HAL
archives-ouvertes.fr

Les rôles : médiateurs dynamiques entre modèles système et modèles de simulation

Jean-Philippe Schneider

▶ To cite this version:

Jean-Philippe Schneider. Les rôles : médiateurs dynamiques entre modèles système et modèles de simulation. Autre [cs.OH]. Université de Bretagne occidentale - Brest, 2015. Français. NNT : 2015BRES0067 . tel-01986294

HAL Id: tel-01986294
https://tel.archives-ouvertes.fr/tel-01986294

Submitted on 18 Jan 2019

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.



THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

Mention : Sciences et Technologies de l'Information et de la Communication

Spécialité : Informatique

École Doctorale Santé, Information, Communication, Mathématiques, Matière

présentée par

Jean-Philippe SCHNEIDER

Préparée au Lab-STICC UMR CNRS 6285

École Nationale Supérieure de Techniques Avancées Bretagne

ENSTA Bretagne

Les rôles : médiateurs dynamiques entre modèles système et modèles de simulation

Thèse soutenue le 25 novembre 2015

devant le jury composé de :

Hans Vangheluwe
Professeur, University of Antwerp / *Président du Jury*

Vincent Chapurlat
Professeur, École des Mines d'Alès / *Rapporteur*

Luc Fabresse
Maître de conférence, École des Mines de Douai / *Rapporteur*

Alain Plantec
Professeur, Université de Bretagne Occidentale / *Examinateur*

Véronique Serfaty
DGA / *Examinatrice*

Joël Champeau
Enseignant-Chercheur, ENSTA Bretagne / *Encadrant*

Loïc Lagadec
Professeur, ENSTA Bretagne / *Directeur de thèse*

Eric Senn
Maître de conférence / HDR, Université de Bretagne Sud / *Directeur de thèse*

Licence

Cette oeuvre, création, site ou texte est sous licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 4.0 International. Pour accéder à une copie de cette licence, merci de vous rendre à l'adresse suivante http://creativecommons.org/licenses/by-nc-nd/4.0/ ou envoyez un courrier à Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.



Mais alors, dit Alice, si le monde n'a absolument aucun sens, qui nous empêche d'en inventer un ?

Les aventures d'Alice au pays des merveilles
LEWIS CARROLL

Remerciements

Je tiens à remercier madame Véronique Serfaty et messieurs Vincent Chapurlat, Luc Fabresse, Alain Plantec, Hans Vangheluwe, Joël Champeau, Loïc Lagadec et Eric Senn d'avoir accepté de faire partie de mon jury de thèse.

Je remercie particulièrement Vincent Chapurlat et Luc Fabresse d'avoir accepté de rapporter mes travaux ainsi que Hans Vangheluwe pour avoir accepté de présider mon jury de thèse.

Je remercie également Eric Senn pour avoir été mon directeur de thèse, Loïc Lagadec pour avoir été mon co-directeur de thèse et Joël Champeau pour le suivi de mes travaux.

Mes remerciements vont également à mes collègues de l'ENSTA Bretagne qui ont su m'accompagner durant ces dernières années. Je remercie tout particulièrement Bruno Aizier pour m'avoir fait découvrir le monde de l'ingénierie système. Je te remercie pour tes conseils et ton expertise qui m'ont beaucoup aidé dans mes travaux. Tu as également su relever le défi de me faire travailler sur des projets mécaniques (ce qui était un challenge en soi). Nous nous sommes aussi bien amusés avec SysML. Nos aventures avec SysML nous ont même amené à nous "promener" en avion alors que des rafales à 140km/h soufflaient au-dessus de Brest.

Je tiens aussi à remercier Pierre-Yves Pillain pour nos discussions techniques autour de la modélisation et du sens que l'on peut lui donner dans un contexte industriel et recherche. Il est vraiment dommage que nous n'ayons pas pu travailler plus longuement ensemble.

Je remercie également Pascale Gautron pour nos discussions autour de Java, des bases de données, des systèmes d'information, de l'enseignement et de la pédagogie. J'espère avoir bien retenu la leçon et que je pourrai employer le résultat de nos discussions dans ma future vie professionnelle. Je te remercie également pour tes conseils avisés.

Je remercie Zoé Drey pour les différentes relectures d'articles et les conseils de rédaction en anglais.

Je remercie chaleureusement les membres de l'équipe IDM Radio Logicielle pour nos discussions autour d'un café. Je remercie Michel Inizan pour les cours de golf en salle café et pour les carreaux de chocolat qui ont fait du bien à mon moral.

Ces remerciements ne pourraient pas être sans l'aide efficace et le soutien de Michelle Hofmann et d'Annick Billon-Coat.

Je remercie également toute l'équipe du Bureau du Soutien Informatique : Cédric Gaudron, Jean-Yves Le Roux, Philippe Bruno, Pierrick Veillon, Jean-Paul Kirkbride, Bruno Frédéric et

tous les autres. Merci pour votre bonne humeur, les discussions techniques et les batailles pour l'installation de Rhapsody. J'en profite pour remercier les personnels des différents services de l'ENSTA Bretagne avec qui j'ai eu plaisir de travailler. Je ne peux pas tous vous citer mais je suis sûr que vous vous reconnaîtrez.

J'adresse aussi mes remerciements aux membres du club badminton de l'association Escale de l'ENSTA Bretagne, merci pour les fous rires et les beaux échanges.

Je tiens également à remercier ici Arnaud Monégier du Sorbier et toute l'équipe de Thales (Danièle, Renaud, Benoit, Eric et tous les autres) avec laquelle j'ai fait une année de césure. Travailler avec vous m'a donné envie de concevoir et développer du bon logiciel. Cette thèse est donc aussi le fruit de mon (bref) passage parmi vous.

Je ne peux terminer ces remerciements (last but not least) sans remercier ma famille et surtout mes parents ainsi que mes amis qui ont su me soutenir pendant les années de ma thèse malgré mes périodes de mauvaise humeur.

Table des matières

1. Introduction **1**
- 1.1. Motivations : retour d'expérience sur le projet MeDON . . . 2
- 1.2. Ingénierie système et prototypage virtuel . . . 3
- 1.3. Contraintes liés aux systèmes de systèmes . . . 4
- 1.4. Problèmes . . . 5
- 1.5. Plan du mémoire . . . 5

2. Etat de l'art **7**
- 2.1. Introduction . . . 7
- 2.2. Ingénierie Système . . . 8
  - 2.2.1. La notion de système . . . 9
  - 2.2.2. Les Systèmes de Systèmes . . . 10
  - 2.2.3. L'ingénierie système . . . 11
  - 2.2.4. Un exemple de processus d'ingénierie Système : la norme ISO 15 288 . . . 12
- 2.3. Modélisation et Simulation . . . 15
  - 2.3.1. Model-Based Systems Engineering . . . 16
  - 2.3.2. Prototypage et simulation au service de l'ingénierie système . . . 20
  - 2.3.3. Passage de la modélisation conceptuelle à la spécification de la simulation 22
  - 2.3.4. Synthèse des approches basées sur l'ingénierie dirigée par les modèles . . . 28
- 2.4. Rôles . . . 33
  - 2.4.1. Définition et propriétés des rôles . . . 33
  - 2.4.2. Utilisation des rôles . . . 35
- 2.5. Positionnement . . . 40

3. L'environnement Role4All **43**
- 3.1. Introduction . . . 43
- 3.2. Architecture globale de Role4All . . . 45
  - 3.2.1. Rappel du contexte . . . 45
  - 3.2.2. Présentation de l'architecture générale . . . 46
  - 3.2.3. Justification du choix du concept de rôle . . . 48
  - 3.2.4. Justification du choix des parsers combinateurs . . . 48

3.3. Choix des caractéristiques des rôles . . . 49
3.3.1. Description et sélection des caractéristiques des rôles dans le contexte ingénierie système . . . 50
3.3.2. Utilisation des caractéristiques sélectionnées . . . 55
3.4. Langage de rôle . . . 65
3.4.1. Définitions préliminaires . . . 66
3.4.2. Structure d'un rôle . . . 67
3.4.3. Relation entre rôle et objet . . . 68
3.4.4. Contexte pour les rôles . . . 73
3.5. Utilisation d'importateurs modulaires à base de parsers combinateurs . . . 75
3.5.1. Opérateurs de combinaison de parsers . . . 76
3.5.2. Modularité apportée par les parsers combinateurs . . . 77
3.5.3. Intégration des parsers combinateurs . . . 78
3.6. Conclusion . . . 80

4. Réalisation de l'environnement Role4All 83
4.1. Introduction à la mise en œuvre . . . 83
4.2. Analyse . . . 84
4.2.1. Analyse fonctionnelle . . . 86
4.2.2. Analyse physique . . . 88
4.3. Définition à la volée de rôles . . . 89
4.3.1. Description du processus de définition et d'allocation des rôles . . . 90
4.3.2. Implémentation de la création des types de rôles . . . 91
4.3.3. Implémentation de l'allocation des rôles . . . 93
4.4. Définition à la volée des adaptateurs . . . 96
4.4.1. Processus de définition des adaptateurs . . . 96
4.4.2. Implémentation des adaptateurs dynamiques . . . 97
4.5. Rôles et contextes . . . 100
4.5.1. Définition d'un contexte . . . 100
4.5.2. Mise en place du contexte . . . 101
4.5.3. Sérialisation d'un contexte . . . 102
4.6. Implémentation d'importateurs modulaires avec des parsers combinateurs . . . 103
4.6.1. Processus de définition d'importateurs modulaires à base de parsers combinateurs . . . 103
4.6.2. Le framework PetitParser . . . 104
4.6.3. Implémentation des importateurs . . . 107
4.7. Réalisation d'une interface dédiée . . . 108
4.8. Conclusion . . . 111

5. Validation dans le contexte Système de Systèmes 113
5.1. Introduction . . . 114
5.2. Contexte du cas d'étude . . . 114
5.2.1. Description de l'observatoire sous-marin MeDON . . . 116
5.2.2. Modélisation des sous-systèmes d'un observatoire . . . 119
5.2.3. Problèmes soulevés . . . 122
5.3. Import de modèles hétérogènes . . . 125
5.3.1. Définition des représentations internes des méta-modèles . . . 125

5.3.2. Analyse des formats de sérialisation des modèles . . . 128
5.3.3. Définition des parsers pour importer les modèles sérialisés . . . 129
5.3.4. Intégration des parsers à l'environnement Role4All . . . 133
5.4. Manipulation de modèles hétérogènes . . . 134
5.4.1. Analyse de l'outillage d'exécution de modèles . . . 135
5.4.2. Définition des types de rôles . . . 138
5.4.3. Définition des associations des rôles avec les éléments de modèles . . . 140
5.4.4. Définition des adaptateurs . . . 141
5.4.5. Définition d'un contexte . . . 143
5.4.6. Exécution du modèle . . . 144
5.5. Adaptation du modèle de rôles pour permettre l'utilisation d'un nouvel environnement de simulation . . . 146
5.5.1. Analyse de l'outil de simulation de réseau OMNeT++ . . . 146
5.5.2. Modèle de rôles associé à des rôles . . . 148
5.5.3. Association de rôles avec des rôles . . . 149
5.5.4. Ecriture du visiteur . . . 150
5.5.5. Simulation obtenue . . . 151
5.6. Adaptation à un remplacement d'outil de simulation . . . 153
5.6.1. Présentation du simulateur ADEVS . . . 153
5.6.2. Ecriture d'un visiteur pour la création des éléments de modèle ADEVS . 154
5.6.3. Ecriture d'adaptateurs dynamiques . . . 155
5.7. Adaptation à un changement dans les langages de modélisation . . . 156
5.7.1. Modification du méta-modèle du langage SysML et du format de sérialisation . . . 157
5.7.2. Impact sur le parser pour le langage de modélisation . . . 158
5.7.3. Impact sur le modèle de rôles . . . 159
5.8. Génération de documentation . . . 161
5.8.1. Analyse du générateur de documentation . . . 161
5.8.2. Définition du modèle de rôle et des adaptateurs dynamiques . . . 162
5.9. Conclusion . . . 163

6. Conclusion **165**
6.1. Bilan du travail . . . 165
6.1.1. Analyse des problématiques . . . 165
6.1.2. Role4All : les rôles pour créer un point de vue homogène sur des modèles hétérogènes . . . 167
6.1.3. Limitations de Role4All . . . 169
6.2. Perspectives à court terme . . . 170
6.3. Perspectives à moyen terme . . . 170
6.4. Perspectives à long terme . . . 171
6.5. Conclusion . . . 171

A. Role4All utilisé en parallèle de Pharo **173**

B. Modèle SysML sérialisé pour les hydrophones **175**

C. Modèle eFFBD sérialisé pour le système d'informations de l'observatoire MeDON **181**

Bibliographie 185

Table des figures

2.1. Idenfication des processus techniques du standard ISO 15288 13
2.2. Roue de Demming décrivant le cycle d'un processus d'amélioration continue. . . 14
2.3. Illustration des différents modèles définis par le MDA et l'utilisation des transformations de modèles. 24
2.4. Illustration de la définition de l'approche par intégration. 29
2.5. Illustration de la définition de l'approche par unification. 30
2.6. Illustration de la définition de l'approche par fédération. 30

3.1. Présentation de l'outillage proposé 47
3.2. Modèle SysML présentant le bloc itsHydrophone comme émetteur d'un flux signal et le bloc itsItSystem comme récepteur du flux 55
3.3. Modèle eFFBD présentant la fonction Hydrophone comme émetteur d'un flux Signal et la fonction ItSystem comme récepteur du flux 56
3.4. Association des rôles Emitter et Receiver sur des objets définis dans les langages SysML et eFFBD. 57
3.5. Rôles Emetteur et Recepteur possèdant un attribut meanTimeBetweenFailures pour pallier à son absence dans les blocs modélisés dans le langage SysML de l'exemple 58
3.6. Rôle Subsystem associé aux blocs SysML en plus des rôles Emiter et Receiver. . 59
3.7. L'objet itsHydrophone joue deux fois le rôle d'Emitter. 59
3.8. Bloc SysML itsHydrophone acquérant et abandonnant le rôle Receiver 60
3.9. Application du rôle Agent sur les rôles Emetteur et Récepteur appliqués à des blocs SysML 61
3.10. Comportement d'un rôle comme interface d'accès pour un élément de modèle. . 61
3.11. Spécialisation du rôle Emitter par le rôle ContinuousEmitter. 62
3.12. Rôles Emitter et Receiver associés aux blocs SysML possèdant un port de sortie et d'entrée. 63
3.13. Définition du rôle Connection qui regroupe les rôles Emitter et Receiver associés aux blocs itsHydrophone et itsItSystem. 64
3.14. Méta-modèle complet du langage de rôle. 66

3.15. Méta-classe Role possèdant des propriétés. 67
3.16. Relation *ownsFrom* modélisant le partage de propriétés entre rôles. 68
3.17. Relation d'agrégation définie sur la méta-classe Role. 68
3.18. Association entre les méta-classes Role et Type. 69
3.19. Ajout de multiplicité pour permettre à un élément de modèle de jouer plusieurs rôles 69
3.20. Héritage entre les méta-classes Player et Role pour que des rôles puissent jouer des rôles. 70
3.21. Attribut uniqueIdentifier fournissant une identité propre à toutes les instances de rôles et d'éléments de modèles 71
3.22. Réification de la relation entre les méta-classes Player et Role et ajout de la méta-classe RoleAdapter fournissant les comportements d'interprétation des rôles. 72
3.23. Illustration du patron chaine de responsabilité. 72
3.24. Ajout de méthodes pour spécifier la relation entre Role et Player 73
3.25. Diagramme de classe définissant la structure d'un contexte de rôles 74
3.26. Description de la structure des règles d'association des rôles et des adaptateurs dynamiques. 75
3.27. Exemple simple de diagramme de bloc interne SysML. 76
3.28. Identification des impacts sur les parsers 78
3.29. Architecture pour l'utilisation de l'introspection pour faciliter l'intégration d'importateurs afin d'importer des modèles. 79

