlan et al. 2010] GARLAN, David ; MONROE, Robert ; WILE, David : Acme : an architecture description interchange language. In : *CASCON First Decade High Impact Papers* IBM Corp. (event), 2010, p. 159–173
- [Gaudel 1992] GAUDEL, Marie-Claude : Structuring and Modularizing Algebraic Specifications : the PLUSS specification language, evolutions and perspectives. In : *Annual Symposium on Theoretical Aspects of Computer Science* Springer (event), 1992, p. 1–18
- [George et al. 1992] GEORGE, Chris et al. : *The RAISE specification language*. Prentice-Hall Hemel Hempstead, 1992
- [Gerber et al. 2010] GERBER, Christian ; PREUSSE, Sebastian ; HANISCH, Hans-Michael : A complete framework for controller verification in manufacturing. In : *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)* IEEE (event), 2010, p. 1–9
- [Gerking 2013] GERKING, Christopher : Transparent Uppaal-based verification of MechatronicUML models. In : *Master's thesis, University of Paderborn* (2013)
- [Ghazel et al. 2009] GHAZEL, Mohamed ; MASMOUDI, Malek ; TOGUYENI, Armand : Verification of temporal requirements of complex systems using UML patterns, application to a railway control example. In : *IEEE International Conference on System of Systems Engineering* IEEE (event), 2009, p. 1–6
- [Gnesi et al. 1999] GNESI, Stefania ; LATELLA, Diego ; MASSINK, Mieke : Model checking UML statechart diagrams using JACK. In : *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on* IEEE (event), 1999, p. 46–55
- [Goguen et al. 1987] GOGUEN, Joseph ; KIRCHNER, Claude ; KIRCHNER, H el ene ; M EGRELIS, Aristide ; MESEGUER, Jos e ; WINKLER, Timothy : An introduction to OBJ 3. In : *International Workshop on Conditional Term Rewriting Systems* Springer (event), 1987, p. 258–263
- [Goldson 2002] GOLDSON, Doug : Formal verification of μ -charts. In : *Software Engineering Conference, 2002. Ninth Asia-Pacific* IEEE (event), 2002, p. 129–136
- [G omez-Mart nez et al. 2014] G OMEZ-MART NEZ, Elena ; RODR GUEZ, Ricardo J. ; ELORZA, Leire E. ; REZABAL, Miren I. ; EARLE, Clara B. : Model-based verification of safety contracts. In : *International Conference on Software Engineering and Formal Methods* Springer (event), 2014, p. 101–115
- [Gordon et Melham 1993] GORDON, Michael J. ; MELHAM, Tom F. : *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge New York : Cambridge University Press, 1993. – ISBN 978-0521441896

- [Goubali 2017] GOUBALI, Olga : *Apport Des Techniques De Programmation Par Démonstration Dans Une Démarche De Génération Automatique D'applicatifs De Contrôle-Commande*, L'École Nationale Supérieure de Mécanique et d'Aérotechnique, Ph.D. thesis, 2017
- [Goubali et al. 2014] GOUBALI, Olga ; BIGNON, Alain ; BERRUET, Pascal ; GIRARD, Patrick ; GUITTET, Laurent : Anaxagore, an Example of Model-driven Engineering for Industrial Supervision. In : *Proceedings of the 2014 Ergonomie Et Informatique Avancée Conference - Design, Ergonomie Et IHM : Quelle Articulation Pour La Co-conception De L'Interaction*. New York, NY, USA : ACM, 2014 (Ergo'IA '14), p. 58–65. – URL <http://doi.acm.org/10.1145/2671470.2671478>. – ISBN 978-1-4503-2970-5
- [Goubali et al. 2016] GOUBALI, Olga ; GIRARD, Patrick ; GUITTET, Laurent ; BIGNON, Alain ; KESRAOUI, Djamel ; BERRUET, Pascal ; BOUILLON, Jean-Frédéric : Designing Functional Specifications for Complex Systems. In : *International Conference on Human-Computer Interaction* Springer (event), 2016, p. 166–177
- [GroV] GROV, Gudmund : Verifying the correctness of Hume programs-an approach combining algorithmic and deductive reasoning. In : *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE-05)*, p. 444–447
- [Guelfi et Mammari 2005] GUELFI, Nicolas ; MAMMARI, Amel : A formal semantics of timed activity diagrams and its PROMELA translation. In : *12th Asia-Pacific Software Engineering Conference (APSEC'05)* IEEE (event), 2005, p. 8–pp
- [Gurevich et al. 1995] GUREVICH, Yuri et al. : Evolving algebras 1993 : Lipari guide. In : *Specification and validation methods* (1995), p. 9–36
- [Hagen et Tinelli 2008] HAGEN, George ; TINELLI, Cesare : Scaling up the formal verification of Lustre programs with SMT-based techniques. In : *Formal Methods in Computer-Aided Design, 2008. FMCAD'08* IEEE (event), 2008, p. 1–9
- [Halbwachs et al. 1991] HALBWACHS, Nicholas ; CASPI, Paul ; RAYMOND, Pascal ; PILAUD, Daniel : The synchronous data flow programming language LUSTRE. In : *Proceedings of the IEEE* 79 (1991), Nr. 9, p. 1305–1320
- [Hamilton et Zeldin 1976] HAMILTON, Margaret ; ZELDIN, Saydean : Higher order software-A methodology for defining software. In : *IEEE Transactions on Software Engineering* (1976), Nr. 1, p. 9–32
- [Harel et Pnueli 1985] HAREL, David ; PNUELI, Amir : On the development of reactive systems. In : *Logics and models of concurrent systems*. Springer, 1985, p. 477–498

- [Hartson et Hix 1989] HARTSON, H R. ; HIX, Deborah : Human-computer interface development : concepts and systems for its management. In : *ACM Computing Surveys (CSUR)* 21 (1989), Nr. 1, p. 5–92
- [Haskins 2010] HASKINS, Cecilia : *INCOSE Systems Engineering Handbook v. 3.2*. International Council on Systems Engineering, 2010
- [Heitmeyer et al. 1998] HEITMEYER, Constance ; KIRBY, James ; LABAW, Bruce ; ARCHER, Myla ; BHARADWAJ, Ramesh : Using abstraction and model checking to detect safety violations in requirements specifications. In : *IEEE Transactions on software engineering* 24 (1998), Nr. 11, p. 927–948
- [Héon et al. 2010] HÉON, Michel ; BASQUE, Josianne ; PAQUETTE, Gilbert : Validation de la sémantique d'un modèle semi-formel de connaissances avec OntoCASE. In : *21èmes 21es Journées Francophones d'Ingénierie des Connaissances* Ecole des Mines d'Alès (event), 2010, p. 55–à
- [Hietter 2009] HIETTER, Yann : *Synthèse algébrique de lois de commande pour les systèmes à évènements discrets logiques*, École normale supérieure de Cachan-ENS Cachan, Ph.D. thesis, 2009
- [Hoare et al. 1985] HOARE, Charles Antony R. et al. : *Communicating sequential processes*. Volume 178. Prentice-hall Englewood Cliffs, 1985
- [Holzmann 2004] HOLZMANN, Gerard J. : *The SPIN model checker : Primer and reference manual*. Volume 1003. Addison-Wesley Reading, 2004
- [Hu et al. 2008] HU, He-xuan ; GEHIN, Anne-lise ; BAYART, Mireille : A formal framework of reconfigurable control based on model checking. In : *American Control Conference, Seattle, Washington, USA, 2008*
- [IEC 2013] IEC : *Programmable controllers - Part 3 : Programming languages*. International Electrotechnical Commission, 2013
- [IEEE-1220 1999] IEEE-1220 : IEEE Standard for Application and Management of the Systems Engineering Process. In : *IEEE Std 1220-1998* (1999), p. i–
- [Iliarov et Romanovsky 2012] ILIAISOV, Alexei ; ROMANOVSKY, Alexander : SafeCap domain language for reasoning about safety and capacity. In : *Workshop on Dependable Transportation Systems/Recent Advances in Software Dependability (WDTS-RASD) IEEE* (event), 2012, p. 1–10
- [ISA 1992] ISA : *5.1 Instrumentation Symbols and Identification*. 1992

- [ISO 2000] ISO, NFEN : 9001 : Systèmes de management de la qualité-Exigences. In : *Édition Afnor, décembre* (2000)
- [ISO-15288 2002] ISO-15288 : ISO 15288 Systems engineering-System life cycle processes. In : *International Standards Organisation* (2002)
- [Itani et Logrippo 2005] ITANI, Wissam ; LOGRIPPO, Luigi : Formal approaches to requirements engineering : from behavior trees to alloy. In : *Canadian Conference on Electrical and Computer Engineering, 2005*. IEEE (event), 2005, p. 916–919
- [Jackson 2002] JACKSON, Daniel : Alloy : a lightweight object modelling notation. In : *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), Nr. 2, p. 256–290
- [Jackson 2012] JACKSON, Daniel : *Software Abstractions-Logic, Language, and Analysis, Revised Edition*. 2012
- [Jarvinen et Kurki-Suonio 1991] JARVINEN, H. M. ; KURKI-SUONIO, R. : DisCo specification language : marriage of actions and objects. In : *[1991] Proceedings. 11th International Conference on Distributed Computing Systems*, May 1991, p. 142–151
- [Jones 1986] JONES, Cliff B. : *Systematic software development using VDM*. Volume 2. Citeseer, 1986
- [Juarez Orozco 2008] JUAREZ OROZCO, Zulema : *Vérification de propriétés quantitatives des systèmes logiques par model-checking hybride*, Cachan, Ecole normale supérieure, Ph.D. thesis, 2008
- [Junwei et al. 2008] JUNWEI, Du ; ZHONGWEI, Xu ; MENG, Mei : Verification of Scenario-Based Safety Requirement Specification on Components Composition. In : *International Conference on Computer Science and Software Engineering Volume 2* IEEE (event), 2008, p. 686–689
- [Jurjens et Shabalín 2004] JURJENS, Jan ; SHABALIN, Pasha : A foundation for tool-supported critical systems development with UML. In : *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems* IEEE (event), 2004, p. 398–405
- [Kaufmann et Moore 1996] KAUFMANN, Matt ; MOORE, J S. : ACL2 : An industrial strength version of Nqthm. In : *Proceedings of the Eleventh Annual Conference on Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security*. IEEE (event), 1996, p. 23–34
- [Keele 2007] KEELE, Staffs : Guidelines for performing systematic literature reviews in software engineering. In : *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*. 2007