4.1. Objectifs de la réalisation informatique de l'environnement Role4All. 85
4.2. Identification des acteurs et des fonctions d'un outillage basé rôle pour l'exécution de modèles hétérogènes 86
4.3. Processus de définition et d'allocation des rôles 90
4.4. Méta-modèle complet du langage de rôle de l'environnement Role4All 91
4.5. Héritage entre classe de types de rôle et la classe Role 92
4.6. Processus pour définir les adaptateurs dynamiques et leur comportement. . . . 97
4.7. Processus de définition des parsers 103
4.8. Diagramme de classe pour l'implémentation du parser de la version allégée du langage SysML. 105
4.9. Diagramme de classe de l'implémentation de parsers avec actions pour la version allégée du langage SysML. 106
4.10. Interface graphique de l'outil dédié implémenté 109
4.11. Relations de dépendances entre les classes RoleGui, RoleOrganizer et RoleContext 109
4.12. Dépendances entre la définition de l'interface Role4All et les parsers servant à l'import de modèles. 110

5.1. Structure générale d'un observatoire sous-marin. 116
5.2. Identification des acteurs impliqués dans la réalisation d'un observatoire sous-marin. 117
5.3. Block Definition Diagram correspondant à un extrait de l'analyse fonctionnelle d'un hydrophone déployé dans le cadre du projet MeDON. 119
5.4. Internal Block Diagram correspondant à l'analyse fonctionnelle des hydrophones déployés dans le cadre du projet MeDON. 120

5.5. Modèle eFFBD illustrant l'architecture fonctionnelle du système d'information de l'observatoire sous-marin MeDON. 121
5.6. Situation problématique du projet MeDON 123
5.7. Utilisation de Role4All pour l'observatoire MeDON 124
5.8. Version adaptée du méta-modèle SysML concernant la définition d'un diagramme de blocs. 127
5.9. Description de la version adaptée de la partie du méta-modèle du langage SysML décrivant les diagrammes de blocs internes. 127
5.10. Méta-modèle simplifié pour le langage eFFBD. 128
5.11. Héritage entre les classes SysmlImporter et EffbdImporter et ModelImporter. . 133
5.12. Méta-modèle pour la description d'architectures fonctionnelles. 136
5.13. Architecture du moteur d'exécution basé acteurs. 137
5.14. Définition des rôles orientés architecture fonctionnelle et association avec un contexte au sein de notre outillage. 139
5.15. Affichage obtenu après l'initialisation du moteur d'exécution avec les rôles. . . . 144
5.16. Extrait du diagramme de séquence obtenu lors de l'exécution du modèle d'architecture fonctionnelle. 145
5.17. Partie du méta-modèle d'OMNeT++ décrivant la structure d'une simulation. . 147
5.18. Partie du méta-modèle d'OMNeT++ décrivant la structure d'un réseau. 147
5.19. Extrait de l'export GraphViz de l'allocation des rôles décrivant un système DEVS. 150
5.20. Illustration du résultat attendu de la création du modèle OMNeT++ 150
5.21. Résultat de la génération de la simulation à partir des instances des éléments OMNeT++. 152
5.22. Graphique de l'activité du module ReadEntry1. 152
5.23. Modèle du *binding* Java pour le simulateur ADEVS (réalisé d'après la documentation en ligne). 154
5.24. Diagramme de classes pour les visiteurs générant des simulations basées événements discrets. 155
5.25. Modification du méta-modèle du langage SysML pour inclure un diagramme d'activités. 157
5.26. Impact sur les parsers de l'ajout d'un lien vers un diagramme d'activités dans la description d'un bloc SysML. 159

6.1. Structure de l'environnement Role4All 168

A.1. Role4All peut être utilisé en même temps que tous les outils Smalltalk natifs disponibles dans l'environnement Pharo 173

1 Introduction

L'ingénierie système est un ensemble de méthodes pour permettre de concevoir, développer, déployer et maintenir des systèmes complexes. Un système complexe se caractérise notamment par :

— l'implication d'équipes pluri-disciplinaires,
— des échanges entre les différentes équipes,
— une évolutivité forte au cours du cycle de vie du système.

L'utilisation d'abstractions, comme des modèles, facilite la communication en permettant de se concentrer sur les points essentiels du système par rapport à un domaine donné. Ces modèles peuvent être analysés et exécutés, par exemple par des techniques de simulation.

Cependant, la pluri-disciplinarité et l'évolutivité des systèmes complexifient l'exécution des modèles systèmes du fait de :

— l'hétérogénéité des langages de modélisation système,
— l'hétérogénéité des formalismes de simulation,
— l'évolutivité des langages de modélisation système.

La problématique est, alors, de pouvoir modéliser, indépendamment des langages de modélisation système et des formalismes de simulation hétérogènes, un point de vue dynamique cohérent sur les modèles systèmes pour simuler ces derniers.

Dans le monde des bases de données, le concept de rôle a été introduit pour permettre de modéliser un point de vue sur les données stockées dans une base de données indépendamment de leur type.

Nous avons réutilisé le concept de rôle afin de créer des points de vue homogènes, basés sur les formalismes de simulation, sur des modèles systèmes hétérogènes. Nous avons créé un environnement, nommé Role4All, qui permet d'associer ou de dissocier des rôles avec des éléments de modèles. Nous avons utilisé Role4All sur des modèles systèmes, exprimés dans les langages SysML et eFFBD, issus du projet d'observatoire MeDON. Nous avons interprété l'architecture fonctionnelle décrite par ces modèles et créé des simulations basées événements discrets à partir de ces modèles. Nous avons également utilisé la capacité d'associer ou de dissocier dynamiquement des rôles pour gérer des évolutions dans la définition des modèles systèmes d'entrée.

1.1. Motivations : retour d'expérience sur le projet MeDON

Avant mon doctorat, j'ai travaillé pendant deux ans, dans le cadre d'un projet européen d'observatoire sous-marin côtier câblé : le projet MeDON [61]. Un observatoire sous-marin est un système de recueil et d'analyse de données sur l'environnement sous-marin. L'objectif de ce projet était d'installer un ensemble de capteurs sous-marins pour observer la population de dauphins installée à proximité de Brest. Une infrastructure pour collecter et traiter ces données a été déployée dans le parc marin d'Iroise, plus précisément sur l'île de Molène. Un système d'information a été mis en place à Brest pour diffuser les données vers des utilisateurs tiers.

L'observatoire MeDON est un système complexe du fait de la diversité des technologies utilisées mais aussi à cause de l'implication d'équipes provenant d'entreprises et de domaines de compétence différents. Pour illustrer la complexité issue de la variété des domaines, les interactions entre experts des domaines de la mécanique, du traitement du signal, de l'électronique et de l'informatique peuvent être décrites.

Les experts du domaine de la mécanique étaient responsables du déploiement de l'ensemble de l'infrastructure sous-marine de l'observatoire (câbles, capteurs, infrastructure réseau immergée) et de sa résistance à la corrosion.

Les experts de l'acoustique sous-marine avaient à élaborer des algorithmes de traitement du signal utilisés pour traiter les données recueillies par les capteurs.

Les experts en électronique étaient également impliqués dans le projet pour la conception des capteurs. Pour réaliser cette tâche, les électroniciens devaient travailler de concert avec les experts mécaniciens pour concevoir des capteurs résistant à la pression de l'eau.

Pour les scientifiques, les capteurs devaient acquérir les signaux d'intérêts à la fréquence d'échantillonnage nécessaire pour l'étude des dauphins. Or, les capteurs étaient installés en mer, dans une zone isolée. L'alimentation électrique des capteurs était limitée. Ainsi, la conception d'un capteur fut un compromis entre la fréquence d'échantillonnage de l'acquisition des sons sous-marins, consommatrice en énergie, et l'économie de cette dernière. Les capteurs étaient également branchés à un système d'information réalisé par des informaticiens. Les informaticiens devaient intégrer les différents types de capteurs tout en tenant compte, lors de la conception du système d'information, de l'utilisation future de nouveaux capteurs. Les informaticiens devaient aussi travailler avec les scientifiques pour que les traitements les plus lourds soient réalisés directement dans le système d'information. En effet, l'isolement géographique de l'observatoire faisait que le lien réseau entre l'observatoire et la partie du système d'information basée sur le continent n'avait pas un débit suffisant pour transporter, dans des délais satisfaisants, les données brutes acquises par les capteurs. Les traitements n'étaient, cependant, pas destinés à être utilisés dans un système d'information.

Enfin, un observatoire sous-marin requiert beaucoup d'efforts de maintenance. Le fait d'installer l'observatoire près des côtes rendait l'observatoire vulnérable aux activités humaines comme la pêche. Par exemple, des chaluts pouvaient endommager le câble alimentant les capteurs. Cela nécessitait de faire intervenir une équipe de plongeurs et, dans le pire des cas, un navire spécialisé dans la pose de câbles sous-marin. De plus, des plongeurs devaient périodiquement plonger pour nettoyer les capteurs qui sont recouverts d'algues.

L'expérience acquise lors du projet MeDON m'a permis de constater que la diversité des domaines d'expertise est une source d'incompréhensions entre les partenaires du projet. Chaque domaine possède son propre point de vue sur l'observatoire. Ces différentes visions s'étaient notamment confrontées au moment du déploiement. Il fallait notamment répondre à la question de la localisation de l'observatoire. Fallait-il installer l'observatoire dans un endroit à haute

valeur scientifique mais présentant des difficultés d'installation ? Fallait-il privilégier la facilité d'installation et de maintenance par rapport aux objectifs scientifiques ? Une autre question concernait la place de l'ingénierie logicielle dans le projet. Le logiciel n'était il qu'un outil pour un observatoire, ou avait il une place plus importante ? Au vu des coûts engagés dans les autres domaines (coût de la réalisation des structures immergées, coût des capteurs, coût du déploiement), le logiciel avait un poids plus faible dans le budget et par conséquent était considéré comme une variable d'ajustement dans le projet. Cependant, si le logiciel ne fonctionnait pas, tout le projet aurait été en échec. En effet, il aurait été impossible de récupérer les données, de les traiter et de les diffuser alors qu'il s'agissait des objectifs principaux du projet.

Malgré tous les problèmes rencontrés lors de la conception, l'observatoire MeDON fut déployé et mis en service avec succès [60]. Le bilan de ce projet a mis en avant le besoin de coordonner les actions des experts des différents domaines. Cela passe notamment par une meilleure compréhension de l'observatoire dès les premières phases de sa conception et le partage de l'information et de la vision du système entre les différentes équipes.

1.2. Ingénierie système et prototypage virtuel

Pour remédier à ce problème de partage d'information entre les partenaires du projet, je me suis intéressé à l'ingénierie système. L'ingénierie système est une approche pluri-disciplinaires. Son intention est de permettre de faire travailler ensemble des experts de différents domaines en les traitant de manière égale. Les compromis à faire pour la conception du système sont faits en fonction des besoins exprimés par les utilisateurs du système. De plus, l'ingénierie système définit plusieurs niveaux d'abstraction sur le système. Il est ainsi possible de se concentrer sur les problèmes importants à un instant donné dans le cycle de conception du système.

Pour faciliter la communication entre les différents partenaires, l'utilisation de modèles est un atout. Un modèle fournit une abstraction du système modélisé. Il peut donc être compris par des non-spécialistes du domaine du système modélisé. De ce fait, il est possible pour des experts d'un domaine de faire comprendre leurs problématiques à des experts d'un autre domaine. Cette compréhension mutuelle permet le débat et la recherche d'un compromis qui réponde aux exigences de toutes les parties. Exécuter le modèle du système permet de comprendre les fonctions du système qui ont été modélisées.

Pour voir le système fonctionner, un prototype du système peut être créé. Cependant, pour un observatoire sous-marin, le matériel à déployer est, au moins en partie, du matériel spécifique. Par conséquent, le matériel est conçu et développé comme un sous-système. Il n'est donc pas possible d'utiliser ce matériel pour créer un prototype durant les premières phases de conception de l'observatoire. Pour les matériels achetés sur étagère, leur coût implique d'en acheter un nombre restreint. Or dans une approche de prototypage, plusieurs alternatives sont comparées. Il faut donc être capable de construire plusieurs prototypes en parallèle. Le coût des matériels rend cette approche impossible.

Nous avons donc dû construire des prototypes virtuels du système, c'est-à-dire un modèle informatique du système qui est simulé. L'exécution des prototypes virtuels améliore la communication autour du système. En effet, en observant le système virtuel fonctionner, il est possible de faire comprendre son fonctionnement à tous les partenaires du projet et ainsi les impliquer dans le projet. Des prototypes virtuels simulés peuvent également être utilisés à tous les niveaux d'abstraction du système. L'implication de tous les acteurs permet alors de corriger au plus tôt les problèmes de conception qui proviennent d'incompréhension entre les différents partenaires.

L'exécution du système virtuel permet également de mieux comprendre les contraintes des différents domaines qui s'exercent sur le système. Cette communication et cette compréhension mutuelle permet de trouver plus facilement des compromis entre les différents partenaires.

1.3. Contraintes liés aux systèmes de systèmes

L'observatoire MeDON est un système dont les sous-systèmes sont également des systèmes. Les constituants de l'observatoire sont gérés par des équipes différentes, peuvent fonctionner indépendamment les uns des autres, évoluent, collaborent et sont distribués géographiquement. L'observatoire MeDON est donc un système de systèmes.

Dans le cycle de conception d'un système, l'architecture fonctionnelle, la définition de ce que le système va faire, fait le lien entre les besoins des clients et la solution technique à définir pour le système. L'architecture fonctionnelle apporte une réponse aux besoins des clients sans entrer dans les détails techniques de la solution. Il s'agit donc d'une phase se prêtant bien à la communication entre des experts de domaines différents et travaillant sur des systèmes différents d'un système de systèmes. L'exécution de modèles d'architecture fonctionnelle permet également aux différents partenaires de visualiser dynamiquement les dépendances entre les fonctions. Un partenaire peut appréhender l'objectif des fonctions réalisées par les autres partenaires ainsi que les relations que ses fonctions entretiennent avec celles des autres partenaires. Cependant, les modèles systèmes sont réalisés dans des langages de modélisation dédiés et les modèles de simulation dans d'autres langages de modélisation. Pour assurer la cohérence entre les modèles système et les modèles simulés, il faut être capable de créer les modèles de simulation directement à partir des modèles système. Dans le contexte de la conception d'un système plusieurs contraintes sont à prendre en compte.

Tout d'abord, plusieurs langages de modélisation système peuvent être utilisés. Chaque système d'un système de systèmes est réalisé avec son langage de modélisation. Les informations contenues dans les différents modèles doivent être utilisées conjointement et en cohérence pour créer les modèles de simulation. Il faut donc une capacité à travailler avec des modèles hétérogènes.

Ensuite, les modèles systèmes sont sous la responsabilité d'équipes différentes. Les modèles réalisés par ces équipes ne peuvent pas être modifiés, d'autant plus qu'ils concernent des domaines requérant un niveau d'expertise élevé. Ces modèles sont réalisés dans des langages de modélisation avec des outils qui ont été sélectionnés par les équipes en charge de la réalisation des modèles. Il n'est donc pas possible non plus de modifier la définition des langages de modélisation. En effet, les outils de modélisation reposent sur la définition des langages. Par conséquent, modifier la définition d'un langage implique de modifier l'outillage. Cela présente un risque de rejet par les équipes en charge de la conception des modèles. Dans certains domaines, les langages de modélisation sont standardisés. Modifier la définition du langage de modélisation brise le respect du standard. La variété des domaines d'expertise impliqués dans la conception d'un système implique la variété des outils de simulation. Il faut ainsi être capable de générer plusieurs types de simulation à partir des modèles système.