- [Kellomäki 1997] KELLOMÄKI, Pertti : Verification of reactive systems using DisCo and PVS. In : *International Symposium of Formal Methods Europe* Springer (event), 1997, p. 589–604
- [Khoury et al. 2010] KHOURY, Joud ; ABDALLAH, Chaouki T. ; HEILEMAN, Gregory L. : Towards formalizing network architectural descriptions. In : *Abstract State Machines, Alloy, B and Z*. Springer, 2010, p. 132–145
- [Kim et Garlan 2010] KIM, Jung S. ; GARLAN, David : Analyzing architectural styles. In : *Journal of Systems and Software* 83 (2010), Nr. 7, p. 1216–1235
- [Kim et Carrington 2000] KIM, S-K ; CARRINGTON, David : An integrated framework with UML and Object-Z for developing a precise and understandable specification : the light control case study. In : *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific IEEE* (event), 2000, p. 240–248
- [Kim et al. 2005] KIM, Taeho ; STRINGER-CALVERT, David ; CHA, Sungdeok : Formal verification of functional properties of a SCR-style software requirements specification using PVS. In : *Reliability Engineering & System Safety* 87 (2005), Nr. 3, p. 351–363
- [Kitchenham et al. 2011] KITCHENHAM, Barbara A. ; BUDGEN, David ; BRERETON, O P. : Using mapping studies as the basis for further research—a participant-observer case study. In : *Information and Software Technology* 53 (2011), Nr. 6, p. 638–651
- [Kitchenham et al. 2009] KITCHENHAM, Barbara ; BRERETON, O P. ; BUDGEN, David ; TURNER, Mark ; BAILEY, John ; LINKMAN, Stephen : Systematic literature reviews in software engineering—a systematic literature review. In : *Information and software technology* 51 (2009), Nr. 1, p. 7–15
- [Kolski et al. 2001] KOLSKI, C ; EZZEDINE, H ; ABED, M : Développement du logiciel : des cycles classiques aux cycles enrichis sous l’angle des IHM. In : *Analyse et conception de l’IHM, Interaction Homme-Machine pour les SI, Hermes, Paris* (2001), p. 145–174
- [Kolski 1995] KOLSKI, Christophe : *Méthodes et modèles de conception et d’évaluation des interfaces homme-machine*, Université de Valenciennes et du Hainaut-Cambrésis, Habilitation à diriger des recherches en mathématiques et en informatique (HDR), 1995
- [Kossak et Mashkoor 2016] KOSSAK, Felix ; MASHKOOR, Atif : How to Evaluate the Suitability of a Formal Method for Industrial Deployment : A Survey. In : *Technical report SCCH-TR-1603, Software Competence Center Hagenberg GmbH*, 2016
- [Krause et al. 2012] KRAUSE, Anna ; OBST, Marcus ; URBAS, Leon : Extraction of safety relevant functions from CAE data for evaluating the reliability of communications systems. In : *IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA) IEEE* (event), 2012, p. 1–7

- [Kudo et al. 2015] KUDO, Taciana N. ; INF-UFMG, Renato de Freitas B. ; NETO, Dr ; ADJUNTO III, INFUFMG : Requirements Specification of Context-Aware Systems : A Systematic Mapping. In : *Revista de Sistemas de Informaçao da FSMA* (2015), Nr. 16, p. 41–51
- [Lakas et al. 1996] LAKAS, Abderrahmane ; GORDON, GS ; CHETWYND, Amanda : Specification and verification of real-time properties using LOTOS and SCTL. In : *Proceedings of the 8th International Workshop on Software Specification and Design* IEEE Computer Society (event), 1996, p. 75
- [Lallican 2007] LALLICAN, Jean-Louis : *Proposition d'une approche composant pour la conception de la commande des systèmes transistives*, Lorient, Ph.D. thesis, 2007
- [Lamb et al. 1978] LAMB, SS ; LECK, VG ; PETERS, LJ ; SMITH, GL : SAMM : A modeling tool for requirements and design specification. In : *Computer Software and Applications Conference, 1978. COMPSAC'78. The IEEE Computer Society's Second International* IEEE (event), 1978, p. 48–53
- [de Lamotte 2006] LAMOTTE, Florent F. de : *Proposition d'une approche haut niveau pour la conception, l'analyse et l'implantation des systèmes reconfigurables*, Université de Bretagne Sud, Ph.D. thesis, 2006
- [Lamport 2002] LAMPORT, Leslie : *Specifying systems : the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002
- [Lamsweerde 2000] LAMSWEEERDE, Axel v. : Formal specification : a roadmap. In : *Proceedings of the Conference on the Future of Software Engineering* ACM (event), 2000, p. 147–159
- [Larbi et Mohamad 2004] LARBI, RM ; MOHAMAD, JJ : VALID-2 : a practical modeling, simulation and verification software for distributed systems. In : *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* IEEE (event), 2004, p. 244
- [Larsen et al. 1997] LARSEN, Kim G. ; PETTERSSON, Paul ; YI, Wang : UPPAAL in a nutshell. In : *International Journal on Software Tools for Technology Transfer (STTT)* 1 (1997), Nr. 1, p. 134–152
- [Lee 2006] LEE, Eun J. : *Reconfiguration dynamique de la commande d'un système manufacturier : approche par la synthèse de la commande*, Ecole Centrale de Lille, Ph.D. thesis, 2006
- [Lee et al. 1995] LEE, Jonathan ; PAN, J-I ; HUANG, Wei T. : Integrating object-oriented requirements specifications with formal notations. In : *Proceedings., Seventh International Conference on Tools with Artificial Intelligence* IEEE (event), 1995, p. 34–41

- [Lin et al. 1996] LIN, Jyhjong ; KUNG, David C. ; HSIA, Pei : Object-oriented specification and formal verification of real-time systems. In : *Annals of Software Engineering 2* (1996), Nr. 1, p. 161–198
- [Linhaires et al. 2007] LINHARES, Marcos V. ; OLIVEIRA, Rômulo S de ; FARINES, Jean-Marie ; VERNADAT, François : Introducing the modeling and verification process in SysML. In : *IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)* IEEE (event), 2007, p. 344–351
- [Long et Denley 1990] LONG, John ; DENLEY, I : Evaluation for patrice. In : *Ergonomics Society Annual Conference* (1990)
- [Luzeaux et Ruault 2013] LUZEAUX, Dominique ; RUAULT, Jean-René : *100 questions pour comprendre et agir : L'ingénierie système*. Afnor Editions, 2013
- [Mallek et al. 2012] MALLEK, Sihem ; DACLIN, Nicolas ; CHAPURLAT, Vincent : The application of interoperability requirement specification and verification to collaborative processes in industry. In : *Computers in Industry* 63 (2012), Nr. 7, p. 643–658
- [Manna et Pnueli 2012] MANNA, Zohar ; PNUELI, Amir : *The temporal logic of reactive and concurrent systems : Specification*. Springer Science & Business Media, 2012
- [Mao et Qi 2012] MAO, Li ; QI, Deyu : Formal specification and proof of gridjack. In : *Computational Intelligence and Design (ISCID), 2012 Fifth International Symposium on Volume 1* IEEE (event), 2012, p. 110–114
- [Marangé 2008] MARANGÉ, Pascale : *Synthese et filtrage robuste de la commande pour des systeme manufacturiers sûrs de fonctionnement*, Université de Reims-Champagne Ardenne, Ph.D. thesis, 2008
- [Markey 2003] MARKEY, Nicolas : *Logiques temporelles pour la vérification : expressivité, complexité, algorithmes*, Orléans, Ph.D. thesis, 2003
- [Martinie De Almeida 2011] MARTINIE DE ALMEIDA, Célia : *Une approche à base de modèles synergiques pour la prise en compte simultanée de l'utilisabilité, la fiabilité et l'opérabilité des systèmes interactifs critiques*, Université de Toulouse, Université Toulouse III-Paul Sabatier, Ph.D. thesis, 2011
- [März et al. 2010] MÄRZ, Lothar ; KRUG, Wilfried ; ROSE, Oliver ; WEIGERT, Gerald : *Simulation and optimization in production and logistics : Practical guide with case studies*. Springer-Verlag, 2010
- [McAvinew et Mulley 2004] MCAVINEW, Thomas ; MULLEY, Raymond : *Control System Documentation : Applying Symbols and Identification*. ISA, 2004

- [McMillan 1999] MCMILLAN, Kenneth L. : Circular compositional reasoning about liveness. In : *Advanced Research Working Conference on Correct Hardware Design and Verification Methods* Springer (event), 1999, p. 342–346
- [Medvidovic et Taylor 2000] MEDVIDOVIC, Nenad ; TAYLOR, Richard N. : A classification and comparison framework for software architecture description languages. In : *Software Engineering, IEEE Transactions on* 26 (2000), Nr. 1, p. 70–93
- [Mekki 2012] MEKKI, Ahmed : *Contribution à la Spécification et à la Vérification des Exigences Temporelles : Proposition d'une extension des SRS d'ERTMS niveau 2*, Ecole Centrale de Lille, Ph.D. thesis, 2012
- [Mentré et al. 2012] MENTRÉ, David ; MARCHÉ, Claude ; FILLIÂTRE, Jean-Christophe ; ASUKA, Masashi : Discharging proof obligations from Atelier B using multiple automated provers. In : *International Conference on Abstract State Machines, Alloy, B, VDM, and Z* Springer (event), 2012, p. 238–251
- [Mertke et Frey 2001] MERTKE, Thomas ; FREY, Georg : Formal verification of PLC programs generated from signal interpreted Petri nets. In : *IEEE International Conference on Systems, Man, and Cybernetics* Volume 4 IEEE (event), 2001, p. 2700–2705
- [Mesli-Kesraoui et al. 2016a] MESLI-KESRAOUI, S. ; TOGUYENI, A. ; BIGNON, A. ; OQUENDO, F. ; KESRAOUI, D. ; BERRUET, P. : Formal and Joint Verification of Control Programs and Supervision Interfaces for Socio-technical Systems Components. In : *IFAC-PapersOnLine* 49 (2016), Nr. 19, p. 426 – 431. – URL <http://www.sciencedirect.com/science/article/pii/S2405896316321942>. – 13th IFAC Symposium on Analysis, Design, and Evaluation of Human-Machine Systems HMS 2016 Kyoto, Japan, 30 August-2 September 2016. – ISSN 2405-8963
- [Mesli-Kesraoui et al. 2016b] MESLI-KESRAOUI, Soraya ; BIGNON, Alain ; KESRAOUI, Djamel ; TOGUYENI, Armand ; OQUENDO, Flavio ; BERRUET, Pascal : Vérification Formelle de Chaînes de Contrôle-Commande d'Éléments de Conception Standardisés. In : *MOSIM 2016, 11ème Conférence Francophone de Modélisation, Optimisation et Simulation*, 2016, p. 1–10
- [Mesli-Kesraoui et al. 2016c] MESLI-KESRAOUI, Soraya ; KESRAOUI, Djamel ; OQUENDO, Flavio ; BIGNON, Alain ; TOGUYENI, Armand ; BERRUET, Pascal : *Formal Verification of Software-Intensive Systems Architectures Described with Piping and Instrumentation Diagrams*. p. 210–226. In : TEKINERDOGAN, Bedir (Editor) ; ZDUN, Uwe (Editor) ; BABAR, Ali (Editor) : *Software Architecture : 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 – December 2, 2016, Proceedings*. Cham : Springer International Publishing, 2016. – URL http://dx.doi.org/10.1007/978-3-319-48992-6_16. – ISBN 978-3-319-48992-6

- [Mhenni et al. 2013] MHENNI, Faïda ; NGUYEN, Nga ; KADIMA, Hubert ; CHOLEY, Jean-Yves : Safety analysis integration in a SysML-based complex system design process. In : *Systems Conference (SysCon), 2013 IEEE International IEEE (event)*, 2013, p. 70–75
- [Miller et Heimdahl 2004] MILLER, Steven P. ; HEIMDAHL, Mats P. : Early Validation of Requirements. In : *Building the Information Society*. Springer, 2004, p. 521–526
- [Milner 1989] MILNER, Robin : *Communication and concurrency*. Volume 84. Prentice hall New York etc., 1989
- [Mokadem et al. 2010] MOKADEM, Houda B. ; BERARD, Béatrice ; GOURCUFF, Vincent ; DE SMET, Olivier ; ROUSSEL, Jean-Marc : Verification of a timed multitask system with UPPAAL. In : *Automation Science and Engineering, IEEE Transactions on* 7 (2010), Nr. 4, p. 921–932
- [Mouchard 2002] MOUCHARD, Jean-Sébastien : *Proposition d'une approche méthodique pour la conception des systèmes automatisés de production : application aux systèmes transitiques*, Lorient, Ph.D. thesis, 2002
- [Moussa et Kolski 1991] MOUSSA, Faouzi ; KOLSKI, Christophe : ERGO-CONCEPTOR : système à base de connaissances ergonomiques pour la conception d'interface de contrôle de procédé industriel. In : *Technologies Avancées* 2 (1991), p. 5–14
- [Moussa et Kolski 1992] MOUSSA, Faouzi ; KOLSKI, Christophe : Vers une formalisation d'une démarche de conception de synoptiques industriels : application au système ERGO-CONCEPTOR. In : *Proceedings Colloque ERGO-IA Ergonomie et Informatique Avancée*, 1992, p. 7–9
- [Moussa et al. 2002] MOUSSA, Faouzi ; RIAHI, Meriem ; KOLSKI, Christophe ; MOALLA, Mohamed : Interpreted Petri Nets used for human-machine dialogue specification. In : *Integrated Computer-Aided Engineering* 9 (2002), Nr. 1, p. 87–98
- [Newcombe 2014] NEWCOMBE, Chris : Why amazon chose TLA+. In : *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* Springer (event), 2014, p. 25–39
- [Ng et Butler 2003] NG, Muan Y. ; BUTLER, Michael : Towards formalizing UML state diagrams in CSP. In : *First International Conference on Software Engineering and Formal Methods* IEEE (event), 2003, p. 138–147
- [Nguyen et al. 1997] NGUYEN, Tho-Hau ; ABRAN, Alain ; BAYARD, Raymond ; DECHANTAL, Odette : Les concepts de la réutilisation du logiciel et la pratique institutionnelle dans les entreprises québécoises. In : *GENIE LOGICIEL-TOULOUSE THEN PARIS-* (1997), p. 40–48

- [Niel et Craye 2002] NIEL, Eric ; CRAYE, Etienne : Maîtrise des risques et sûreté de fonctionnement des systèmes de production. In : *Productique : information, commande, communication*. Lavoisier (2002)
- [Nipkow et al. 2002] NIPKOW, Tobias ; PAULSON, Lawrence C. ; WENZEL, Markus : *Isabelle/HOL : a proof assistant for higher-order logic*. Volume 2283. Springer Science & Business Media, 2002
- [Ogata et Futatsugi 2003] OGATA, Kazuhiro ; FUTATSUGI, Kokichi : Proof scores in the OTS/CafeOBJ method. In : *International Conference on Formal Methods for Open Object-Based Distributed Systems* Springer (event), 2003, p. 170–184
- [Ogata et al. 2004] OGATA, Kazuhiro ; YAMAGISHI, Daigo ; SEINO, Takahiro ; FUTATSUGI, Kokichi : Modeling and verification of hybrid systems based on equations. In : *Design Methods and Applications for Distributed Embedded Systems*. Springer, 2004, p. 43–52
- [OMG 2014] OMG : OMG Object Constraint Language (OMG OCL), version 2.4. In : *OMG Adopted Specification (formal/03-02-14)* (2014)
- [OMG 2015a] OMG : OMG Systems Modeling Language (OMG SysML). In : *OMG Adopted Specification (formal/2015-06-03)* (2015)
- [OMG 2015b] OMG : OMG Unified Modeling Language (OMG UML), version 2.5. 2015. – Research Report
- [OMG 2015c] OMG : Semantics of Business Vocabulary and Rules (SBVR), version 1.3. 2015. – Research Report
- [OMG 2016] OMG : *Semantics Of A Foundational Subset For Executable UML Models (FUML), version 1.2.1*. 2016
- [OMG 2003] OMG, MDA : Guide Version 1.0. 1. In : *Object Management Group* 62 (2003), p. 34
- [Oquendo 2004] OQUENDO, Flavio : π -ADL : an Architecture Description Language based on the higher-order typed π -calculus for specifying dynamic and mobile software architectures. In : *SIGSOFT Softw. Eng. Notes* 29 (2004), may, Nr. 3, p. 1–14. – URL <http://doi.acm.org/10.1145/986710.986728>. – ISSN 0163-5948
- [Owre et al. 1999] OWRE, Sam ; SHANKAR, Natarajan ; RUSHBY, John M. ; STRINGER-CALVERT, David W. : PVS system guide. In : *Computer Science Laboratory, SRI International, Menlo Park, CA* 1 (1999), Nr. 5, p. 7
- [Palanque et Bastide 1995] PALANQUE, Philippe A. ; BASTIDE, Remi : Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In : *Interactive systems : Design, specification, and verification*. Springer, 1995, p. 383–400

- [Palomares et al. 2014] PALOMARES, Cristina ; FRANCH, Xavier ; QUER, Carme : Requirements reuse and patterns : a survey. In : *International Working Conference on Requirements Engineering : Foundation for Software Quality* Springer (event), 2014, p. 301–308
- [Paterno 1997] PATERNO, Fabio : Formal reasoning about dialogue properties with automatic support. In : *Interacting with computers* 9 (1997), Nr. 2, p. 173–196
- [Paternò et al. 1997] PATERNÒ, Fabio ; MANCINI, Cristiano ; MENICONI, Silvia : Concur-TaskTrees : A diagrammatic notation for specifying task models. In : *Human-Computer Interaction INTERACT'97* Springer (event), 1997, p. 362–369
- [de Paula et al. 2000] PAULA, Virginia C. de ; JUSTO, GRB ; CUNHA, PR F. : Specifying and verifying reconfigurable software architectures. In : *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on* IEEE (event), 2000, p. 21–31
- [Petersen et Gencel 2013] PETERSEN, Kai ; GENCEL, Cigdem : Worldviews, research methods, and their relationship to validity in empirical software engineering research. In : *Joint Conference of the 23rd International Workshop on Software Measurement and the Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA)* IEEE (event), 2013, p. 81–89
- [Petersen et al. 2015] PETERSEN, Kai ; VAKKALANKA, Sairam ; KUZNIARZ, Ludwik : Guidelines for conducting systematic mapping studies in software engineering : An update. In : *Information and Software Technology* 64 (2015), p. 1–18
- [Petri 1966] PETRI, Carl A. : Communication with automata / Technical report, DTIC Document. 1966. – Research Report
- [Pham 2007] PHAM, Hoang : *System software reliability*. Springer Science & Business Media, 2007
- [Prähofer et Hurnaus 2010] PRÄHOFER, Herbert ; HURNAUS, Dominik : MONACO-A domain-specific language supporting hierarchical abstraction and verification of reactive control programs. In : *2010 8th IEEE International Conference on Industrial Informatics* IEEE (event), 2010, p. 908–914
- [Prat et al. 2016] PRAT, Sophie ; RAUFFET, Philippe ; BERRUET, Pascal ; BIGNON, Alain et al. : A multi-level requirements modeling for sociotechnical system simulation-based checking. In : *SMC*, 2016
- [Prat et al. 2015] PRAT, Sophie ; RAUFFET, Philippe ; BIGNON, Alain ; BERRUET, Pascal : Vers l'intégration d'une approche de génération automatique de modèle de simulation dans un flot de conception de contrôle-commande. In : *JDJN MACS*, 2015