Les systèmes de systèmes ont également une longue durée de vie. Les langages de modélisation utilisés pour les concevoir vont donc évoluer. Il faut être capable d'accompagner ces évolutions. Il en est de même pour les outils de simulation. Il faut ainsi être également capable de s'adapter à des changements d'outils de simulation.

1.4. Problèmes

Dans le cadre de la conception d'un système, le passage de modèles système à des modèles de simulation, sous les contraintes explicitées, est important.

Un premier problème est de créer automatiquement les modèles de simulation à partir des modèles système. Cela évite les incohérences entre les deux types de modèles. Cela augmente le niveau de qualité du processus de création des prototypes virtuels. La création d'au moins une partie du modèle de simulation est un gain de temps et donc un gain de productivité. Elle permet également au modeleur de la simulation de se concentrer sur l'objectif de la simulation plutôt que sur l'extraction de l'information pertinente pour lui qui est cachée dans les modèles système.

Un deuxième problème est de donner la capacité aux experts métiers de continuer à travailler avec leurs modèles et leurs outils de modélisation. Ils n'ont pas à essayer de comprendre des outils de simulation qui sont en dehors de leur domaine d'expertise. Ils sont ainsi capables de réaliser des modifications rapides de leurs modèles et de les simuler dans la foulée. Ils sont alors capable d'observer immédiatement les effets des modifications réalisées sur leurs modèles.

Un troisième problème est de gérer l'hétérogénéité des langages de modélisation, utilisés pour modéliser les différents systèmes du système de systèmes, et des formalismes de simulation, utilisés pour réaliser les différentes exécutions des modèles. Pour pouvoir simuler les modèles du système, il est nécessaire de créer et de maintenir une cohérence, une unification, entre les concepts des différents langages de modélisation.

Un quatrième problème est le besoin de prendre en compte les évolutions au cours du temps des langages de modélisation et des outils de simulation. Prendre en compte ces facteurs permet de garantir la pérennité de l'utilisation des modèles. En effet, de nouveaux modèles réalisés dans de nouveaux langages de modélisation doivent pouvoir être intégrés. Mais, les modèles anciens réalisés dans des outils de modélisation obsolètes et qui ne sont plus disponibles doivent toujours pouvoir être utilisés. Il en est de même pour les outils de simulation où de nouveaux peuvent être mis en service. Il faut ainsi pouvoir générer de nouvelles simulations pour ces outils sans perdre la capacité de créer des simulations pour les outils déjà utilisés. La gestion de l'obsolescence des outils de simulation implique la mise en place d'une politique dédiée et la préparation en amont de l'utilisation d'un nouvel outil de simulation sans perdre la capacité de travailler avec l'outil de simulation devenant obsolète.

1.5. Plan du mémoire

Ces différents problèmes ont motivé les travaux décrits dans le présent mémoire. Le chapitre 2 présente l'approche ingénierie système pour mettre en lumière les contraintes précédemment citées. Ce chapitre présente également les bénéfices de la simulation ainsi que les approches existantes, telles que l'intégration, l'unification ou la fédération de modèle, pour faire de l'unification sémantique afin de permettre de passer de modèles de haut niveau comme les modèles système à des modèles de simulation. Cependant, par rapport aux contraintes liées au contexte ingénierie système, ces approches ont des limitations notamment pour s'adapter aux évolutions des langages de modélisation. Pour répondre à ces limitations, un concept de modélisation défini dans le cadre des base de donnée et étendu aux langages de modélisation et aux chaînes d'outils est repris : la notion de rôle.

Le chapitre 3 introduit notre utilisation du concept de rôle dans un environnement de modé-

lisation dédié que nous avons appelé Role4All. Nous avons sélectionné des propriétés des rôles en accord avec les contraintes du contexte ingénierie système. Ces propriétés nous ont servi à définir un méta-modèle pour un langage de rôle spécifique à notre contexte. Comme l'environnement Role4All est externe aux outils de modélisation, nous avons défini des importateurs dynamiques de modèles en utilisant des parsers combinateurs pour leurs aspects modulaires.

Le chapitre 4 présente une implémentation de l'environnement Role4All en Smalltalk. Nous avons implémenté le méta-modèle de rôle et nous lui avons ajouté un langage dédié pour permettre la création et la manipulation de rôles au travers d'un DSL interne. Les importateurs dynamiques ont également été implémentés.

Le chapitre 5 revisite le projet MeDON et montre l'intérêt d'utiliser l'environnement Role4All sur un système de systèmes. Nous démontrons notamment l'import de modèles hétérogènes dans l'environnement et l'utilisation des rôles pour manipuler ces différents modèles. L'exemple est aussi utilisé pour montrer comment les rôles et les importateurs modulaires à base de parsers combinateurs permettent de gérer les évolutions des langages de modélisation et des outils de simulation.

Le chapitre 6 dresse un bilan du travail de thèse et liste des perspectives pour Role4All.

2

Etat de l'art

Sommaire

2.1. Introduction 7
2.2. Ingénierie Système 8
2.2.1. La notion de système 9
2.2.2. Les Systèmes de Systèmes 10
2.2.3. L'ingénierie système 11
2.2.4. Un exemple de processus d'ingénierie Système : la norme ISO 15 288 . 12
2.3. Modélisation et Simulation 15
2.3.1. Model-Based Systems Engineering 16
2.3.2. Prototypage et simulation au service de l'ingénierie système 20
2.3.3. Passage de la modélisation conceptuelle à la spécification de la simulation 22
2.3.4. Synthèse des approches basées sur l'ingénierie dirigée par les modèles . 28
2.4. Rôles 33
2.4.1. Définition et propriétés des rôles 33
2.4.2. Utilisation des rôles 35
2.5. Positionnement 40

2.1. Introduction

La complexité des systèmes ne cesse d'augmenter. Cette complexité est, en partie, technique avec l'intégration et le développement de nouvelles technologies. Par exemple, dans le domaine aéronautique, la conception d'un avion durant la seconde guerre mondiale nécessitait des compétences en mécanique et en électronique. L'introduction de l'informatique a permis de soulager le pilote d'un grand nombre de tâches. Cependant, cette évolution a impliqué la nécessité d'avoir de nouvelles compétences pour concevoir un avion. Avec le temps, de plus en plus de fonctions sont automatisées par ordinateur. L'augmentation du nombre de lignes de code contribue à la complexification des systèmes. Cette complexification se retrouve également au niveau de la gestion de projet. L'exemple de l'Airbus A380 est frappant de ce point de vue. Les différentes parties de l'A380 sont produites dans différents pays. L'assemblage final requiert de rassembler

les différentes parties en respectant un calendrier précis. L'organisation du travail et de la logistique sont des facteurs clés de la réussite du projet et doivent être intégrées à l'organisation globale du projet de l'A380.

De nombreux facteurs d'échecs ont été identifiés dans les "Chaos Report" du Standish Group [58] dont le manque d'implication des utilisateurs, des exigences mal exprimées et changeantes, etc. La mise en place de processus et de techniques qui permettent de réduire les risques pour les projets complexes est un enjeu économique. L'ingénierie système définit un ensemble de processus et de méthodes pour remplir cet objectif. L'ingénierie système doit être outillée et doit permettre la communication entre les différentes parties prenantes du système. Cette communication permet d'échanger avec les clients et aider à préciser et stabiliser les exigences du système. Elle permet également aux différentes équipes en charge de la conception du système de bien comprendre le système en cours de développement.

L'utilisation de modèles permet d'échanger des informations entre partenaires en utilisant un support pour les activités de conception qui permet la visualisation et l'analyse des éléments conçus. L'utilisation de modèle permet également de pérenniser la connaissance du système. La communication par modèle est renforcée par l'exécution des modèles à l'aide de techniques de simulation. Or, les outils de simulation utilisent leurs propres langages de modélisation pour spécifier des simulations. Ces langages sont différents de ceux utilisés pour la modélisation système. Les modèles système doivent, pourtant, servir de base à l'élaboration des modèles de simulation. Modèles système et modèles de simulation ne sont pas au même niveau d'abstraction. Les modèles système sont conceptuels alors que les modèles de simulation sont concrets pour permettre leur exécution.

La variété des points de vue à adopter sur le système et l'implication d'équipes et d'organisation différentes dans la conception d'un système résultent en une grande variété des langages de modélisation système utilisés. Les objectifs des analyses réalisées sur les modèles sont également très divers. Un seul simulateur ne peut pas couvrir tous les objectifs d'analyse. Ainsi, plusieurs simulateurs sont utilisés pour analyser les modèles. La différence de niveaux d'abstraction, la variété des langages de modélisation système utilisés et la diversité des simulateurs utilisés pour l'analyse des modèles rendent la transition entre les modèles système et les modèles de simulation difficile.

Le présent chapitre a pour objectif :

- de mettre en lumière les facteurs de complexité des systèmes et des systèmes de systèmes et de leur ingénierie (section 2.2),
- de présenter l'utilisation de la modélisation et de la simulation pour l'ingénierie système et les approches actuelles d'intégration, d'unification ou de fédération de modèles pour l'utilisation de modèles systèmes pour spécifier des simulations ainsi que leurs limitations dans un contexte ingénierie système (section 2.3),
- de présenter une approche alternative pour le passage des modèles système aux modèles de simulation en utilisant des modèles de rôles et les limites de l'utilisation actuelle des rôles par rapport aux besoins de l'ingénierie système basée modèles (section 2.4).

2.2. Ingénierie Système

Un observatoire sous-marin est composé d'un grand nombre de constituants et des experts de domaines différents et d'entreprises différentes sont impliquées dans sa conception et son déploiement. Un observatoire sous-marin peut donc être considéré comme un système complexe.

Pour gérer des systèmes complexes tout au long de leur cycle de vie, l'ingénierie système est mise en avant. Cette section introduit la notion de système (sous-section 2.2.1. Or, la tendance actuelle est de faire collaborer les systèmes les uns avec les autres ce qui aboutit à la création de systèmes de systèmes (sous-section 2.2.2). L'ingénierie système (sous-section 2.2.3) définit des processus pour permettre de gérer la complexité des systèmes et des systèmes de systèmes. Ces processus peuvent être déclinés de nombreuses manières. L'une de ces déclinaisons est le standard ISO 15 288 (sous-section 2.2.4).

2.2.1. La notion de système

Ces dernières années, le milieu industriel a mis la notion de système au centre de ses préoccupations. En effet, le penser système implique de voir l'ensemble des facteurs qui influencent un système. Il s'agit d'anticiper les risques et donc les coûts associés et souvent cachés.

Plusieurs définitions ont été proposées pour définir la notion de système parmi lesquelles celle de l'organisation internationale de standardisation (ISO).

Définition 1

Un système est une combinaison d'éléments en interaction organisés pour atteindre un ou plusieurs objectifs définis [48].

Un système n'est pas seulement défini par ses constituants mais aussi par :
- leur organisation dans plusieurs dimensions comme la hiérarchie ou la géographie,
- leurs interactions aux niveaux des flux physiques, d'informations et de contrôle,
- l'objectif à atteindre qui doit être clairement défini.

Afin de préciser la nature des éléments constituants un système, l'INCOSE (International Council on Systems Engineering) a proposé la Définition 2.

Définition 2

Un système est un ensemble intégré d'éléments, de sous-systèmes ou de groupements d'éléments qui accomplissent un objectif défini. Ces éléments incluent des produits techniques (plateforme matérielle, logiciel, micro-logiciel), des process, des humains, de l'information, des techniques de travail, des installations, des services et d'autres éléments de supports [38].

Cette définition complète celle de l'ISO en précisant que les systèmes ne sont pas que techniques. La dimension humaine est partie intégrante du système et ne doit pas être négligée.

En résumé de ces définitions, un système se définit par :
- Un ensemble de constituants humains, matériels et logiciels,
- une organisation des constituants,
- des communications entre les constituants,
- un ou des objectifs clairement identifiés,
- l'utilisation d'autres systèmes, dits de supports, pour concevoir le système et lui permettre de remplir sa mission.

La complexité d'un système provient de la nature hétérogène de ses constituants. La conception et le développement d'un constituant matériel, d'une interaction humaine ou d'un logiciel sont totalement différents. La conception d'un système implique donc la collaboration entre des domaines distincts. Chaque domaine utilise des outils dédiés au domaine. Pour permettre de

concevoir efficacement le système, il est nécessaire de faire collaborer les outils des différents domaines.

Les systèmes évoluent également au fil du temps et peuvent se retrouver intégrés à des systèmes de plus grande taille alors appelés systèmes de systèmes [20].

2.2.2. Les Systèmes de Systèmes

La complexité et la distribution géographique des constituants des systèmes actuels tels que les réseaux électriques connectés ou les systèmes de défense anti-missiles ont amené à la définition du concept de Système de Système [56]. La Définition 3 est la définition proposée par le département américain de la défense pour la notion de Système de Systèmes.

Définition 3

Un système de systèmes est un ensemble ou un arrangement de systèmes indépendants et utiles qui crée un système plus large fournissant ses propres services [95].

Toutefois, cette définition ne suffit pas pour caractériser un système de systèmes. La définition manque de critères qui permettent d'identifier un système de systèmes. Or, une mauvaise identification d'un système de systèmes fait peser des risques sur la conception du système. Le risque de prendre un système pour un système de systèmes est d'augmenter la complexité de la conception du système pour rien. Au contraire, ne pas identifier qu'un système est un système de systèmes implique que l'ingénierie du système de systèmes sera incapable de remplir les missions prévues pour le système de systèmes [57]. Maier a proposé cinq critères pour identifier un système de systèmes [57] :

Indépendance managériale Les constituants du système de systèmes sont sous la responsabilité d'équipes différentes.

Indépendance fonctionnelle Les constituants du système de systèmes ont leurs propres objectifs et doivent continuer à les remplir même si une partie du système de systèmes est en panne.

Evolution Les constituants du système de systèmes évoluent indépendamment les uns des autres.

Coopération Les constituants du système de systèmes coopèrent pour atteindre les objectifs du Système de Systèmes.

Distribution géographique Les constituants du système de systèmes ne sont pas déployés au même endroit géographique.

Un système satisfaisant ces différents critères doit être considéré comme un système de systèmes. Cependant, ces critères ne font pas l'objet d'un consensus. D'autres critères ont été proposés au fil du temps. Boardmann et Sauser, par exemple, citent les critères suivants en lieu et place des critères de Maier [9] :

Autonomie Les systèmes existent pour remplir un objectif propre.

Appartenance Des systèmes sont assemblés pour créer un système de systèmes afin de répondre à un objectif qu'ils ne peuvent remplir seul.

Connectivité Les systèmes d'un système de systèmes sont connectés les uns aux autres.

Diversité Un système de systèmes doit posséder une grande diversité de fonction pour s'adapter à un environnement en constante évolution.

Emergence Un système de système possède des capacités qui émergent de l'intégration des systèmes constituants.

Différents types de système de systèmes ont été identifiés en fonction du niveau de coordination qui peut exister lors de la conception entre les différents constituants du système de systèmes [95] :

Système de systèmes virtuel Il n'existe aucune coordination prévue entre les constituants du Système de Systèmes.

Système de systèmes collaboratif Les constituants du système de systèmes entretiennent des communications volontaires mais restreintes pour atteindre un objectif commun.

Système de systèmes reconnu L'objectif du système de systèmes est clairement identifié et les collaborations entre les constituants sont conçues, toutefois les constituants conservent leurs indépendances de gestion et fonctionnelle complètes.

Système de systèmes dirigé Les constituants du système de systèmes sont conçus et gérés de manière à remplir les objectifs du système de systèmes.

Les systèmes de systèmes introduisent une complexité supplémentaire pour leur conception. Il est nécessaire de faire collaborer les systèmes constituants le système de systèmes alors que la collaboration entre les équipes en charge des systèmes constituants n'entretiennent pas forcément de collaboration.