- [Quinton et Graf 2008] QUINTON, Sophie ; GRAF, Susanne : Contract-based verification of hierarchical systems of components. In : *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM'08*. IEEE (event), 2008, p. 377–381
- [Rafe et al. 2009a] RAFE, Vahid ; RAFEH, Reza ; MIRALVAND, Mohamad Reza Z. ; ALAVI-ZADEH, Alavie S. : Automated Model Checking of Stochastic Graph Transformation Systems. In : *International Conference on Computer Technology and Development, ICCTD'09*. Volume 1 IEEE (event), 2009, p. 211–215
- [Rafe et al. 2009b] RAFE, Vahid ; RAHMANI, Adel T. ; BARESI, Luciano ; SPOLETINI, Paola : Towards automated verification of layered graph transformation specifications. In : *IET software* 3 (2009), Nr. 4, p. 276–291
- [Rechard 2015] RECHARD, Julien : *Introduction of ergonomic criteria in an automatic generation system of supervision interfaces*, Université de Bretagne Sud, Ph.D. thesis, november 2015. – URL <https://tel.archives-ouvertes.fr/tel-01309300>
- [Reisig 2013] REISIG, Wolfgang : *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. – I–XXVI, 1–230 p. – ISBN 978-3-642-33277-7
- [Ribeiro et al. 2005] RIBEIRO, Oscar R. ; FERNANDES, Joao M. ; PINTO, Luis F. : Model checking embedded systems with PROMELA. In : *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)* IEEE (event), 2005, p. 378–385
- [Rockstrom et Saracco 1982] ROCKSTROM, Anders ; SARACCO, Roberto : SDL-CCITT specification and description language. In : *IEEE Transactions on Communications* 30 (1982), Nr. 6, p. 1310–1318
- [Ross et Lindsay 1994] ROSS, Kelvin J. ; LINDSAY, Peter A. : A precise examination of the behaviour of process models. In : *International Symposium of Formal Methods Europe* Springer (event), 1994, p. 251–270
- [Roussel et Denis 2002] ROUSSEL, Jean-Marc ; DENIS, Bruno : Safety properties verification of ladder diagram programs. In : *Journal européen des systèmes automatisés* 36 (2002), Nr. 7, p. pp–905
- [Runeson et Höst 2009] RUNESON, Per ; HÖST, Martin : Guidelines for conducting and reporting case study research in software engineering. In : *Empirical software engineering* 14 (2009), Nr. 2, p. 131–164
- [Sacha 2010] SACHA, Krzysztof : Verification and implementation of software for dependable controllers. In : *International Journal of Critical Computer-Based Systems* 1 (2010), Nr. 1-3, p. 238–254

- [Sadolewski 2011] SADOLEWSKI, Jan : Automated conversion of ST control programs to Why for verification purposes. In : *Federated Conference on Computer Science and Information Systems (FedCSIS)* IEEE (event), 2011, p. 849–854
- [Sadoun et al. 2013] SADOUD, Driss ; DUBOIS, Catherine ; GHAMRI-DOUDANE, Yacine ; GRAU, Brigitte : From natural language requirements to formal specification using an ontology. In : *2013 IEEE 25th International Conference on Tools with Artificial Intelligence* IEEE (event), 2013, p. 755–760
- [Sarmiento et al. 2008] SARMENTO, Cleber A. ; SILVA, José R. ; MIYAGI, Paulo E. ; SANTOS FILHO, Diolino J. : Modeling of programs and its verification for programmable logic controllers. In : *IFAC Proceedings Volumes* 41 (2008), Nr. 2, p. 10546–10551
- [Schnoebelen et al. 1999] SCHNOEBELEN, Philippe ; BÉRARD, Béatrice ; BIDOIT, Michel ; LAROUSSINIE, François : *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, 1999
- [Seino et al. 2004] SEINO, Takahiro ; OGATA, Kazuhiro ; FUTATSUGI, Kokichi : Supporting case analysis with algebraic specification languages. In : *The Fourth International Conference on Computer and Information Technology, CIT'04*. IEEE (event), 2004, p. 1073–1080
- [Selby et Selby 2007] SELBY, Paige C. ; SELBY, Richard W. : 4.4. 2 Measurement-Driven Systems Engineering Using Six Sigma Techniques to Improve Software Defect Detection. In : *INCOSE International Symposium Volume 17* Wiley Online Library (event), 2007, p. 640–651
- [Shankar et al. 2001] SHANKAR, N ; OWRE, S ; RUSHBY, JM ; STRINGER-CALVERT, DWJ : The PVS System Guide. In : *SRI International, December* (2001)
- [Shkel et Ferrier 1997] SHKEL, Eugenia ; FERRIER, Nicola J. : Specifying and verifying visual grasping tasks. In : *IEEE International Conference on Robotics and Automation* Volume 1 IEEE (event), 1997, p. 688–694
- [Shu et al. 2002] SHU, Guoqiang ; LI, Chao ; WANG, Qing ; LI, Mingshu : Validating objected-oriented prototype of real-time systems with timed automata. In : *Proceedings. 13th IEEE International Workshop on Rapid System Prototyping* IEEE (event), 2002, p. 99–106
- [Simon et Girault 2001] SIMON, Daniel ; GIRAULT, Alain : Synchronous programming of automatic control applications using Orccad and Esterel. In : *Proceedings of the 40th IEEE Conference on Decision and Control* Volume 4 IEEE (event), 2001, p. 3290–3295
- [Sirjani et al. 2004] SIRJANI, Marjan ; MOVAGHAR, Ali ; SHALI, Amin ; DE BOER, Frank S. : Modeling and verification of reactive systems using Rebeca. In : *Fundamenta Informaticae* 63 (2004), Nr. 4, p. 385–410

- [Smith 2012] SMITH, Graeme : *The Object-Z specification language*. Volume 1. Springer Science & Business Media, 2012
- [Snook et Butler 2001] SNOOK, Colin ; BUTLER, MJ : Using a graphical design tool for formal specification. In : G.KADODA (Editor) : *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group*, 2001, p. 311–321
- [Soliman et Frey 2011] SOLIMAN, Doaa ; FREY, Georg : Verification and validation of safety applications based on PLCopen safety function blocks. In : *Control engineering practice* 19 (2011), Nr. 9, p. 929–946
- [Sourisse et Boudillon 1997] SOURISSE, Claude ; BOUDILLON, Louis : *La sécurité des machines automatisées : Techniques et moyens de prévention opératifs, systèmes de commande, utilisation des machines*. Institut Schneider Formation, 1997
- [Spivey et Abrial 1992] SPIVEY, J M. ; ABRIAL, JR : *The Z notation*. Prentice Hall Hemel Hempstead, 1992
- [Stuart et al. 2001] STUART, Douglas A. ; BROCKMEYER, Monica ; MOK, Aloysius K. ; JAHANIAN, Farnam : Simulation-verification : Biting at the state explosion problem. In : *IEEE Transactions on Software Engineering* 27 (2001), Nr. 7, p. 599–617
- [Sülflow et Drechsler 2008] SÜLFLOW, Andre ; DRECHSLER, Rolf : Verification of PLC programs using formal proof techniques. In : *FORMS/FORMAT* 2008 (2008), p. 43–50
- [Suryadevara 2013] SURYADEVARA, Jagadish : Validating EAST-ADL timing constraints using UPPAAL. In : *39th Euromicro Conference on Software Engineering and Advanced Applications* IEEE (event), 2013, p. 268–275
- [Suryadevara et al. 2011] SURYADEVARA, Jagadish ; SECELEANU, Cristina ; PETERSSON, Paul : Pattern-driven support for designing component-based architectural models. In : *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on* IEEE (event), 2011, p. 187–196
- [Svendsen et al. 2011] SVENDSEN, Andreas ; HAUGEN, Øystein ; MOLLER-PEDERSEN, Birger : Using variability models to reduce verification effort of train station models. In : *2011 18th Asia-Pacific Software Engineering Conference* IEEE (event), 2011, p. 348–356
- [Taylor et al. 2009] TAYLOR, Richard N. ; MEDVIDOVIC, Nenad ; DASHOFY, Eric M. : *Software architecture : foundations, theory, and practice*. Wiley Publishing, 2009
- [Technologies] TECHNOLOGIES, Esterel : *Scade suite product description*. – URL <http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html>