2.2.3. L'ingénierie système

L'ingénierie système est une discipline créée pour gérer les sources de complexité lors de l'ensemble du cycle de vie d'un système. Elle s'attaque à la définition des problèmes potentiels plutôt qu'aux moyens techniques pour les résoudre. Pour tenter de caractériser l'ingénierie système, l'INCOSE a proposé la Définition 4.

Définition 4

L'ingénierie système est une approche pluri-disciplinaire et des moyens pour permettre la réalisation de système. Elle s'intéresse à la définition des besoins des clients et des fonctionnalités requises très tôt dans le cycle de développement, à la documentation des exigences, et ensuite à la conception et à la validation du système tout en considérant le problème dans son ensemble : mise en oeuvre, coût et planification, performance, formation et support, test, construction et mise au rebut. L'Ingénierie Système considère à la fois les aspects métiers et techniques de tous les clients avec pour objectif de fournir un produit de qualité qui répond aux besoins des clients [38].

Cette définition pose les caractéristiques importantes de l'ingénierie système. Tout d'abord, il s'agit d'une approche pluri-disciplinaires sur un problème. L'ingénierie système reconnaît qu'un problème ne peut pas être abordé selon un seul point de vue mais que toutes les parties prenantes, peu importe leur domaine d'expertise, doivent être consultées pour l'analyse et la résolution du problème. Cette vision implique que l'ingénierie système ne s'intéresse pas uniquement aux aspects techniques d'un problème. Cette vision globale se retrouve dans les points d'intérêts de l'ingénierie système qui couvrent l'ensemble du cycle de vie d'un système de la définition du besoin au retrait du service. L'ingénierie système doit fournir les moyens, outils, processus et méthodologie pour la réalisation des activités requises par les différentes étapes du cycle de vie d'un système.

Dans le même esprit que l'analyse d'un problème, les étapes du cycle de vie sont réalisées en fonction de plusieurs points de vue à la fois techniques, organisationnels et financiers. L'objectif de toutes les activités d'ingénierie système est l'obtention d'un système de qualité répondant aux besoins du client. L'implication du client dans la définition du système et sa validation est ainsi un facteur clé de la réussite du projet.

Le facteur qualité du système est pris en compte en mettant en place des stratégies de mesure et de tests sur le système tout au long de son cycle de vie aussi bien durant sa conception que son temps de service. Cela implique l'utilisation de moyens de tests adéquats et la mise en place d'une stratégie de Maintien en Conditions Opérationnelles (MCO). La MCO permet de garder un système opérationnel alors que son environnement de fonctionnement a évolué.

L'ingénierie système fournit les principes de base pour gérer un système sur l'ensemble de son cycle de vie. Néanmoins, ces principes doivent être déclinés en processus et les activités à réaliser au sein de ces processus doivent être définies de manière précise et partagée. C'est l'objectif des normes comme l'ISO 15288.

2.2.4. Un exemple de processus d'ingénierie Système : la norme ISO 15 288

Plusieurs normes ont été définies pour spécifier les activités d'ingénierie système. Parmi ces normes il y a par exemple l'EIA 632, l'IEEE 1220 ou encore l'ISO 15 288. Par rapport aux autres, l'ISO 15 288 s'intéresse à l'ensemble du cycle de vie du système et non pas seulement à la phase de conception. La première version de la norme ISO 15 288 [48] est parue en 2003. Elle a été revue et complétée en 2008 puis en 2015. Cette sous-section détaille les éléments clés de la version 2008 de la norme.

La norme ISO 15 288 identifie quatre groupes de processus qui coopèrent tout au long du cycle de vie du système. Ces quatre groupes sont :

— les processus contractuels,
— les processus d'entreprise,
— les processus de gestion de projet,
— les processus techniques.

Les processus contractuels sont l'ensemble des processus liés à l'élaboration des contrats avec les clients et les fournisseurs. Les processus d'entreprise sont l'ensemble des processus de l'entreprise qui supportent la conception, le développement et le suivi du système après sa livraison. Les processus de gestion de projet sont l'ensemble des processus destinés à superviser les activités de conception, de développement et de suivi du système après sa livraison. Les processus techniques sont les processus liés à l'analyse, la conception, l'implémentation, la vérification, la livraison, la maintenance et le retrait du service du système. Ces processus sont illustrés Figure 2.1.

Le premier processus est l'analyse du besoin du client et l'élaboration des exigences de toutes les parties prenantes du système. L'analyse du besoin du client permet de définir le périmètre du système. Il est alors possible d'identifier toutes les parties qui sont impactées par le fonctionnement du système et qui forment, avec le client, les parties prenantes du système [52].

Le processus d'analyse des exigences des parties prenantes permet de définir les exigences techniques que le système doit remplir pour satisfaire aux exigences des parties prenantes. Les exigences techniques servent d'entrées au processus de définition de l'architecture du système.

Il faut distinguer deux types différents d'architecture qui sont réalisés lors du processus de conception d'architecture : l'architecture logique (appelée architecture fonctionnelle dans

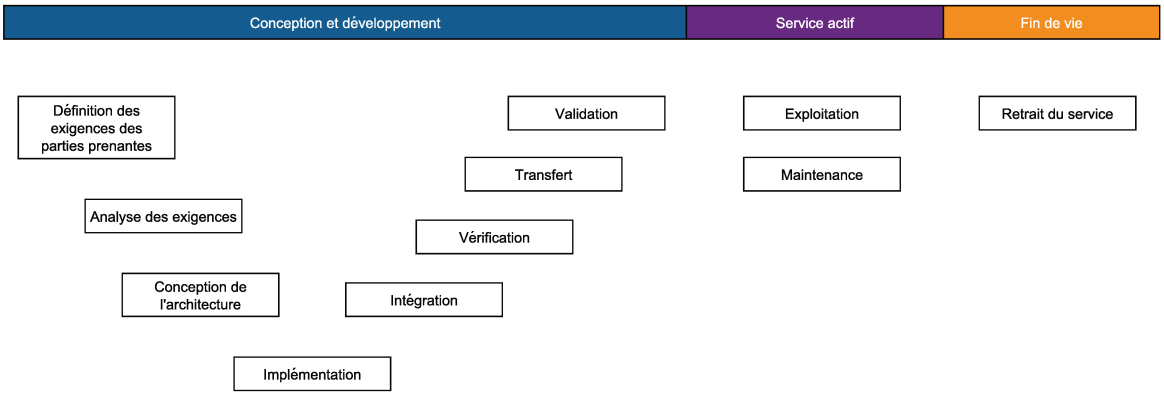


FIGURE 2.1. – Identification des processus techniques définis par le standard ISO 15288 et leurs relations avec les processus de gestion de projet. Extrait de [48].

d'autres approches) et l'architecture physique. L'architecture logique consiste en la définition des fonctions du système avec leurs performances puis l'assignation de ces fonctions aux composants logiques du système découverts par une analyse structurelle du système.

Cette architecture logique doit ensuite être dérivée dans une architecture physique où des choix technologiques sont faits pour la réalisation de l'architecture logique.

L'architecture physique est ensuite implémentée lors du processus d'implémentation durant laquelle le système est réalisé concrètement. A la fin du processus d'implémentation, chaque composant du système est réalisé.

Le processus d'intégration sert à vérifier que les différents composants peuvent être assemblés ensemble et qu'ils communiquent de manière correcte par rapport aux exigences.

Le processus de vérification du système se déroule à la fin du processus d'intégration. Il s'agit de vérifier que le système répond correctement aux exigences techniques identifiées pour le système.

En cas de réponse positive pour la vérification, le processus de transition est engagé. Il s'agit de la partie technique de la livraison du système au client.

Le processus de validation peut alors commencer. Il s'agit de vérifier que le système répond aux exigences des différentes parties prenantes dans son environnement de fonctionnement normal.

Une fois le système en opération, les processus d'exploitation sont mis en œuvre. Il s'agit notamment d'assurer le support aux utilisateurs pour s'assurer que le système leur rend le service attendu. En parallèle, les processus de maintenance permettent d'anticiper les pannes du système ou de réparer ces dernières.

A la fin de la vie du système, les processus de retrait de service se déroulent. Ces processus consistent à désassembler le système et appliquer une politique de recyclage des composants.

La réussite d'un projet de système nécessite une bonne collaboration entre les acteurs des différentes familles de processus. Par exemple, la réalisation physique du système est sous la responsabilité des acteurs des processus techniques. La réalisation physique nécessite de disposer des matières premières et des pièces détachées nécessaires à l'assemblage du système. Ces fournitures doivent être achetées auprès de fournisseurs et font donc l'objet de contrats avec les fournisseurs. L'écriture de ces contrats est sous la responsabilités des acteurs des processus contractuels. Les contrats stipulent les dates de livraison de chacune des fournitures qui ne sont

pas nécessairement la date de début du projet. La date de livraison dépend du déroulement du projet et son établissement est donc sous la responsabilité des acteurs des processus de gestion de projet. Enfin, le système réalisé doit répondre à des critères de qualité pour pouvoir être livré au client. La vérification de la qualité du système est sous la responsabilité des acteurs des processus d'entreprise.

En plus de la collaboration, l'ingénierie système met l'accent sur les aspects qualité. Afin d'augmenter la qualité des systèmes conçus et produits, une démarche d'amélioration continue, comme celle proposée par Demming [32] et illustrée par la roue de Demming de la Figure 2.2, peut être mise en place.



FIGURE 2.2. – Roue de Demming décrivant le cycle d'un processus d'amélioration continue.

La roue de Demming est constituée de quatre parties :
— Plan (planification),
— Do (réalisation),
— Check (vérification),
— Act (actions de correction).

D'un point de vue haut niveau sur l'ensemble des processus de l'ISO 15 288 :
— la planification regroupe les processus contractuels et de gestion de projet,
— la réalisation regroupe les processus techniques,
— la vérification regroupe les processus d'entreprise liés aux aspects qualité et les processus contractuels,
— les actions de corrections regroupent les processus d'entreprise liés aux aspects qualité, les processus techniques et les processus de gestion de projet.

La roue de Demming propose un schéma pour une démarche d'amélioration continue. Ce schéma peut être reproduit pour toutes les familles de processus d'ingénierie système définis par l'ISO 15 288 et notamment pour les processus techniques. Au niveau des processus techniques, la planification regroupe les processus de définition des exigences des parties prenantes et d'analyse des exigences. Le processus de définition de l'architecture logique, aussi appelée architecture fonctionnelle, est à la frontière entre la planification et la réalisation. En effet, l'architecture fonctionnelle est une analyse conceptuelle de ce que le système va faire. Il s'agit donc d'une prévision de ce que le système va faire et fait ainsi partie de la planification du système. Mais l'architecture fonctionnelle fait aussi partie de la réalisation puisque l'architecture fonctionnelle est le premier processus qui s'intéresse aux services que le système rend. En plus d'une partie du processus d'architecture fonctionnelle, la réalisation regroupe les processus d'architecture physique et d'implémentation. La vérification de la roue de Demming regroupe tous les processus d'intégration, de vérification, de transfert et de validation. Les actions de correction regroupent

les processus de maintenance.

Le schéma d'amélioration continue peut également être appliqué pour chacun des processus techniques afin de s'assurer au plus tôt de l'adéquation entre le résultat du processus et le besoin du client [27]. Par exemple, l'extraction des fonctions que le système doit rendre à partir de l'analyse des exigences permet de planifier la réalisation de l'architecture fonctionnelle. Une fois réalisée, l'architecture fonctionnelle doit être vérifiée par rapport aux exigences. Si l'architecture fonctionnelle ne répond pas à toutes les exigences, elle doit être corrigée.

La complexité d'un système ou d'un système de systèmes provient, en partie, de l'implication d'experts de domaines techniques différents et d'organisations différentes. Pour faciliter la conception, le développement et le déploiement d'un système, l'ingénierie système propose un ensemble de processus et de méthodes. L'ingénierie système met l'accent sur la collaboration entre les acteurs impliqués dans le cycle de vie du système et sur la qualité. La collaboration entre acteurs nécessite le partage d'une vision commune du système mais qui peut être adaptée à chaque acteur. Le besoin de qualité nécessite de vérifier de manière continue et à tous les niveaux les réalisations des acteurs pour s'assurer de leur concordance avec le besoin exprimé par le client. Ce besoin est très présent dans le processus d'élaboration de l'architecture fonctionnelle qui est important du fait de son caractère de pivot entre le besoin du client et la réalisation technique. L'architecture fonctionnelle doit donc être validée par rapport aux besoins pour s'assurer que les réalisations techniques seront bien réalisées en fonction des besoins du client.

Pour faciliter la communication entre les différentes équipes malgré leurs domaines d'expertises différents, la modélisation et la simulation peuvent être utilisées. Elles peuvent être utilisées aux différentes étapes du cycle de conception d'un système [48]. La simulation permet de reproduire les comportements complexes qui ont lieu au sein d'un système ou d'un système de systèmes pour tenter de les comprendre. De plus, elle donne la capacité de modéliser et de simuler un environnement. Il est ainsi possible d'étudier les évolutions du système pour s'adapter à un nouvel environnement [95].

2.3. Modélisation et Simulation

La modélisation et la simulation sont des outils qui aident à la conception de systèmes. Ils permettent de communiquer autour d'une représentation abstraite du système et de voir cette représentation s'exécuter dans un environnement contrôlé. La sous-section 2.3.1 introduit l'hétérogénéité des langages et des pratiques de modélisation dans un contexte d'ingénierie système. La sous-section 2.3.2 décrit l'utilisation de la modélisation et de la simulation pour créer des prototypes virtuels d'un système et met en avant le problème de la transition entre les modèles système conceptuels et les modèles qui définissent des simulations. La sous-section 2.3.3 présente un ensemble d'approches décrites dans la littérature pour tenter de résoudre ce problème. La sous-section 2.3.4 présente une synthèse de ces approches.

2.3.1. Model-Based Systems Engineering

Notion de modélisation et de modèle

Le centre national de ressources textuelles et lexicales a proposé la Définition 5 pour la notion de modélisation.

Définition 5

La modélisation est l'opération par laquelle on établit le modèle d'un système complexe, afin d'étudier plus commodément et de mesurer les effets sur ce système des variations de tel ou tel de ses composants [14].

L'une des activités essentielles de l'ingénierie système est la comparaison d'alternatives d'architecture notamment au niveau fonctionnel [27]. D'après sa définition, la modélisation semble être un support efficace pour analyser différentes architectures système. La modélisation repose sur la notion de modèle dont la définition ne fait pas consensus. L'Object Management Group (OMG) a proposé la Définition 6.

Définition 6

Un modèle d'un système est la description ou la spécification de ce système et de son environnement pour un objectif donné [66].

Dans sa définition, l'OMG met en avant le fait qu'un modèle est réalisé en fonction d'un objectif. Un modèle peut servir à décrire le système pour mieux le comprendre ou à le spécifier pour guider la conception et la réalisation du système. De son côté, Brown a proposé la Définition 7.

Définition 7

Un modèle fournit des abstractions d'un système physique qui permettent aux ingénieurs de raisonner sur le système en ignorant les détails non essentiels et de se concentrer sur ceux qui sont importants [12].

Par rapport à la définition de l'OMG, Brown met l'accent sur l'abstraction fournie par les modèles. Ainsi, d'après Brown, un modèle ne doit représenter que ce qui est important pour analyser le système.

Ces deux définitions, prises ensemble, définissent un modèle comme une représentation abstraite d'un système réalisée pour analyser le système en fonction d'un objectif donné.

L'utilisation de modèles dans le cadre de l'ingénierie système a été appelée Model-Based Systems Engineering (MBSE). L'utilisation de modèles dans le cadre de l'ingénierie système facilite la communication. Morel a indiqué que l'un des moyens pour diminuer le risque de prendre une décision absurde est de communiquer notamment par la visualisation [63].