- [Teichroew et al. 1980] TEICHROEW, Daniel ; MACASOVIC, Petar ; HERSHEY, Ernest A. ; YAMAMOTO, Yuzo : Application of the entity-relationship approach to information processing systems modelling. In : *Proceedings of the 1st International Conference on the Entity-Relationship Approach to Systems Analysis and Design* North-Holland Publishing Co. (event), 1980, p. 15–38
- [Tisserant et al. 2007] TISSERANT, Edouard ; BESSARD, Laurent ; SOUSA, Mário de : An open source IEC 61131-3 integrated development environment. In : *5th IEEE International Conference on Industrial Informatics* Volume 1 IEEE (event), 2007, p. 183–187
- [Toguyeni 1992] TOGUYENI, Abdoul : *Surveillance et diagnostic en ligne dans les ateliers flexibles de l'industrie manufacturière*, Ph.D. thesis, 1992
- [Toguyeni et al. 2007] TOGUYENI, Armand ; DANGOUMAU, Nathalie ; LEE, Eun J. : A Synthesis Approach for Reconfigurable Manufacturing Systems Design Based on Petri Nets. In : *Studies in informatics and control* 16 (2007), Nr. 1, p. 115
- [Traonouez 2009] TRAONOUZ, Louis-Marie : *Vérification et dépliages de réseaux de Petri temporels paramétrés*, Université de Nantes, Ph.D. thesis, 2009
- [Trattnig et Kerner 1980] TRATTNIG, Werner ; KERNER, Helmut : EDDA-A very high-level programming and specification language in the style of SADT. In : *Proceedings of IEEE Annual International Computer Software and Applications Conference (COMPSAC'80)*, 1980, p. 436–443
- [Truong et Souquières 2004] TRUONG, N-T ; SOUQUIÈRES, Jeanine : An approach for the verification of UML models using B. In : *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems* IEEE (event), 2004, p. 195–202
- [Tse et Pong 1991] TSE, TH ; PONG, L : An examination of requirements specification languages. In : *The Computer Journal* 34 (1991), Nr. 2, p. 143–152
- [Vicente 1999] VICENTE, Kim J. : *Cognitive work analysis : Toward safe, productive, and healthy computer-based work*. CRC Press, 1999
- [Wang et al. 2004] WANG, Farn et al. : Formal verification of timed systems : A survey and perspective. In : *Proceedings of the IEEE* 92 (2004), Nr. 8, p. 1283–1305
- [Wang et al. 2009] WANG, Rui ; GU, Ming ; SONG, Xiaoyu ; WAN, Hai : Formal specification and code generation of programable logic controllers. In : *14th IEEE International Conference on Engineering of Complex Computer Systems* IEEE (event), 2009, p. 102–109

- [Weng et Litz 2001] WENG, Xiyang ; LITZ, Lothar : Model checking of signal interpreted petri nets. In : *IEEE International Conference on Systems, Man, and Cybernetics* Volume 4 IEEE (event), 2001, p. 2748–2752
- [Wiegers et Beatty 2013] WIEGERS, Karl ; BEATTY, Joy : *Software requirements*. Pearson Education, 2013
- [Wing 1990] WING, Jeannette M. : A specifier's introduction to formal methods. In : *Computer* 23 (1990), Nr. 9, p. 8–22
- [Wohlin et al. 2012] WOHLIN, Claes ; RUNESON, Per ; HÖST, Martin ; OHLSSON, Magnus C. ; REGNELL, Björn ; WESSLÉN, Anders : *Experimentation in software engineering*. Springer Science & Business Media, 2012
- [Wong et al. 2008] WONG, Stephen ; SUN, Jing ; WARREN, Ian ; SUN, Jun : A scalable approach to multi-style architectural modeling and verification. In : *13th IEEE International Conference on Engineering of Complex Computer Systems* IEEE (event), 2008, p. 25–34
- [Wonham et Ramadge 1988] WONHAM, W M. ; RAMADGE, Peter J. : Modular supervisory control of discrete-event systems. In : *Mathematics of control, signals and systems* 1 (1988), Nr. 1, p. 13–30
- [Woodcock et al. 2009] WOODCOCK, Jim ; LARSEN, Peter G. ; BICARREGUI, Juan ; FITZGERALD, John : Formal methods : Practice and experience. In : *ACM computing surveys (CSUR)* 41 (2009), Nr. 4, p. 19
- [Xiang et al. 2004] XIANG, Jianwen ; FUTATSUGI, Kokichi ; HE, Yanxiang : Fault tree and formal methods in system safety analysis. In : *The Fourth International Conference on Computer and Information Technology, CIT'04*. IEEE (event), 2004, p. 1108–1115
- [Xin-feng et al. 2009] XIN-FENG, Zhu ; JIAN-DONG, Wang ; BIN, Li ; JUN-WU, Zhu ; JUN, Wu : Methods to tackle state explosion problem in model checking. In : *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on* Volume 2 IEEE (event), 2009, p. 329–331
- [Yamauchi et al. 2014] YAMAUCHI, Masato ; ITO, Nobuhiro ; KAWABE, Yoshinobu : Toward Formal Verification of ECU for Gasoline Direct Injection Engines. In : *IIAI 3rd International Conference on Advanced Applied Informatics (IIAIAI)* IEEE (event), 2014, p. 889–894
- [Yang et al. 2001] YANG, SH ; STURSBERG, O ; CHUNG, PWH ; KOWALEWSKI, S : Automatic safety analysis of computer-controlled plants. In : *Computers & Chemical Engineering* 25 (2001), Nr. 4, p. 913–922
- [Yavuz-Kahveci et Bultan 2005] YAVUZ-KAHVECI, Tuba ; BULTAN, Tevfik : Verification of parameterized hierarchical state machines using action language verifier. In : *Proceedings*.

- Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE (event), 2005, p. 79–88
- [Yu 1997] YU, Eric S. : Towards modelling and reasoning support for early-phase requirements engineering. In : *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on* IEEE (event), 1997, p. 226–235
- [Zhang et al. 2012] ZHANG, Jiexin ; LIU, Yang ; AUGUSTON, Mikhail ; SUN, Jun ; DONG, Jin S. : Using monterey phoenix to formalize and verify system architectures. In : *2012 19th Asia-Pacific Software Engineering Conference Volume 1* IEEE (event), 2012, p. 644–653
- [Zhang et al. 2011] ZHANG, Tao ; HUANG, Shaobin ; LV, Tianyang ; HUANG, Hongtao : Specification and verification of policy using RAISE and modelchecking. In : *4th International Conference on Biomedical Engineering and Informatics (BMEI) Volume 4* IEEE (event), 2011, p. 2082–2086
- [Zhang et Mackworth 1996] ZHANG, Ying ; MACKWORTH, Alan K. : Specification and verification of hybrid dynamic systems with timed \forall -automata. In : *Hybrid Systems III.* Springer, 1996, p. 587–603

Annexes



Données de la Systematic Mapping

A.1 Requêtes de recherche dans les différentes bases

Base	Request
IEEE	((("requirements specification" OR "requirements modeling" OR "requirements modelling" OR "requirements languages" OR "specification languages") AND ("formal verification") NOT (network OR test OR testing OR hardware OR web OR service OR circuit OR agent))
ACM	Title : (((("requirements specification" or "requirements modeling" or "requirements modelling" or "requirements languages" or "specification languages") and ("formal verification") not (network or test or testing or hardware or web or service or circuit or agent)))) or Abstract : (((("requirements specification" or "requirements modeling" or "requirements modelling" or "requirements languages" or "specification languages") and ("formal verification") not (network or test or testing or hardware or web or service or circuit or agent)))) or Keywords : (((("requirements specification" or "requirements modeling" or "requirements modelling" or "requirements languages" or "specification languages") and ("formal verification") not (network or test or testing or hardware or web or service or circuit or agent))))
Science-Direct	TITLE-ABS-KEY (("requirements specification" OR "requirements modeling" OR "requirements modelling" OR "requirements languages" OR "specification languages") AND ("formal verification") AND NOT (network OR test OR testing OR hardware OR web OR service OR circuit OR agent))
Springer-Link	((("requirements specification" OR "requirements modeling" OR "requirements modelling" OR "requirements languages" OR "specification languages") AND ("formal verification") NOT (network OR test OR testing OR hardware OR web OR service OR circuit OR agent))

Tableau A.1 – Requêtes de recherche pour les différentes bases

A.2 Listes des articles retenus pour la *systematic mapping*

Tableau A.2 – Articles retenus pour l'étude

ID	Ref	Titre	Année
1	[Brink et al. 1998]	Automatic analysis of embedded systems specified in Astral	1998
2	[Alemán et Álvarez 2000]	Can intuition become rigorous? Foundations for UML model verification tools	2000
3	[Junwei et al. 2008]	Verification of Scenario-Based Safety Requirement Specification on Components Composition	2008
4	[GroV]	Verifying the correctness of Hume programs-an approach combining algorithmic and deductive reasoning	
5	[Cha et Yoo 2012]	A safety-focused verification using software fault trees	2012
6	[Furia et al. 2007]	Automated compositional proofs for real-time systems	2007
7	[Kim et al. 2005]	Formal verification of functional properties of a SCR-style software requirements specification using PVS	2005
8	[Mallek et al. 2012]	The application of interoperability requirement specification and verification to collaborative processes in industry	2012
9	[Yamauchi et al. 2014]	Toward Formal Verification of ECU for Gasoline Direct Injection Engines	2014
10	[Basit-Ur-Rahim et al. 2014]	Formal verification of sequence diagram using DiVinE	2014
11	[Mertke et Frey 2001]	Formal verification of PLC programs generated from signal interpreted Petri nets	2001
12	[Fuxman et al. 2001]	Model checking early requirements specifications in Tropos	2001
13	[Bloom et al. 1996]	Verifying SOS specifications	1996
14	[Cimatti et Tonetta 2012]	A property-based proof system for contract-based design	2012
15	[Linhares et al. 2007]	Introducing the modeling and verification process in SysML	2007
16	[Dong et al. 2001]	Model checking UML statecharts	2001
17	[Seino et al. 2004]	Supporting case analysis with algebraic specification languages	2004
18	[Alenjunga et Lenartson 2009]	Formal verification of PLC controlled systems using sensor graphs	2009
19	[Goldson 2002]	Formal verification of μ -charts	2002
20	[Truong et Souquières 2004]	An approach for the verification of UML models using B	2004