Un avantage de l'utilisation des modèles est le stockage de la connaissance. Les modèles permettent d'expliciter le point du système qui a été modélisé. Ils permettent donc de pérenniser tout le travail qui a été réalisé pour comprendre l'aspect d'intérêt du système modélisé. Tout le travail étant documenté, il est possible de s'en inspirer dans le cadre de systèmes similaires. Cet usage est notamment étudié dans le cadre du projet FP7 DANSE [11]. Une bibliothèque de patrons de modélisation système a été construite à partir de l'étude de modèles de systèmes déjà réalisés. Chacun de ces patrons est accompagné d'une notice expliquant dans quels cas il

est possible de l'utiliser. Les modèles ont ainsi permis de pérenniser la compétence et les choix qui ont aboutis à l'élaboration de ces patrons.

Un second avantage est d'utiliser les modèles en tant que représentation abstraite d'un système permettant de visualiser et d'analyser différents points d'intérêts du système. Chacune de ces visualisations ou analyses peut être réalisée dans des notations différentes adaptées au public ciblé par le modèle. Par conséquent, les modèles permettent d'expliciter les problèmes et de les rendre intelligibles par tous les acteurs concernés par ces problèmes.

Enfin, la définition d'un objectif et l'utilisation d'abstractions sont des outils qui permettent d'obtenir une séparation des considérations telle que suggérée par Dijkstra [23].

Dans le cadre de l'ingénierie système, la séparation des considérations a été formalisée au sein de cadres d'architecture, comme le DoDAF (Department of Defense Architecture Framework) [96], à l'aide de points de vue.

Notion de vue et de point de vue

Le DoDAF a été mis en place pour permettre de diffuser plus largement les pratiques de conception des systèmes de très grande taille au sein du secteur de la défense américaine. Il sépare les considérations en définissant huit points de vue sur le système. Le DoDAF fournit la Définition 8 pour la notion de point de vue.

Définition 8

Un point de vue est un ensemble choisi de données d'architecture organisées autour de concepts centraux [96].

Les huit points de vue du DoDAF sont [7] :

All Viewpoint qui définit les concepts communs à tous les autres points de vue comme le vocabulaire employé au sein du projet.

Data and Information Viewpoint qui définit la modélisation des données.

Standards Viewpoint qui définit les standards et les contraintes auxquels le projet est soumis.

Capability Viewpoint qui définit les capacités du système et les délais de livraison.

Operational Viewpoint qui définit les scénarios opérationnels du système.

Services Viewpoint qui définit les services rendus par les différents acteurs qui interviennent dans le système.

Systems Viewpoint qui définit l'articulation des systèmes entre eux.

Project Viewpoint qui fait le lien entre les points de vue techniques précédents et le point de vue stratégique ainsi que le point de vue gestion de projet (dans le cadre du processus d'acquisition du DoD).

Chacun de ces points de vue est découpé en plusieurs vues qui permettent de décrire des aspects spécifiques inclus dans le point de vue. Cinquante vues différentes existent et doivent être utilisées en fonction de l'objectif à atteindre dans la modélisation. Par exemple, la modélisation de l'architecture fonctionnelle consiste à définir les fonctions réalisées par chaque constituant du système ou du système de systèmes et les échanges entre ces fonctions. Les fonctions d'un constituant rendent des services aux autres constituants afin de satisfaire les capacités du système. Les fonctions sont donc modélisées au sein du point de vue *Services Viewpoint*. La définition

des fonctions (ou services) utilise les types de données modélisées dans le point de vue *Data and Information Viewpoint* pour modéliser les données échangées entre les fonctions.

Les concepts et les méthodes définis pour permettre de réaliser des modèles pour les différents points de vue correspondent aux processus d'acquisition de systèmes du département de la défense américain.

D'autres pays ont développé leur propre cadre d'architecture comme la Grande-Bretagne qui a mis en place le MoDAF (Ministry of Defense Architecture Framework) [94] ou la France avec la méthode Agate [24]. D'autres cadres d'architecture ont vu le jour notamment dans le domaine des systèmes d'information où le TOGAF [37] s'est imposé comme l'un des standards à suivre pour la conception et le développement de tels systèmes.

Pour fédérer toutes ces approches et permettre des travaux au sein de consortium multi-nationaux, l'OTAN a lancé le NATO Architecture Framework (NAF) [18]. Le NAF, à l'instar du DoDAF, utilise la notion de points de vue découpés en vues pour permettre la modélisation des systèmes de grande taille. Le NAF définit sept points de vue pour un total de quarante huit vues. Les points de vue et les vues du NAF sont presque alignés avec ceux du DoDAF. Cependant, dans la proposition de la version 4 du NAF [19], l'ensemble des points de vue et des vues a été réorganisé sous la forme d'une grille contenant 6 couches qui peuvent être rapprochées des points de vues et un ensemble de colonnes décrivant les vues utilisables pour un point de vue donné. Cette nouvelle organisation permet d'avoir une cohérence dans la structure du cadre d'architecture et dans le contenu des vues et des points de vues [5].

Chaque cadre d'architecture fournit des concepts pour modéliser les points de vue. Ainsi, chaque cadre d'architecture utilise un vocabulaire différent. Les concepts modélisant les points de vue des cadres d'architecture sont définis au sein de méta-modèles. L'OMG a proposé la Définition 9.

Définition 9

Un méta-modèle est un modèle qui définit un langage abstrait pour définir d'autres modèles [65].

Le méta-modèle des cadres d'architecture est souvent utilisé pour étendre la sémantique de langage de modélisation existant. Par exemple, le profil UPDM [39] étend le langage SysML [29] à l'aide de stéréotypes pour permettre d'utiliser SysML pour modéliser des points de vue définis par le DoDAF ou le MoDAF. Ainsi, pour un même cadre d'architecture, plusieurs langages de modélisation peuvent être utilisés. Les cadres d'architecture n'imposent pas les langages de modélisation à utiliser pour modéliser leurs points de vue. Un même point de vue peut donc être modélisé avec des langages de modélisation différents. Chaque point de vue d'un cadre d'architecture peut également être modélisé avec des langages de modélisation différents [96]. L'utilisation de ces différents langages de modélisation force à avoir la capacité d'utiliser de manière homogène des modèles exprimés dans des langages différents.

Méthodologies associées au MBSE

Pour permettre l'utilisation des modèles dans un cadre ingénierie système, un besoin de méthodologies pour accompagner les équipes dans l'élaboration des modèles existe. Les différents cadres d'architecture proposent des méthodologies pour guider leur utilisation mais ce ne sont pas les seules méthodologies utilisables pour faire de l'ingénierie système basée modèles. Estefan a réalisé un *survey* de différentes méthodologies parmi lesquelles Rationale Harmony, Vitech

MBSE methodology et Object Oriented System Engineering Methodology [26].

Harmony est une méthodologie définie par IBM [43]. Harmony reprend les processus techniques de l'ISO 15 288 et utilise des modèles SysML [29] pour modéliser le système, stocker et échanger l'information entre les différentes activités des processus. La méthodologie Harmony définit l'organisation du modèle SysML sous la forme de paquetages. Chacun des paquetages correspond à un processus technique de l'ISO 15 288. Harmony décrit également quels diagrammes SysML doivent être utilisés pour chaque processus. La description de l'organisation des modèles et des diagrammes à utiliser assure une homogénéité de l'organisation du travail.

La méthodologie Vitech MBSE Methodology [17] identifie quatre activités dans le processus d'ingénierie système : l'élicitation des exigences, l'analyse fonctionnelle, l'architecture et la validation. L'élicitation des exigences consiste à définir les besoins du client pour le système et à traduire ces besoins en exigences que le système doit satisfaire. L'analyse fonctionnelle consiste à définir les fonctions du système et à identifier les flux de données. Le processus d'architecture consiste à définir quels composants humains, matériels et logiciels doivent être utilisés dans le système. La phase de validation consiste à s'assurer que le système conçu dans les phases d'analyse fonctionnelle et d'architecture répond aux exigences issues de la première phase. Une approche itérative de la construction des modèles est mise en avant. Cette approche permet de se concentrer sur la résolution d'un problème avant de se focaliser sur les détails de la solution. La méthodologie de Vitech est indépendante des langages de modélisation.

Une troisième méthodologie est Object Oriented System Engineering Methodology (OOSEM). Comme son nom l'indique, cette méthodologie reprend les principes de l'analyse orientée objet du monde logiciel en les adaptant à l'étude de systèmes. OOSEM reprend les activités du standard ISO 15 288 en utilisant des concepts orientés objets pour leur réalisation. Bien que réutilisant des concepts logiciels, OOSEM a prouvé son intérêt dans des systèmes à dominante mécanique comme la conception de sous-marins [71].

Depuis l'écriture du *survey* d'Estefan, de nouvelles méthodologies ont vue le jour. Parmi celles-ci, nous pouvons citer la méthodologie Arcadia de Thales récemment placée en *open source* sous le nom de Capella [100]. Capella met en avant une méthodologie extensible et intégrée. Capella doit s'adapter aux usages d'une large population d'ingénieurs système tout en respectant les standards de l'ingénierie système. Capella considère l'architecture fonctionnelle comme une étape clé de la conception d'un système. Cette approche est intégrée au sein d'un outil dédié. L'utilisation de l'outillage est guidé par la méthodologie.

Ces différents exemples montrent la diversité et l'évolutivité du monde du MBSE. Les activités de l'ingénierie système sont bien identifiées. Cependant, la manière de les mettre en œuvre dépend des objectifs et de l'expérience des ingénieurs qui travaillent sur le système. Les méthodologies et les langages de modélisation adoptés peuvent donc être différents pour chaque constituant d'un système ou d'un système de systèmes. Ainsi, un même concept de modélisation peut être utilisé pour modéliser des objets (à prendre ici au sens large, il peut s'agir de fonctions système ou d'éléments physiques) différents. De même, des concepts de modélisation différents peuvent être utilisés pour modéliser des objets équivalents. Dans ce contexte, l'utilisation de la modélisation nécessite d'analyser de manière homogène des modèles contenant des concepts équivalents mais modélisés de manières différentes ou des modèles définis dans le même langage mais structurés différemment. De plus, les systèmes et les objectifs des ingénieurs évoluent au cours du temps. Les méthodologies et les langages de modélisation évoluent également pour s'adapter. L'utilisation de la modélisation requiert d'être capable d'analyser des modèles réalisés dans des versions différentes d'un même langage de modélisation et donc de conserver la capacité d'analyser des modèles anciens.

2.3.2. Prototypage et simulation au service de l'ingénierie système

L'utilisation de prototypes permet de s'assurer que la conception, à tous les niveaux fonctionnels comme physiques, d'un système répond aux besoins exprimés par les parties prenantes. Cependant, les coûts de réalisation de prototypes concrets sont prohibitifs. Par conséquent, les prototypes sont virtualisés en utilisant des modèles. Ces modèles peuvent ensuite être analysés grâce à des techniques de simulation. Cependant, les langages de modélisation utilisés pour définir les simulations sont différents de ceux utilisés pour modéliser le système ou le système de systèmes. Il est donc nécessaire de combler cette différence entre modèles système et modèle de simulation.

Prototypage virtuel pour l'ingénierie système

Par nature, un système est amené à évoluer. Cette évolution peut entraîner des modifications mineures du système ou impliquer l'émergence de nouvelles fonctionnalités. La prise en considération de cette variabilité a un impact très fort sur l'infrastructure du système. Les évolutions du système peuvent modifier l'architecture fonctionnelle du système par l'ajout de nouvelles fonctions et / ou le flux de données entre fonctions. L'architecture physique peut aussi être impactée. La mise en place d'une démarche d'ingénierie système permet de faciliter le déroulement de la conception d'un système. En lui associant une approche d'amélioration continue, le respect réel des besoins utilisateurs peut être assuré [27]. Cette approche repose sur l'utilisation de prototypes du système. Ces prototypes permettent de tester différentes alternatives de solutions pour sélectionner celle qui répond le mieux aux besoins des utilisateurs.

Cependant, dans des secteurs comme l'industrie de défense ou l'aérospatiale, les coûts d'acquisition des matériels sont très élevés et prohibitifs. Il n'est alors pas possible de créer des prototypes complets. De plus, la vérification des artéfacts de l'ingénierie système à toutes les étapes des processus fait partie des grandes préoccupations des ingénieurs système. L'architecture fonctionnelle doit également être vérifiée par rapport aux besoins exprimés par le client lors de son élaboration. Or, lors de cette phase du processus d'architecture système, la cible matérielle du système n'est pas connue. Il n'est donc pas possible de créer un prototype physique pour vérifier l'architecture fonctionnelle. L'élaboration de l'architecture fonctionnelle requiert également de comparer différentes alternatives d'architecture. La création d'un prototype physique par alternative serait coûteuse. Le standard ISO 15 288 a établi une proposition de moyens pour permettre de vérifier les différents artéfacts de l'ingénierie système [48]. La création de prototypes virtuels et la simulation font partie des moyens proposés.

Le prototypage virtuel est utilisé avec succès dans de nombreux domaines comme la mécanique ou la conception de l'implantation d'usine et cela dès les phases amonts de la conception [77] comme celle de l'architecture fonctionnelle. Par exemple, l'atelier de conception 3D Catia de Dassault Systems permet de concevoir en trois dimensions un système mécanique et de réaliser des études mécaniques sur la conception comme la résistance des matériaux ou une étude de la répartition des efforts.

La méthodologie employée pour le prototypage virtuel d'un système se décompose en quatre étapes [41] :

1. Sélection des différentes architectures possibles pour le système.
2. Modélisation des constituants des différentes architectures. Les caractéristiques et le comportement des éléments sont identifiés. Un élément peut apparaître dans plusieurs architectures, le modèle doit donc être créé en fonction de l'objectif recherché dans la création

du prototype (i.e. il faut choisir le bon niveau d'abstraction).

3. Modélisation du comportement de l'environnement et de ses constituants.
4. Exécution du modèle et analyse des résultats afin de retenir l'architecture qui répond le mieux aux besoins exprimés par les parties prenantes.

Le prototypage virtuel permet de tester différentes alternatives d'architectures, fonctionnelles ou physiques, sans faire intervenir de matériel physique. Le prototypage virtuel permet ainsi de comparer les alternatives d'architectures pour trouver celle qui correspond le mieux aux besoins courants des clients. L'utilisation intensive de la simulation, en support d'une politique de tests, permet également de s'assurer avant le déploiement de la qualité des réalisations et de leur accord avec le besoin des parties prenantes du système.

La place de la simulation dans le prototypage virtuel

Le prototypage virtuel tient compte des aspects structurels. Pour les aspects dynamiques, le prototype doit être exécuté, par exemple par des techniques de simulation. Le département de la défense américain a proposé la Définition 10 pour la notion de simulation.

Définition 10

Une simulation est une implémentation d'un modèle dans le temps [97].

Le prototypage virtuel requiert l'exécution d'un modèle virtuel du système à créer et donc l'utilisation de la simulation. La création d'une simulation est l'objet d'un processus dont une description est fournie par Cetinkaya et al. [15] :

1. Définir le problème.
2. Faire la modélisation conceptuelle de la simulation.
3. Spécifier la simulation.
4. Implémenter la simulation.
5. Expérimenter.

Cette version du processus de simulation met en avant deux étapes de modélisation : la modélisation conceptuelle de la simulation puis la spécification de la simulation.

L'étape de définition du problème consiste à définir les objectifs de la simulation, c'est-à-dire les questions auxquelles la simulation doit apporter une réponse.

La modélisation conceptuelle modélise le problème de manière indépendante de tout outil de simulation. D'après Robinson, la modélisation conceptuelle est une étape importante du processus de simulation [76]. Robinson indique que la modélisation conceptuelle dépend fortement du domaine applicatif dans lequel la simulation est utilisée. Il convient donc de préciser les activités regroupées par la modélisation conceptuelle pour chaque simulation. Dans notre contexte, les activités d'ingénierie système peuvent permettre d'identifier les grands groupes d'activités de la modélisation conceptuelle qui nous intéressent. Dans notre contexte d'ingénierie système, les langages SysML [29] ou eFFBD [55] peuvent être utilisés.

La spécification de la simulation consiste à définir la simulation qui sera exécutée dans un formalisme de simulation indépendamment de l'outil de simulation utilisé. De nombreux formalismes de simulation existent [98]. Cette étape consiste à choisir le ou les formalismes de simulation comme par exemple *Discrete EVent Specification* (DEVS) [103] ou les réseaux de

Pétri [72]. La spécification de la simulation dépend du modèle conceptuel et du formalisme de modélisation choisi.

L'implémentation de la simulation consiste à choisir un outil de simulation qui correspond au formalisme de simulation choisi à l'étape précédente et de réaliser concrètement la simulation à l'aide de cet outil. Par exemple, pour le cas du formalisme DEVS, il est possible d'utiliser les outils OMNeT++ [99] ou Adevs [1].

La dernière étape du processus consiste à exécuter la simulation et à analyser les résultats.

Cependant, le passage de la modélisation conceptuelle à la spécification de la simulation est une tâche ardue [15, 46, 59]. En effet, les concepts utilisés dans la modélisation conceptuelle sont ceux du domaine du problème alors que ceux de la spécification de la simulation concernent le domaine de la simulation.

De plus, les langages de modélisation utilisés pour modéliser le système ou le système de systèmes au niveau conceptuel sont hétérogènes. Cette hétérogénéité augmente la difficulté du passage des modèles conceptuels à la spécification de la simulation.

Prendre en compte la durée de vie des systèmes complexifie encore le passage des modèles conceptuels à la spécification de la simulation. Du côté des modèles conceptuels, de nouveaux outils de modélisation continuent d'apparaître mais il faut continuer à être capable de travailler avec d'anciens modèles. Les outils de simulation évoluent en parallèle.

Dans un contexte d'ingénierie système, passer de la modélisation conceptuelle à la spécification de la simulation requiert d'être capable de :

— manipuler des modèles système exprimés dans des langages différents
— de manipuler des spécifications de simulation exprimés dans des formalismes différents,
— de tenir compte des évolutions des langages de modélisation système et de spécification des simulations (i.e. être capable de travailler avec d'anciennes versions de ces langages).

2.3.3. Passage de la modélisation conceptuelle à la spécification de la simulation

Les approches proposées pour passer de la modélisation conceptuelle à la spécification de la simulation reposent sur l'automatisation du passage entre les deux niveaux de modélisation. Il s'agit de générer une partie de la spécification de la simulation à partir des modèles conceptuels. Les approches peuvent être réalisées par :

a la définition de la sémantique d'exécution au sein de la définition des langages de modélisation conceptuelle,

b la transformation directe des modèles,

c la réalisation d'un environnement de modélisation avec flux de modèles.

a. Intégration de la sémantique d'exécution dans le langage de modélisation conceptuelle

Les modèles conceptuels ne sont pas forcément exécutables. Une approche pour exécuter un modèle conceptuel consiste à intégrer, au moins en partie, la description de l'exécution dans le langage de modélisation utilisé pour créer le modèle conceptuel. Il s'agit ainsi de définir la sémantique d'exécution des modèles conceptuels. Cette définition facilite la création des modèles de simulation car la manière d'exécuter les éléments de modèles conceptuels est clairement définie. Cette intégration de la sémantique d'exécution dans les langages de modélisation conceptuelle peut se faire de deux manières : soit la signification des éléments du langage

de modélisation est restreinte soit le méta-modèle du langage est étendu avec la sémantique d'exécution.

La première solution a été choisie pour la création du langage fUML. Le langage fUML est une restriction du langage de modélisation UML où les points de variations de la norme soumis à interprétation ont été supprimés [68]. Ceci permet d'avoir des exécutions de modèles fUML identiques dans tous les environnements de modélisation qui supportent fUML comme Papyrus[91].

La seconde solution a été implémentée sur le langage de modélisation d'architecture fonctionnelle eFFBD [64]. La sémantique d'exécution correspond aux travaux sur l'exécution et la validation de modèle eFFBD à l'aide de réseaux de Pétri [84]. Cette approche crée un dialecte du langage eFFBD qui peut alors être outillé et associé à un moteur d'exécution basé sur les réseaux de Pétri.

Le projet GeMOC [92] s'inscrit également dans cette approche d'intégrer la sémantique d'exécution directement dans le langage de modélisation. Le projet GeMOC propose une approche basée sur un langage de méta-métamodélisation nommé MoCCML [21] (en adaptant la Définition 9, un langage de méta-métamodélisation permet de définir des méta-modèles). Ce langage sert à définir un méta-modèle pour une sémantique d'exécution donnée. Ce méta-modèle est ensuite associé aux concepts définis dans le méta-modèle d'un langage de modélisation. Cette association permet de créer un nouveau langage de modélisation exécutable suivant la sémantique décrite avec MoCCML. Comme les deux méta-modèles sont séparés, il est possible de modifier la sémantique d'exécution. Cette approche crée également un dialecte de langage de modélisation par sémantique d'exécution associée. De plus, la définition d'associations entre les méta-modèles du langage de modélisation et de la sémantique d'exécution oblige de redéfinir les différentes associations lors des changements dans un des méta-modèles.

Restreindre la signification des éléments d'un langage de modélisation peut rendre incohérent des modèles réalisés avant la restriction. En effet, la restriction des concepts d'un langage résulte en la modification de l'utilisation même du langage. Par conséquent, les modèles réalisés avant la restriction des concepts n'ont pas le même sens que ceux réalisés après. D'où une possible perte de cohérence entre des modèles dans le même langage de modélisation.

L'intégration de la sémantique d'exécution dans un langage de modélisation implique d'avoir la capacité de modifier la définition du langage de modélisation. Dans le cadre de l'utilisation de langages de modélisation dédiés, cette approche peut être facilement mise en place. Dans le cadre de l'utilisation de langages de modélisation standards, cette approche crée des dialectes du langages de modélisation qui peuvent compliquer la compréhension des modèles si les dialectes ne sont pas partagés entre toutes les parties prenantes au projet. Dans ce cadre, un modèle peut être utilisé pour capturer la sémantique d'exécution que l'on souhaite donner aux modèles conceptuels sans modifier la définition des langages de modélisation.

b. Approches par transformations directes de modèles

L'utilisation de transformations de modèles est prônée par l'ingénierie dirigée par les modèles qui regroupe [78] :

— la définition de langages dédiés à l'aide de méta-modèles,
— l'utilisation de transformations entre modèles et entre modèles et langage de programmation.

L'ingénierie dirigée par les modèles s'intéresse à l'ensemble des activités d'ingénierie pour la conception et la réalisation d'un logiciel ou d'un système. L'architecture dirigée par les modèles

(ou Model Driven Architecture, MDA, en anglais) proposée par l'OMG [69] peut être vue comme une restriction de l'ingénierie dirigée par les modèles aux activités d'architecture.

Différentes solutions [31, 16] proposées pour passer des modèles conceptuels à la spécification de la simulation font usage des principes du MDA. Les caractéristiques principales de MDA sont :

— l'utilisation de différents types de modèles pour décrire une architecture,
— l'utilisation de transformations de modèles entre ces différents types de modèles.

MDA définit quatre catégories de modèles :

— Computation Independent Model (CIM),
— Platform Independent Model (PIM),
— Platform Description Model (PDM),
— Platform Specific Model (PSM).

Un CIM est une description du système indépendamment de l'implémentation du système. Un PIM est une description de l'architecture et du fonctionnement du système indépendamment de la plateforme sur laquelle le système sera déployé. Un PDM est une description de la plateforme qui accueillera le système. Un PSM est une description du système tel qu'il sera déployé sur la plateforme.

Les différents modèles sont définis dans des langages de modélisation qui sont formalisés par un méta-modèle. Il est possible de définir des correspondances entre les méta-modèles de langages différents et d'en déduire des transformations de modèles. La transformation de modèles qui est mise en avant par MDA consiste à transformer un PIM en PSM à l'aide d'informations contenues dans le PDM. Le principe du MDA est illustré Figure 2.3.

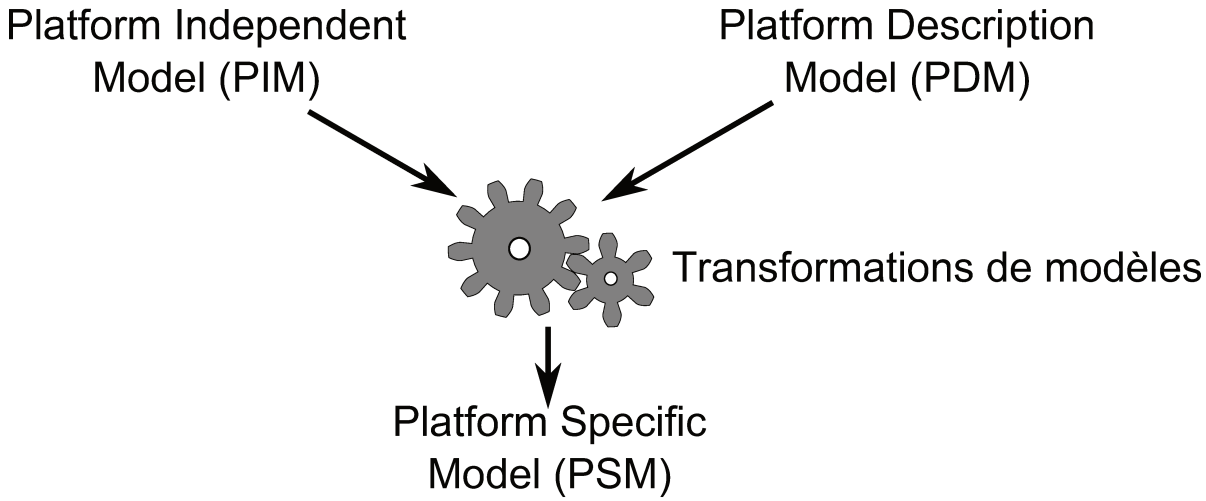


FIGURE 2.3. – Illustration des différents modèles définis par le MDA et l'utilisation des transformations de modèles.

La modélisation d'une architecture commence par l'analyse du domaine qui est capturée au sein d'un CIM. Ce dernier est utilisé pour définir l'architecture du système qui est modélisée au sein d'un PIM. En parallèle, la plateforme de déploiement de l'architecture est choisie. Cette plateforme est modélisée par un PDM. Le PIM et le PDM sont donnés en entrée d'un outil de transformations de modèles. A l'aide de règles de transformation et des données contenues dans le PIM et le PDM, l'outil de transformation génère un PSM.

Cette transformation n'est cependant pas la seule définie dans l'approche MDA. En effet, des transformations peuvent être définies au sein d'un même niveau de modèles. Il est possible de définir des transformations de modèles PIM (respectivement PSM) en modèles PIM (resp. PSM). Il s'agit souvent de raffiner les modèles sources de la transformation sans les modifier.

L'approche MDA peut être adaptée pour cibler une approche de simulation particulière comme la modélisation et la simulation basée agents comme dans le cas de l'approche ADMS [31]. Dans ce cas, un méta-modèle CIM est défini et contient les concepts permettant de définir un système à base d'agents. Un agent est une entité autonome capable d'agir en fonction d'observations de son environnement et d'interagir avec d'autres agents [93]. Un système à base d'agents inclut les agents, leurs activités et leur environnement. Un méta-modèle PIM est également défini pour permettre de représenter les modélisations des simulations basées agents. Le passage des modèles conformes au méta-modèle CIM vers des modèles conformes au méta-modèle PIM se fait à l'aide de règles de correspondances réalisées sous la forme de règles de transformation de modèles. Le problème de cette approche est de supposer au niveau du CIM que la modélisation se fait uniquement à l'aide d'agents. Dans un cadre d'ingénierie système, les éléments modélisés ne sont pas uniquement des agents. Une approche plus générique doit être utilisée au niveau du CIM pour tenir compte de tous les éléments qui doivent être modélisés en ingénierie système.

Cetinkaya et al. ont formalisé une dérivation de la pile des niveaux d'abstraction du MDA dans un cadre de simulation [16] appelée MDD4MS. Les quatre niveaux proposés sont :

1. Conceptual Model (CM équivalent du CIM de MDA)
2. Platform Independent Simulation Model (PISM équivalent du PIM de MDA).
3. Platform Specific Simulation Model (PSSM équivalent du PSM de MDA).
4. Simulation Implementation.

Dans cette approche, le modèle conceptuel est défini comme le modèle du système de plus haut niveau d'abstraction non-exécutable [53]. MDA permet de définir plusieurs modèles au même niveau d'abstraction et de créer des transformations de modèles entre ces modèles. Il s'agit notamment de permettre de raffiner les modèles au fur et à mesure de l'avancée de la conception du système. MDD4MS a été testée en utilisant le langage de modélisation de processus BPMN et des simulations suivant le formalisme DEVS [103].

Les approches basées sur la transformation de modèles ont également été utilisées avec des langages standards de la communauté ingénierie système tel que SysML [44, 59]. Dans ces travaux, les modèles conceptuels sont directement transformés en modèle de simulation pour une plateforme spécifique. Les transformations sont réalisées à partir de la sérialisation des modèles dans des fichiers XMI à l'aide de transformations exprimées à l'aide de XSLT (eXtensible Stylesheet Language Transformation) [1]. Ces travaux montrent la faisabilité de l'approche dans un contexte ingénierie système en utilisant les langages de modélisation standards du domaine comme langage de modélisation conceptuelle. Cependant, les travaux de Huang et al. et McGinnis et Ustun ne font pas usage de la couche PISM. Les transformations de modèles définies dépendent fortement du langage de modélisation système et de l'environnement de simulation utilisés. L'utilisation de la couche intermédiaire PISM permet de réduire le couplage entre modèle conceptuel et modèle pour l'outil de simulation. Ainsi un changement dans l'une des

1. XSLT est un langage de transformation pour des données contenues dans des fichiers XML.

couches CM ou PSSM est rendu transparent pour l'autre. Il est donc possible de faire évoluer indépendamment les outils de modélisation et de simulation.

Les modèles systèmes tiennent compte de multiples points de vue. Les modèles conceptuels et les modèles de simulation (PISM et PSSM) sont donc eux aussi constitués de plusieurs points de vue [46]. Une cohérence doit être trouvée entre les différents niveaux. Celle-ci peut être obtenue manuellement (approche présentée par Iba et al. dans [46]) ou automatiquement via les transformations de modèles exprimées à l'aide de règles et implémentées en XSLT ou à l'aide d'outils de transformations de modèles comme ATL [49] ou MDWorkbench [87].

L'automatisation des transformations peut se faire par l'utilisation d'un modèle pivot [40]. Dans leur approche Helle et al. disposent de trois modèles de simulation à faire exécuter dans leur environnement de simulation : les activités des acteurs de la simulation, les entités physiques et l'environnement. Helle et al. ont défini un outillage de transformations de modèles à l'aide de transformations de fichiers XML appelé Arc2Sim. Cet outillage est capable de générer une simulation à partir des trois types de modèles précédents. Les transformations utilisent un langage intermédiaire qui permet de définir les aspects structurels à l'aide de la notion de bloc et de connecteur. Les modèles d'entrée sont transformés dans un modèle conforme à ce méta-modèle intermédiaire. Ce modèle intermédiaire peut ensuite être transformé dans le modèle de la simulation qui sera réellement exécuté.

Ces travaux illustrent la difficulté de l'utilisation de plusieurs types de modèles. Pour conserver la sémantique complète de leurs modèles à chaque étape, Iba et al. doivent réaliser leurs transformations de modèles manuellement. Pour des modèles de grande taille, cette approche est prohibitive. Pour automatiser leurs transformations, Helle et al. ont défini un modèle pivot très générique au risque de perdre le sens des éléments modélisés à l'origine.