ID	Ref	Titre	Année
21	[Xiang et al. 2004]	Fault tree and formal methods in system safety analysis	2004
22	[Alavizaedh et al. 2007]	ReUML: A UML profile for modeling and verification of reactive systems	2007
23	[Iliasov et Romanovsky 2012]	SafeCap domain language for reasoning about safety and capacity	2012
24	[Gang et Zhiming 2003]	A new method for FMS modeling and formal verification	2003
25	[Hagen et Tinelli 2008]	Scaling up the formal verification of Lustre programs with SMT-based techniques	2008
26	[Aredo et Owe 2005]	Model-based verification in the development of dependable systems	2005
27	[Bonfe et Fantuzzi 2004]	A practical approach to object-oriented modeling of logic control systems for industrial applications	2004
28	[Evans et al. 2004]	How to verify dynamic properties of information systems	2004
29	[Lee et al. 1995]	Integrating object-oriented requirements specifications with formal notations	1995
30	[Stuart et al. 2001]	Simulation-verification: Biting at the state explosion problem	2001
31	[Weng et Litz 2001]	Model checking of signal interpreted petri nets	2001
32	[Hu et al. 2008]	A formal framework of reconfigurable control based on model checking	2008
33	[Jurjens et Shabalin 2004]	A foundation for tool-supported critical systems development with UML	2004
34	[Du et al. 2000]	Tabled resolution+ constraints: A recipe for model checking real-time systems	2000
35	[Zhang et al. 2011]	Specification and verification of policy using RAISE and modelchecking	2011
36	[Chen et al. 2009]	Dependability analysis for AADL models by PVS	2009
37	[Rafe et al. 2009a]	Automated Model Checking of Stochastic Graph Transformation Systems	2009a
38	[Mhenni et al. 2013]	Safety analysis integration in a SysML-based complex system design process	2013
39	[Itani et Logrippo 2005]	Formal approaches to requirements engineering: from behavior trees to alloy	2005
40	[Chang et al. 1994]	Compositional verification of real-time systems	1994
41	[Shu et al. 2002]	Validating objected-oriented prototype of real-time systems with timed automata	2002

ID	Ref	Titre	Année
42	[Guelfi et Mammari 2005]	A formal semantics of timed activity diagrams and its PROMELA translation	2005
43	[Simon et Girault 2001]	Synchronous programming of automatic control applications using Orccad and Esterel	2001
44	[Gnesi et al. 1999]	Model checking UML statechart diagrams using JACK	1999
45	[Suryadevara 2013]	Validating EAST-ADL timing constraints using UPPAAL	2013
46	[Rafe et al. 2009b]	Towards automated verification of layered graph transformation specifications	2009b
47	[Svendsen et al. 2011]	Using variability models to reduce verification effort of train station models	2011
48	[Sadolewski 2011]	Automated conversion of ST control programs to Why for verification purposes	2011
49	[Bastias et al. 2011]	An automated analysis of errors for BPM processes modeled using an in-house Infosys tool	2011
50	[Larbi et Mohamad 2004]	VALID-2: a practical modeling, simulation and verification software for distributed systems	2004
51	[D'Souza et al. 2014]	Architectural semantics of AADL using Event-B	2014
52	[Prähofer et Hurnaus 2010]	MONACO-A domain-specific language supporting hierarchical abstraction and verification of reactive control programs	2010
53	[Yavuz-Kahveci et Bultan 2005]	Verification of parameterized hierarchical state machines using action language verifier	2005
54	[de Paula et al. 2000]	Specifying and verifying reconfigurable software architectures	2000
55	[Ghazel et al. 2009]	Verification of temporal requirements of complex systems using UML patterns, application to a railway control example	2009
56	[Cao et al. 2011]	Automatic generation and verification of interlocking tables based on domain specific language for computer based interlocking systems (DSL-CBI)	2011
57	[Dumas et al. 2010]	Context modelling and partial-order reduction: Application to SDL industrial embedded systems	2010
58	[Ribeiro et al. 2005]	Model checking embedded systems with PROMELA	2005
59	[Apvrille et al. 2004]	TURTLE: A real-time UML profile supported by a formal validation toolkit	2004

ID	Ref	Titre	Année
60	[Fleischhack et Grahlmann 1998]	Towards Compositional Verification of SDL Systems	1998
61	[Chechik et Gannon 2001]	Automatic analysis of consistency between requirements and designs	2001
62	[Mao et Qi 2012]	Formal specification and proof of gridjack	2012
63	[Lakas et al. 1996]	Specification and verification of real-time properties using LOTOS and SCTL	1996
64	[Shkel et Ferrier 1997]	Specifying and verifying visual grasping tasks	1997
65	[Zhang et al. 2012]	Using monterey phoenix to formalize and verify system architectures	2012
66	[Esteve et al. 2012]	Formal correctness, safety, dependability, and performance analysis of a satellite	2012
67	[Heitmeyer et al. 1998]	Using abstraction and model checking to detect safety violations in requirements specifications	1998
68	[Suryadevara et al. 2011]	Pattern-driven support for designing component-based architectural models	2011
69	[Ng et Butler 2003]	Towards formalizing UML state diagrams in CSP	2003
70	[Amorim et al. 2005]	A methodology for mapping live sequence chart to coloured Petri net	2005
71	[Ross et Lindsay 1994]	A precise examination of the behaviour of process models	1994
72	[Miller et Heimdahl 2004]	Early Validation of Requirements	2004
73	[Dhaussy et al. 2009]	Evaluating context descriptions and property definition patterns for software formal validation	2009
74	[Chen et al. 2007]	Machine-assisted proof support for validation beyond Simulink	2007
75	[Gómez-Martínez et al. 2014]	Model-based verification of safety contracts	2014
76	[Dabaghchian et Azgomi 2015]	Model checking the observational determinism security property using PROMELA and SPIN	2015
77	[Ogata et al. 2004]	Modeling and verification of hybrid systems based on equations	2004
78	[Lin et al. 1996]	Object-oriented specification and formal verification of real-time systems	1996
79	[Zhang et Mackworth 1996]	Specification and verification of hybrid dynamic systems with timed \forall -automata	1996

ID	Ref	Titre	Année
80	[Kellomäki 1997]	Verification of reactive systems using DisCo and PVS	1997
81	[Archer et Heitmeyer 1997]	Verifying hybrid systems modeled as timed automata: A case study	1997

B

Guide d'entretien réalisé avec les experts métiers

B.1 Présentation

"Dans le cadre de ma thèse, je suis amenée à vérifier la conception des systèmes complexes. A cette fin nous allons effectuer un entretien en quatre parties. Dans un premier temps mon objectif sera de comprendre votre travail, vos objectifs et comment vous l'effectuez. Dans un deuxième temps, l'objectif sera de comprendre comment vous traitez les exigences. Dans un troisième temps, l'objectif sera d'avoir votre avis sur les différentes étapes de la conception. Dans un quatrième temps mon objectif est d'avoir votre avis sur les différentes techniques de vérification de la conception. Il n'y a pas de mauvaises réponses, mon but est d'avoir votre avis. Pour le bien de l'étude, l'entretien doit être enregistré, cependant cet enregistrement ne sera pas diffusé et je serai la seule à y avoir accès."

1. Pouvez-vous vous présenter brièvement ?
2. Sur quel système avez-vous déjà travaillé ?
3. A quel poste ?

B.2 Analyse des exigences

"Mon objectif est de comprendre les différents types d'exigences, leurs ressources, et la façon dont elles sont interprétées"

4. Dans votre activité, quel est votre rôle vis-à-vis des exigences ?
5. Selon vous, qui doit formuler les exigences au début du projet ?
6. Selon vous, comment doit-on formuler les exigences ?
7. Comment classez-vous les exigences ? Quelles catégories ?
8. Est-ce que les intervenants utilisent les mêmes exigences ?
9. Comment sont-elles tracées pendant la conception ?
10. Pouvez-vous me citer quelques exemples pour Eds :
 - Exigences spécifiées par le client ?

- Exigences implicites (non formulées par le client) mais nécessaires pour le respect des règles ?
- Exigences réglementaires ?
- Exigences définies par le métier ?

B.3 Analyse de la conception système

"On va s'intéresser à l'étape de la conception du système"

11. Quels sont les différents intervenants dans la conception (exemple : Eds) ?
12. Quelles est le métier de chaque intervenant ?
13. Quelles sont les erreurs fréquemment rencontrées pendant la conception ?
14. Selon vous, quelles sont les erreurs à ne jamais faire pendant la conception du :
 - P&ID ?
 - Commande ?
 - Supervision ?

B.4 Analyse de la vérification

"On va s'intéresser à l'étape de la vérification d'une conception, son intérêt, et son type"

15. Selon vous, la vérification est-elle nécessaire dans la conception du projet ? Pourquoi ?
16. Pour quel type de vérification optez-vous pour chaque niveau de conception ?
17. Selon vous, qu'est ce qui est judicieux de vérifier sur le P&ID ?
18. Selon vous, qu'est ce qui est judicieux de vérifier sur la commande ?
19. Selon vous, qu'est ce qui est judicieux de vérifier sur l'interface de supervision ?
20. Selon vous, quels sont les critères de sécurité à prendre en compte pendant la conception ?
21. Selon vous, quel est l'ordre de priorité entre les fonctions ?
22. Selon vous, quelles sont les séquences interdites (pour la commande) ?

B.5 Confrontation avec le P&ID, la commande, et l'interface de supervision

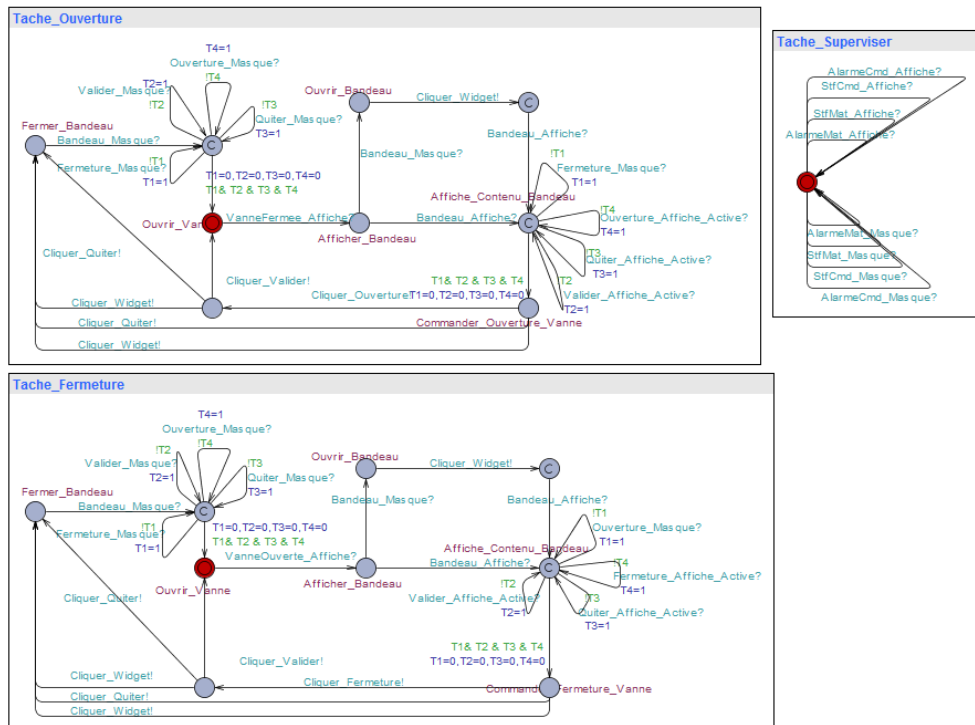
23. Que pensez-vous de cette conception ? *Relances : En termes de structure ? Par rapport aux informations ? En quoi est-elle similaire à celle que vous utilisez habituellement ? En quoi est-elle différente de celle que vous utilisez habituellement ? Selon vous, y a-t-il des éléments manquants à cette conception ? Selon vous, y a-t-il des éléments qui ne sont pas réellement utiles pour cette conception ?*

24. Comment procédez-vous pour la vérifier ?
25. Selon vous, que pourrait apporter un outil pour la vérification automatique ? *Quelles sont ses capacités ?*