Pour automatiser les transformations de modèles capables de travailler avec des modèles de types différents, il faut être capable de définir un modèle intermédiaire qui conserve la sémantique complète de l'information recherchée dans les modèles d'entrée.

c. Définition d'un outil utilisant un flux de modèles

L'utilisation d'un environnement de modélisation mettant en œuvre des flux de modèles permet de gérer de multiples entrées sous la forme de modèles conceptuels et de multiples sorties sous la forme de modèles de simulation. De plus, l'environnement peut être capable de stocker les modèles dans un format propre. Il est ainsi possible de réutiliser les modèles même s'ils sont réalisés dans d'anciennes versions des langages de modélisation système.

Pour faire le lien entre les modèles de différents outils, Model Bus propose une approche basée sur des techniques de l'informatique distribuée : l'utilisation d'un bus de services [8]. Les services définis dans Model Bus sont soit des outils de modélisation soit des transformations de modèles. Les services possèdent des interfaces qui permettent de décrire leur aspect fonctionnel. Ces interfaces permettent de connecter les services entre eux. Les services de transformations servent à faire une adaptation d'interfaces (à base de modèles) entre deux services de modélisation. Le passage d'un service de modélisation à un autre se fait ainsi par la composition de services de transformations. Cette approche amène le postulat de base des chaînes d'outils à son paroxysme. Chaque outil conservant son modèle avec une vision silo. Les données capturées dans un modèle ne sont disponibles que pour un seul outil. Cette approche convient à des démarches très orientées par les processus. En effet, à chaque étape du processus peut correspondre un outil. Chaque outil peut travailler séparément sur ses modèles sans interaction avec d'autres outils. Cependant, elle implique de conserver l'ensemble des outils de modélisation ou de simulation

utilisés durant toute la durée de vie du système. Ce qui n'est pas toujours possible du fait de l'obsolescence des outils. De plus, l'intégration d'un nouvel outil implique de définir plusieurs nouveaux services de transformation de modèles. En effet, le nouvel outil va recevoir et envoyer des modèles dans son propre langage de modélisation qui n'a pas encore été traité. Il faut donc définir les services de transformation de modèles pour recevoir les modèles d'un outil et les envoyer à un autre outil. De plus, si le nouvel outil doit être intégré à plusieurs chaînes d'outils, le nombre de nouveaux services de transformation de modèle à définir est d'autant plus important.

Une approche différente a été utilisée dans l'outil Platypus. Platypus [73] est un outil de modélisation et de méta-modélisation basé sur le standard STEP. Les langages de modélisation sont tous exprimés à l'aide du langage de méta-modélisation EXPRESS. Les méta-modèles exprimés en langage EXPRESS sont traités pour générer automatiquement des programmes d'import et d'export entre les outils et un environnement STEP qui permet de stocker les modèles. L'utilisation d'un stockage externe des modèles permet de s'assurer de pouvoir y accéder même après que l'outil ayant servi à les créer soit retiré du service. Cependant, la définition des importateurs et des exportateurs de modèles sont définis par des correspondances entre les types des éléments de modèles. Or, les changements dans la définition d'un langage de modélisation ont un impact sur la structure des modèles. Il faut alors réécrire toutes les règles de correspondance pour s'adapter aux changements dans la définition d'un langage de modélisation y compris celles qui concernent des éléments de modèles qui ne sont pas affectés par les changements. De plus, la signification de l'élément de modèle est donc perdue au fil des transformations de modèles. En effet, dans un langage de modélisation, des éléments de modèles de même type peuvent représenter des concepts différents. Une transformation uniquement basée sur le type (et donc uniquement sur la structure) n'est pas capable de faire la distinction entre les différents sens que peuvent avoir les éléments de modèles.

Pour tenir compte du sens des éléments de modèle, une approche basée sur les ontologies comme celle proposée par Shani et Broodney peut être utilisée [86]. Les méta-modèles sont exprimés à l'aide d'ontologies au format OWL. Les modèles sont stockés de manière externe au format RDF. Les modèles sont accessibles via le web et les ontologies permettent à des outils de comprendre la signification des éléments qu'ils sont en train d'accéder. Shani et Broodney utilisent les ontologies pour pouvoir définir ce qu'ils appellent des médiations sémantiques. Il s'agit de pouvoir convertir un concept d'un langage de modélisation dans un concept d'un autre langage si les deux concepts possèdent le même sens. Quamar et al. ont montré l'intérêt de l'utilisation des ontologies pour assurer la cohérence entre des données provenant de modèles différents [74]. Cependant, cette approche requiert que les langages de modélisation possèdent une sémantique clairement définie. Or, un manque de sémantique claire a été constaté dans les langages de modélisation actuels. Ce constat d'absence a mené à la création de langage tel que fUML qui propose une sémantique précise pour un sous-ensemble de concepts du langage UML pour permettre son exécution.

L'approche proposée par Shani et Broodney s'intéresse à la signification structurelle des éléments de modèles. Dans le cadre de la simulation, l'exécution des modèles est fondamentale. Diallo s'est intéressé à la conservation de la sémantique d'exécution lors de l'échange de modèles entre deux outils [22]. Pour permettre la conservation de la sémantique d'exécution, Diallo a proposé la notion de couche sémantique définie à l'aide de modèles de concurrence et de communication (MoCC). L'ensemble est formalisé à l'aide d'un langage de modélisation dédié appelé Cometa. L'échange des modèles proprement dit se fait à l'aide de transformations de modèles. Les modèles doivent donc être adaptés entre l'outil d'origine et l'outil de destination.

Cette adaptation est à la fois la traduction des concepts du langage source dans ceux du langage cible et celle de la sémantique d'exécution du langage source dans celle du langage cible. Cette dernière peut être réalisée grâce aux informations contenues dans la couche sémantique définie en Cometa.

Ces approches permettent d'échanger des modèles entre deux outils. Cela répond au besoin d'une chaîne d'outils adossée à un processus. Des approches comme celle proposée par OpenFlexo [35] permettent quant à elles de travailler simultanément sur des modèles réalisés à l'aide d'outils différents. OpenFlexo permet de construire des vues orientées métiers sur les différents modèles. Ainsi, la représentation des éléments dans les modèles correspond à celle communément utilisée par les parties prenantes. De plus, il est possible d'éditer les modèles en ajoutant de nouveaux éléments, en supprimant ou en modifiant des éléments existants. OpenFlexo garantit la cohérence des informations entre les différents modèles lors de l'édition. Les informations ajoutées dans le modèle manipulé sont retranscrites dans les différents modèles qui ont servis d'entrées. OpenFlexo repose sur la définition de modules d'import pour les différentes technologies de stockage de modèles. OpenFlexo fournit la capacité d'importer différents types de modèles dans son environnement mais ne donne pas la possibilité à des outils tiers de venir interagir avec les modèles importés.

Les environnements utilisant des flux de modèles ont démontré leur capacité à permettre la réutilisation de modèles d'outils différents. Cependant, les environnements utilisant des flux de modèles, tout comme les approches basées sur des transformations directes de modèles, se concentrent sur la nature des éléments de modèle. Or, la modélisation est aussi l'expression de l'intention du modeleur. Ce dernier peut utiliser des éléments de modèle de même nature, c'est-à-dire de même type [88], pour modéliser des concepts différents. Par exemple, un bloc SysML peut être utilisé pour modéliser une fonction dans une architecture fonctionnelle ou un composant matériel dans une architecture physique. La signification d'un élément de modèle dépend donc du contexte dans lequel il est utilisé. En se concentrant sur la nature des éléments de modèle, les approches par transformations ou par flux de modèles peinent à tenir compte du contexte d'utilisation d'un élément de modèle. Pour tenir compte du contexte, il faut pouvoir associer différents points de vue, pris dans des contextes précis, à un même élément de modèle mais également pouvoir associer des points de vue différents à des éléments de modèle de même type.

2.3.4. Synthèse des approches basées sur l'ingénierie dirigée par les modèles

Les différentes approches présentées ont démontré l'utilisation de couches d'abstraction et de transformations de modèles ou de correspondances pour réduire l'écart entre les modèles conceptuels et les modèles de simulation. Les approches présentées ont aussi montré l'intérêt de mettre en place des environnements capables de stocker les modèles échangés. Il est possible de classer les approches précédemment présentées en trois familles : intégration, unification et fédération [70]. Guychard et al. [35] citent différents travaux autour de ces trois familles d'approches.

Wasserman [101] décrit différentes stratégies d'intégration d'outils. Ces stratégies sont basées soit sur l'intégration par une plateforme commune, par un affichage commun, par des données communes, par des mécanismes de contrôle communs ou encore des processus communs. Depuis les travaux de Wasserman, la notion d'intégration a fait l'objet d'une standardisation par l'ISO [47]. Nous avons adopté la Définition 11 adaptée à notre contexte.

Définition 11

L'intégration consiste à définir un unique langage de modélisation contenant tous les concepts de modélisation définissant les langages de modélisation qui sont intégrés. Le langage intégré sert à modéliser tous les aspects du système.

La Figure 2.4 illustre cette définition. Les langages *Langage 1* et *Langage 2* sont les langages d'entrée. A la fin du processus d'intégration des langages, représenté par une flèche, un langage dit intégré noté *Langage intégré* contient les concepts définis par les *Langage 1* et *Langage 2*.

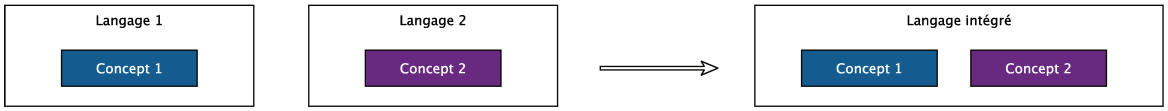


FIGURE 2.4. – Illustration de la définition de l'approche par intégration.

L'unification telle que définie par l'ISO 14258:1998 [47] repose sur l'identification d'éléments communs dans les outils ou les modèles à unifier. Zhang et al. [104] ont identifié que des outils ou des modèles peuvent utiliser le même langage, travailler sur les mêmes modèles ou avec les mêmes concepts. L'identification de ces différents points permet d'obtenir la Définition 12.

Définition 12

L'unification consiste à définir un nouveau langage de modélisation qui contient des concepts permettant d'établir une correspondance entre les concepts de différents langages. Les modèles exprimés dans les langages à unifier (modèles d'entrées) sont transformés vers le langage de modélisation commun (modèle pivot).

La Figure 2.5 illustre cette définition. Les langages *Langage 1* et *Langage 2* sont les langages à unifier. Ils contiennent tous deux des concepts qui leurs sont propres mais qui peuvent avoir un sens commun. La première étape du processus d'unification consiste à réaliser le langage unifié qui contient définit les concepts communs aux langages *Langage 1* et *Langage 2*. Des règles de correspondances des concepts des langages *Langage 1* et *Langage 2* avec les concepts du langage unifié sont établies. Ces règles permettent de définir des transformations de modèles prenant en entrée des éléments de modèles qui sont des instances des concepts des langages *Langage 1* et *Langage 2*. Ces transformations permettent de générer des instances du concept commun défini dans le langage unifié.

La définition de l'ISO sur la fédération [47] ainsi que les travaux de Guychard et al. [35] permettent de proposer la Définition 13 pour la fédération de modèles dans notre contexte.

Définition 13

La fédération consiste à définir des liens sémantiques entre des éléments de modèles exprimés dans des langages différents qui représentent le même objet.

La Figure 2.6 illustre cette définition. La fédération de modèles crée des liens entre les instances des concepts définis dans les langages de modélisation *Langage 1* et *Langage 2* car il existe une correspondance sémantique entre ces instances.

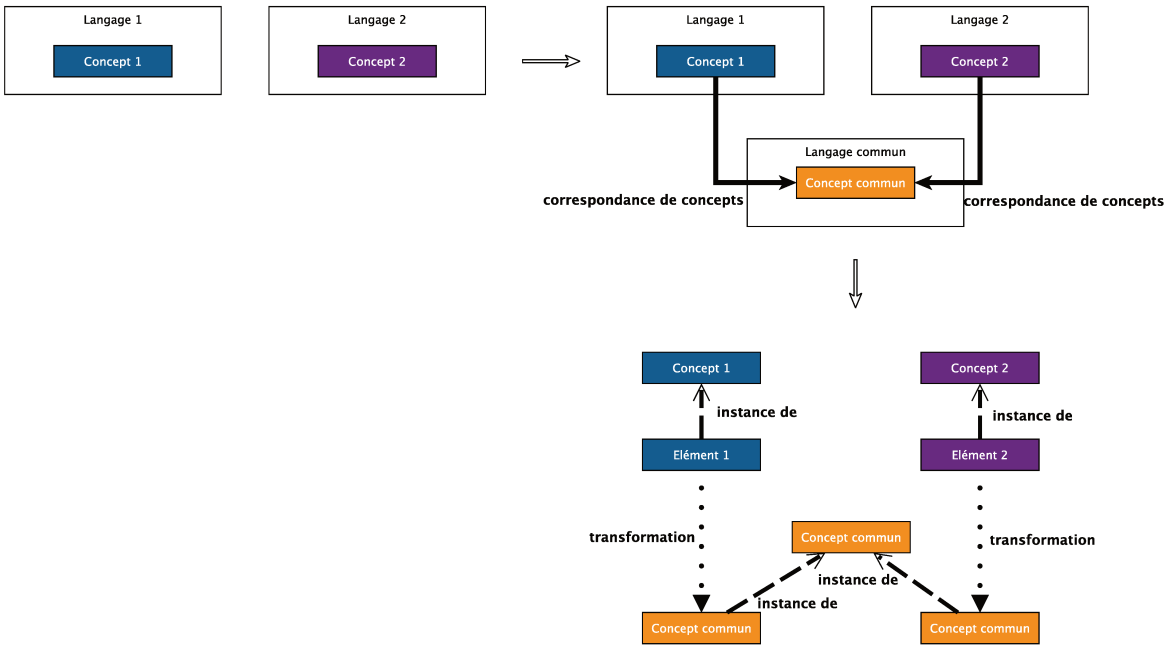


FIGURE 2.5. – Illustration de la définition de l'approche par unification.

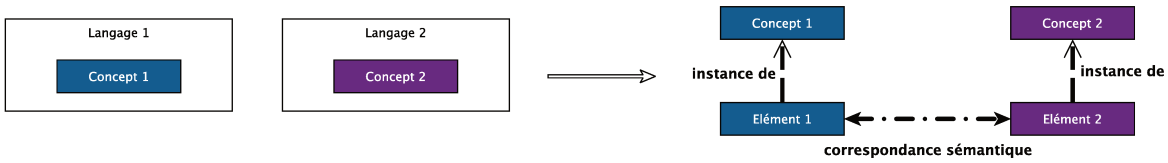


FIGURE 2.6. – Illustration de la définition de l'approche par fédération.

Ces définitions permettent d'établir le tableau 2.1 qui établit une classification des approches précédemment décrites dans les trois familles.

Approche	Intégration	Unification	Fédération
fUML [68, 91]	X		
eFFBD exécutable [64]	X		
GeMOC [92, 21]	X		
MDD4MS [53]		X	
ADMS [31]		X	
McGinnis et Ustun [59]		X	
Iba et al. [46]		X	
Helle et al. [40]		X	
Diallo [22]		X	
Model Bus [8]		X	
Platypus [73]		X	
Shani et Broodney [86]		X	
OpenFlexo [35]			X

TABLE 2.1. – Classification des approches présentées entre intégration, unification et fédération.