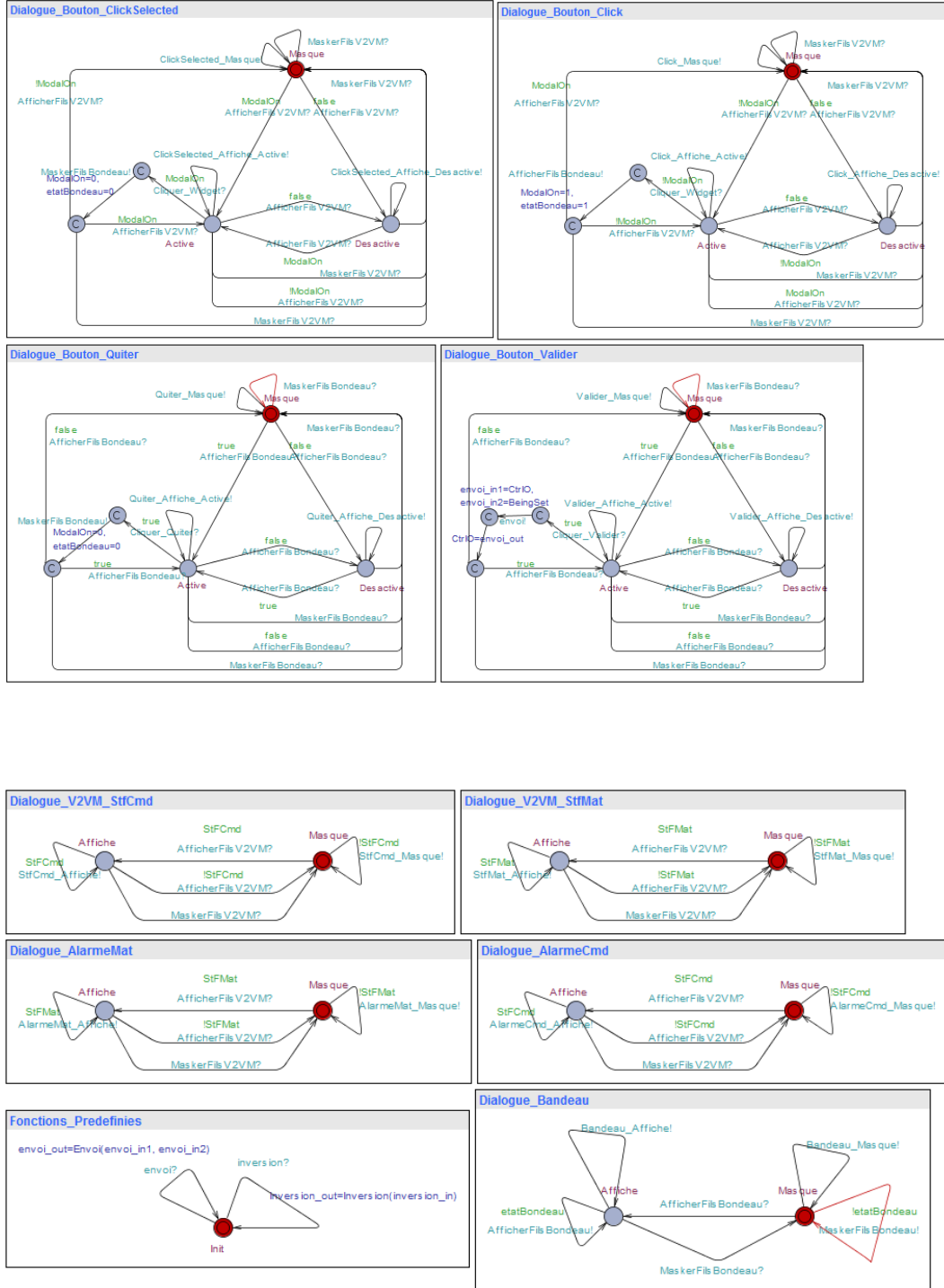


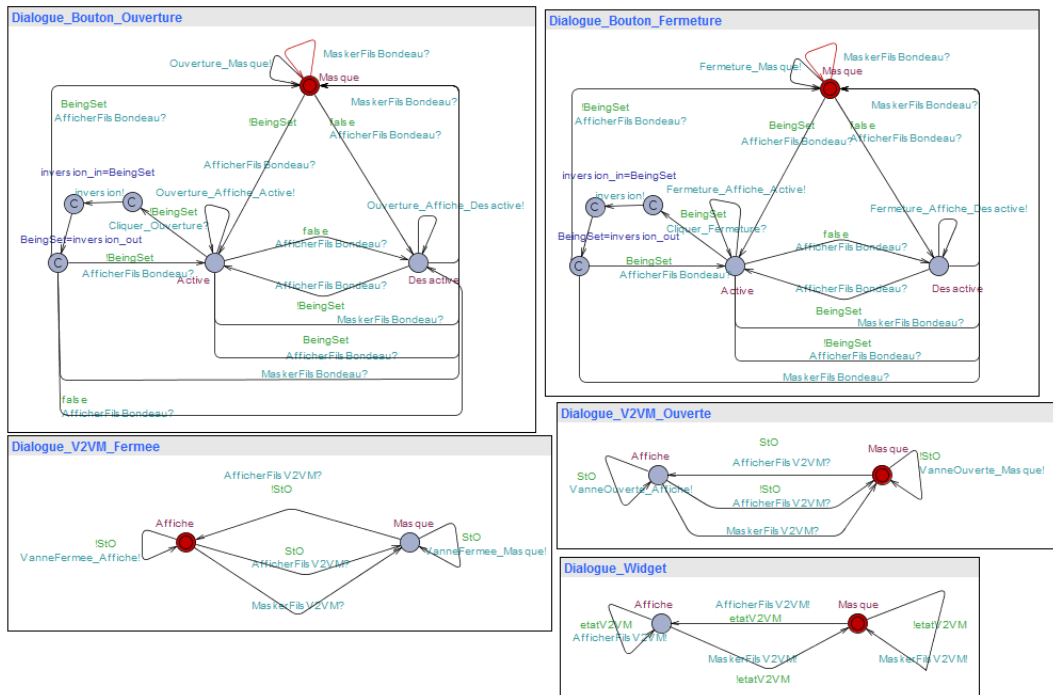
Modèle complet de V2VM

C.1 Tâches utilisateurs

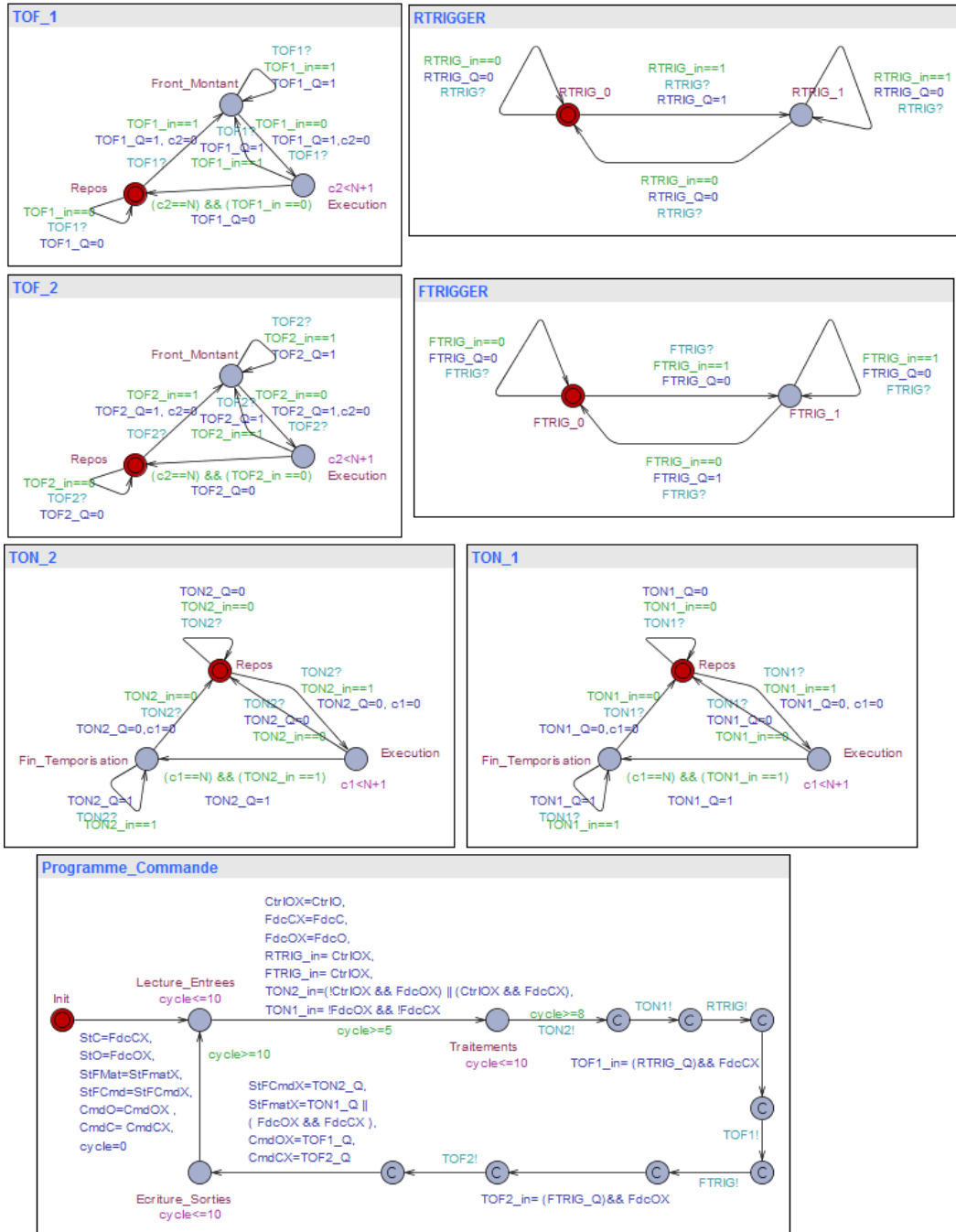


C.2 Interface de supervision

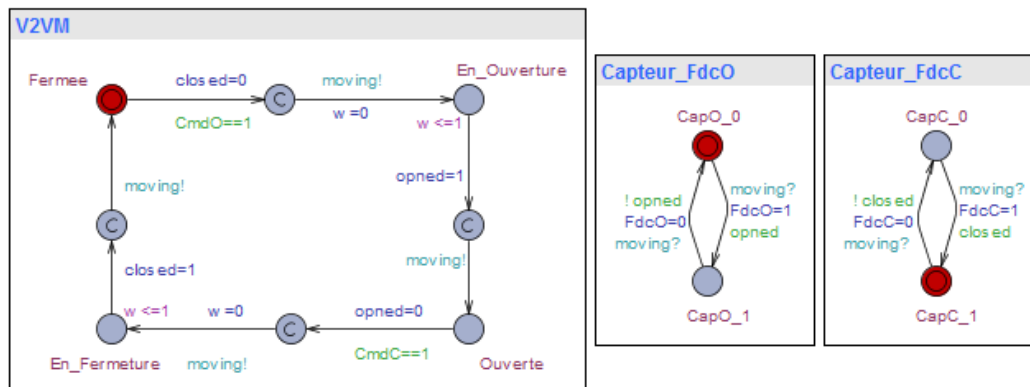




C.3 Programme de commande



C.3.1 Composant physique



C.4 Extrait d'un contre-exemple Alloy en XML

```

1 <alloy builddate="2012-09-25 15:54 EDT">
2 <instance bitwidth="0" maxseq="0" command="Check delUndoesAdd for 3" >
3 <sig label="seq/Int" ID="0" parentID="1" builtin="yes">
4 </sig>
5 <sig label="Int" ID="1" parentID="2" builtin="yes">
6 </sig>
7 ...
8 <field label="addr" ID="7" parentID="6">
9 <tuple> <atom label="Book0"/> <atom label="Name0"/> <atom label="Addr0"/> </tuple>
10 <types> <type ID="6"/> <type ID="4"/> <type ID="5"/> </types>
11 </field>
12
13 <skolem label="delUndoesAdd\_b" ID="8">
14 <tuple> <atom label="Book0"/> </tuple>
15 <types> <type ID="6"/> </types>
16 </skolem>
17 ...
18 </instance>
19
20 <source filename="EdS.als" content="module ..."/>
21 </alloy>

```

FIGURE C.1 – Extrait du contre-exemple en XML



Grammaire VB en xtext

```
grammar org.xtext.example.vb.Vb with org.eclipse.xtext.common.Terminals

generate vb "http://www.xtext.org/example/vb/Vb"

Model:
    (stmt+=Statement)+
;

Statement:
    (IfStatement|Declaration|FunctionCall)
;

IfStatement:
    'If' ifCondition=(Expression) 'Then'
    thenStmt=Block
    (=>'Else' elseStmt=Block)?
    'End If'
;

Declaration:
    var=Variable '=' expression=Expression
;

Variable:
    name=ID (=>' .Value' )? | =>'Parent.' name=ID
;

Block returns Statement:
    {Block}
    statement+=Statement+
;

Expression:
    (BooleanExpression | FunctionCall) ;

FunctionCall:
    =>name=FUNCTIONNAME =>" ("? arg+=Expression (=>"," arg+=Expression)* => ")"?
```

```

;

BooleanExpression returns Expression:
    AndExpression
    (({OrExpression.left=current} op=("Or")) right=AndExpression)*;

AndExpression returns Expression:
    Comparison
    (({AndExpression.left=current} op=("And")) right=Comparison)*
;

Comparison returns Expression:
    Equals
    (({Comparison.left=current} op("<" | ">" | "<=" | ">=") ) right=Equals)*;

Equals returns Expression:
    Addition
    (({Equals.left=current} => op="=" | {NotEquals.left=current} op("<>") right=Addition)*;

Addition returns Expression:
    Multiplication
    (({Plus.left=current} '+' | {Minus.left=current} '-')
     right=Multiplication)*;

Multiplication returns Expression:
    Prefixed (({MultiOrDiv.left=current} op("*"|" /") right=Prefixed)*;

Prefixed returns Expression:
    {BooleanNegation} =>"Not" expression=Atomic | /* right associativity */
    {ArithmeticSigned} => "-" expression=Atomic | /* right associativity */
    Atomic;

Atomic returns Expression:
    '(' Expression ')' |
    {NumberLiteral} value=INT |
    {StringLiteral} value=STRING |
    {BooleanLiteral} value=('True' | 'False') |
    {VariableReference} ref=Variable
;

terminal FUNCTIONNAME:
'PnSetValue' |
'PnEditValue' |
'PnGetValue' |
'PnIncValue' |
'PnInvertValue' |
'PnWaitConstant' |
'PnWaitProperty' |

```

```
' PnOpenView' |  
' PnHideChildView' |  
' PnCloseChildView'  
;
```


Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande : Application aux architectures SCADA

La conception des systèmes de contrôle-commande souffre souvent des problèmes de communication et d'interprétation des spécifications entre les différents intervenants provenant souvent de domaines techniques très variés. Afin de cadrer la conception de ces systèmes, plusieurs démarches ont été proposées dans la littérature. Parmi elles, la démarche dite *mixte* (ascendante/descendante), qui voit la conception réalisée en deux phases. Dans la première phase (ascendante), un modèle du système est défini à partir d'un ensemble de composants standardisés. Ce modèle subit, dans la deuxième phase (descendante), plusieurs raffinages et transformations pour obtenir des modèles plus concrets (codes, applicatifs, etc.). Afin de garantir la qualité des systèmes conçus par cette démarche, nous proposons dans cette thèse, deux approches de vérification formelle basées sur le Model-Checking. La première approche porte sur la vérification des composants standardisés et permet la vérification d'une chaîne de contrôle-commande élémentaire complète. La deuxième approche consiste en la vérification des modèles d'architecture (P&ID) utilisés pour la génération des programmes de contrôle-commande. Cette dernière est basée sur la définition d'un style architectural en Alloy pour la norme ANSI/ISA-5.1. Pour supporter les deux approches, deux flots de vérification formelle semi-automatisés basés sur les concepts de l'IDM ont été proposés. L'intégration des méthodes formelles dans un contexte industriel est facilitée, ainsi, par la génération automatique des modèles formels à partir des modèles de conception maîtrisés par les concepteurs métiers. Nos deux approches ont été validées sur un cas industriel concret concernant un système de gestion de fluide embarqué dans un navire.

Mots clés: Contrôle-Commande, Spécification, Modélisation, Vérification Formelle, Style architectural, IDM, Alloy, Automates Temporisés, CTL, Revue Systématique.

Integration of formal verification techniques into a control-command system design approach: Application to SCADA architectures

The design of control-command systems often suffers from problems of communication and interpretation of specifications between the various designers, frequently coming from a wide range of technical fields. In order to address the design of these systems, several methods have been proposed in the literature. Among them, the so-called *mixed method* (bottom-up/top-down), which sees the design realized in two steps. In the first step (bottom-up), a model of the system is defined from a set of standardized components. This model undergoes, in the second (top-down) step, several refinements and transformations to obtain more concrete models (codes, applications, etc.). To guarantee the quality of the systems designed according to this method, we propose two formal verification approaches, based on Model-Checking, in this thesis. The first approach concerns the verification of standardized components and allows the verification of a complete elementary control-command chain. The second one consists in verifying the model of architecture (P&ID) used for the generation of control programs. The latter is based on the definition of an architectural style in Alloy for the ANSI/ISA-5.1 standard. To support both approaches, two formal semi-automated verification flows based on Model-Driven Engineering have been proposed. This integration of formal methods in an industrial context is facilitated by the automatic generation of formal models from design models carried out by business designers. Our two approaches have been validated on a concrete industrial case of a fluid management system embedded in a ship.

Keywords: Control-Command, Specification, Modelling, Formal Verification, Architectural Style, MDE, Alloy, Timed Automatas, CTL, Systematic Mapping.



n d'ordre : 442

Université Bretagne Sud

Centre de Recherche Christiaan Huygens - rue de Saint Maudé - 56321 Lorient Cedex

Tèl : + 33(0)2 97 87 45 60