Comme l'a montré Diallo [22], l'exécution d'un langage de modélisation peut aussi être modélisée dans un langage de modélisation. Les approches eFFBD exécutable et fUML sont donc des approches par intégration où le modèle d'exécution est intégré au langage de modélisation. La définition d'un langage commun permet de partager le même langage et les mêmes outils entre différentes équipes. Il s'agit d'une approche qui semble bien adaptée au cas d'un système de systèmes dirigé où l'organisation du projet de système de systèmes est prévue dès le lancement du projet. Ainsi, il est possible d'imposer à tous les participants d'utiliser les mêmes outils. Cependant, si différentes équipes n'ayant pas les mêmes méthodes de travail ou la même expérience de modélisation sont impliquées dans le système de systèmes alors cette approche va se révéler problématique. En effet, ajouter l'aspect exécution dans le langage de modélisation crée un dialecte de ce langage qui n'est pas forcément partagé par tous. De plus, l'évolution des besoins du système de systèmes peut impliquer la nécessité de redéfinir l'exécution du langage de modélisation. Par conséquent, il est nécessaire de redéfinir un nouveau dialecte du langage de modélisation. Dans ce cas, il faut s'assurer que les outils pour les deux langages de modélisation peuvent s'échanger des modèles et assurer par un processus cohérent de ne pas mélanger les deux types de modèles.

L'utilisation de couches d'abstraction, comme préconisé par le MDA et le MDE, résulte en la définition de modèles intermédiaires entre les modèles conceptuels et la réalisation du système ou du logiciel. Les approches MDD4MS [53], ADMS [31] et de McGinnis et Ustun [59] utilisent des modèles intermédiaires qui sont des modèles pivots. Ce sont donc des approches par unification. Ces approches sont complétées par celles d'Iba et al. [46] et de Helle et al. [40] qui ont adressé la problématique de l'utilisation de plusieurs types de modèles à différents niveaux d'abstraction. Ces approches permettent de se concentrer sur la modélisation du problème avec les langages de modélisation adéquats. Cependant, le choix des concepts qui constituent un langage intermédiaire, aussi appelé langage pivot, est difficile. Le langage pivot peut être très abstrait comme dans le cas de l'approche de Helle et al. et ainsi perdre en expressivité. Le langage pivot peut également être très détaillé se rapprochant d'une approche par intégration.

Le langage pivot est très bien adapté aux langages de modélisation ciblés. Cependant, dans le contexte système et système de systèmes, les langages de modélisation vont évoluer. Par conséquent, soit la définition du langage pivot doit être modifiée soit les langages de modélisation doivent être adaptés. Le second cas se révèle problématique car il fait perdre l'intérêt d'utiliser des langages qui ont été standardisés. De plus, plusieurs outils peuvent être impliqués soit dans la modélisation conceptuelle soit dans la simulation.

Model Bus, Platypus et l'approche de Shani et Broodney sont des approches par chaîne d'outils. Les modèles sont transformés soit via un enchaînement de transformations entre deux formalismes de modélisation (et donc des méta-modèles intermédiaires) soit dans une représentation commune standardisée qui peut être un modèle de la sémantique des éléments. Ce sont donc également des approches par unification. Model Bus [8], Platypus [73] et l'approche de Shani et Broodney [86] nécessitent de définir un grand nombre de transformations de modèles. Or la durée du cycle de vie du système et les inévitables évolutions des langages de modélisation et des environnements de simulation imposent la mise en place d'une gestion de configuration lourde pour gérer ces transformations de modèles.

La définition de correspondances sémantiques entre des éléments de modèles est une caractéristique de l'approche d'OpenFlexo [35]. Cette approche peut donc être classée dans la famille des approches par fédération. Cette approche est très souple et permet de tenir compte des évolutions dans les modèles [35]. Cependant, la fédération implique de définir des correspondances entre deux concepts de langages de modélisation différents. Par conséquent, les correspondances sont spécifiques à un couple de langages de modélisation. Un grand nombre de correspondances doit donc être défini d'autant plus que le nombre de langages de modélisation peut être élevé.

L'ingénierie système basée sur les modèles utilise des modèles pour gérer la complexité de la conception et du développement de systèmes et de systèmes de systèmes. La simulation permet de vérifier les modèles réalisés par rapport aux besoins des clients. Cette vérification peut être réalisée à toutes les étapes des processus techniques de l'ingénierie système y compris celles qui se trouvent en amont comme la définition de l'architecture fonctionnelle.

Cependant, les modèles système, conceptuels, ne sont pas forcément exécutables. Cela implique qu'il faut passer des modèles système à des modèles de simulation exécutables. Les approches suggérées par l'ingénierie dirigée par les modèles pour réaliser ce passage reposent soit sur l'intégration, soit sur l'unification soit sur la fédération de modèles. Les approches par intégration peinent à tenir compte d'un grand nombre de langages de modélisation comme cela peut être le cas dans la conception d'un système. Les approches par unification peinent à prendre en compte le contexte dans lequel les modèles sont crées et donc le fait que des éléments de modèles de même nature puissent être interprétés de plusieurs manières. Les approches par fédération nécessitent de mettre en place un grand nombre de liens sémantiques entre des élément de modèles. La maintenance de ces liens est donc difficile. De plus, ces trois familles d'approches reposent sur le type des éléments de modèles. Or, les langages de modélisation évoluent dans le temps et donc la structure des éléments de modèles change. Ces changements forcent à reprendre les activités d'intégration ou d'unification ou de fédération.

Nous devons donc disposer d'un moyen d'exprimer un point de vue sur des éléments de modèles en fonction d'un contexte et qui est indépendant de la notion de type. Un concept de modélisation qui peut répondre à ce besoin est le concept de rôle.

2.4. Rôles

Un type décrit la nature d'un élément. Or, les activités de modélisation font également intervenir le point de vue du modeleur. La modélisation de ce point de vue, indépendamment des types qui peuvent être utilisés en son sein, est un enjeu. En effet, cette indépendance donne la capacité de travailler avec des modèles exprimés dans des langages différents. La notion de rôle a été identifiée comme un concept de modélisation permettant de répondre à cet enjeu [54].

La sous-section 2.4.1 donne les définitions utilisées dans la littérature pour la notion de rôle ainsi que les propriétés que les rôles peuvent posséder. La sous-section 2.4.2 fournit des exemples d'utilisation des rôles dans différents domaines.

2.4.1. Définition et propriétés des rôles

La notion de rôle peut être définie de plusieurs manières [88]. Tout d'abord un rôle peut être vu comme l'extrémité d'une relation. Le rôle peut également être vu comme une généralisation ou une spécialisation d'un ou plusieurs type(s). Enfin, le rôle peut être un objet spécifique associé aux instances de types. Les trois définitions précédentes font également intervenir la notion de contexte. L'association d'un rôle avec un type ou une de ses instances dépend d'un contexte. Le choix d'une définition dépend de l'objectif recherché avec l'utilisation des rôles.

La première définition convient parfaitement dans le cas où l'on cherche à déterminer les responsabilités dans une relation. Par exemple, dans une relation de composition, deux rôles sont identifiables : le composite et le composant. La définition de ces rôles ajoute un sens aux éléments auxquels ils sont liés. Le composant n'existe qu'au sein du composite. Ce dernier est responsable de la création et de la destruction des composants qu'il héberge. Ces responsabilités ne peuvent être établies que par l'existence d'une relation entre les éléments.

La seconde définition convient aux cas où l'on cherche à définir des restrictions sur les types. Par exemple, dans un modèle d'entreprise, le type personne peut représenter aussi bien un employé qu'un client. Les rôles employé et client sont ainsi définis comme des spécialisations du type personne.

La troisième définition convient dans les cas où les rôles sont des points de vue sur les instances de types. L'exemple du modèle d'entreprise où une personne peut jouer le rôle d'employé ou de client peut être repris avec cette approche. Dans ce cas, une instance du rôle employé et / ou du rôle client est (seront) associée(s) à l'instance de personne par une relation rôle / joueur.

Steimann a réalisé un état de l'art de différentes descriptions des rôles et en a extrait une liste de quinze propriétés que les rôles peuvent posséder. Ces propriétés sont décrites suivant une approche orientée objets :

Un rôle possède ses propres caractéristiques et comportements Un rôle est un objet comme un autre qui possède des attributs et des méthodes qui peuvent être utilisés indépendamment de l'objet auquel le rôle est associé.

Les rôles attachés à un objet dépendent des relations que l'objet possède avec d'autres objets Les choix de l'affectation des rôles aux objets sont réalisés en fonction de la topologie des objets.

Un objet peut jouer différents rôles simultanément Il n'existe pas d'association unique entre rôles et objets.

Un objet peut jouer plusieurs fois le même rôle simultanément Chaque instance du rôle permet d'avoir une vue particulière sur l'objet auquel elle est lié.

Un objet peut acquérir ou abandonner des rôles dynamiquement L'association entre un objet et un rôle n'est pas définitive.

La séquence d'acquisitions et d'abandons de rôle par un objet peut être contrainte En fonction d'un contexte, une série d'associations et de dissociations entre des rôles et un objet peut suivre certaines règles.

Des objets de types différents peuvent jouer le même rôle Le rôle est indépendant des types des objets auxquels il est associés.

Les rôles peuvent jouer des rôles Les rôles doivent pouvoir être manipulés comme n'importe quel autre objet.

Un rôle peut être transféré d'un objet à un autre L'association entre un rôle et un objet est transparente pour le rôle et pour l'objet.

L'état d'un objet peut être spécifique à un rôle Les valeurs des attributs d'un objet dépendent des rôles associés à cet objet.

Les propriétés d'un objet (attributs et méthodes) peuvent être spécifiques à un rôle Le fait qu'un objet soit associé à un rôle implique que cet objet possède des attributs et des méthodes déterminés.

Les rôles restreignent l'accès aux objets Le fait d'accéder à un objet par l'intermédiaire d'un rôle implique de n'avoir accès qu'aux attributs et aux méthodes de l'objet qui concerne le rôle.

Différents rôles peuvent partager des aspects structurels et comportementaux Des spécialisations de rôles peuvent être définies.

Un objet et ses rôles possèdent la même identité L'objet et les rôles qu'il possède sont vus comme un seul et même objet.

Un objet et ses rôles ont des identités différentes Chaque rôle doit être vu comme une entité à part entière.

Les deux dernières propriétés illustrent concrètement le fait que les rôles ne peuvent pas posséder toutes ces propriétés en même temps. Les deux propriétés sont en effet antinomiques, un rôle ne peut pas à la fois être sans identité propre et en avoir une.

L'utilisation des rôles implique de faire un choix des propriétés qu'ils doivent posséder. Pour guider ce choix, Kühn et al. ont d'abord étendu l'ensemble des propriétés des rôles puis ont proposé un *feature diagram* qui identifie les propriétés indispensables des rôles et celles qui sont optionnelles [54].

Les propriétés supplémentaires proposées sont :

L'association d'un rôle et d'un objet peut être contrainte Il est possible de restreindre l'association d'un rôle et d'un objet sous certaines conditions.

La relation d'association entre un rôle et un objet peut elle-même être restreinte Dans ce cas, c'est la nature de la relation qui est restreinte.

Les rôles peuvent être groupés et contraints comme un ensemble Les rôles peuvent être regroupés. Tous les éléments du groupe ainsi formé vérifient les mêmes contraintes.

Les rôles dépendent d'un contexte L'utilisation des rôles se fait en fonction d'un objectif.

Les contextes ont des attributs et des comportements Les contextes sont des objets également.

Un rôle peut faire partie de plusieurs contextes La relation d'appartenance d'un rôle à un contexte n'est pas exclusive.

Un contexte peut jouer des rôles Les contextes sont des objets comme les autres. A ce titre, ils peuvent également être associés à des rôles.

Un contexte peut agréger d'autres contextes La description d'un contexte peut se faire de manière hiérarchique.

Différents contextes peuvent partager leur structure et leurs comportements Il est possible de définir des spécialisations de contextes.

Les contextes ont leur propre identité Les contextes ont une existence propre.

Les propriétés identifiées par Kühn et al. mettent l'accent sur la notion de contexte associé à l'utilisation des rôles. Le contexte devient un objet à part entière qu'il est possible de manipuler. L'identification de la notion de contexte permet de définir clairement que les rôles sont associés à un cadre d'usage. Le contexte est réutilisable dans le sens où il s'agit d'un ensemble de circonstances qui se reproduisent et sont identifiables. Par conséquent, identifier un contexte permet d'identifier les rôles à utiliser et leurs conditions d'usage. Cela facilite grandement l'utilisation des rôles.

La notion de rôle peut être vue soit comme une place à l'extrémité d'une relation, soit comme une généralisation ou une spécialisation d'un type soit comme un objet décorant un autre objet. Les rôles possèdent plusieurs propriétés comme l'indépendance par rapport aux types des éléments qui peuvent jouer un rôle. Ces propriétés font des rôles un concept de modélisation utilisable dans tous les domaines où une couche d'interprétation de données typées est nécessaire comme les bases de données ou la modélisation.

2.4.2. Utilisation des rôles

Historiquement, la notion de rôle est apparue pour une utilisation sur des données stockées dans des bases de données. Depuis, la notion de rôle a également été utilisée pour la modélisation soit directement au sein de langages de modélisation soit pour permettre de créer des liens entre outils de modélisation.

Les rôles pour les bases de données

Le stockage des modèles dans un environnement externe aux outils de modélisation est similaire au stockage des données d'un système d'information dans une base de données. De plus, pour manipuler les données des modèles, il est intéressant de pouvoir définir des vues sur les éléments de modèles comme cela est possible avec les données stockées dans des bases de données.

Steimann attribue à Bachman et à Daya l'origine des rôles [90]. Bachman et Daya ont identifié la notion de rôles pour la modélisation des données stockées dans des bases de données orientées réseau qui ont été proposées par Bachman [4]. Au sein des bases de données, relationnelles ou non, les données sont reliées les unes aux autres. Classiquement, les données qui se trouvent à une extrémité d'un lien sont d'un type clairement défini. La notion de rôle a été définie pour permettre d'utiliser des données de types différents à la même extrémité d'un lien. Steimann résume les propriétés des rôles de Bachman et Daya sous la forme :

Définition 14: Définition des rôles de Bachman et Daya résumée par Steimann

A role in Bachman's role data model is a type that represents a partial view on entities as they participate in a relationship. All relationships are defined on role types, not entity types. An entity picks up a role by becoming member of a relationship, and drops it by leaving the relationship. Entities of different types can play the same role and the same entity can play roles of different types. From the viewpoint of the relationship, all entities in one place have the same role type and can therefore be treated alike, regardless of their possibly differing entity types [90].

Cette définition implique qu'un rôle est un point de vue sur une donnée qui dépend de la relation que la donnée entretient avec d'autres données. Le traitement de la relation est indépendant des types de données qui sont impliqués dans la relation. A une même extrémité d'une relation, les données jouent le même rôle. Des données de types différents mais situées à une extrémité d'une relation du même genre sont traitées de la même manière. Les données ne sont donc pas traitées en fonction de leur type mais en fonction de l'interprétation qui leur est donnée.

Cette approche des rôles est complétée par Halpin [36]. Halpin présente ORM2, une syntaxe graphique pour la définition de rôles sur une base de données relationnelle. Les rôles peuvent être utilisés pour définir des vues sur les données et générer des requêtes qui correspondent à ces vues. L'utilisation des rôles avec des bases de données implique que les rôles sont attachés à des objets, qui peuvent être appelés joueurs, en fonction de leurs relations. Dans le cadre des bases de données relationnelles, les joueurs sont des entités nommées qui entretiennent des relations les unes avec les autres. L'interprétation des données qui sont des instances des entités doit se faire en fonction des relations entre les entités. Les relations entre entités peuvent être interprétées par sous-ensemble, i.e. chaque relation peut être analysée séparément ou groupée avec d'autres relations pour définir un rôle.

Dans un cadre base de données, plusieurs propriétés des rôles sont intéressantes. Tout d'abord, une entité peut jouer plusieurs rôles. Les rôles définissent un point de vue, ils restreignent ainsi l'accès aux données. D'après Halpin, l'association des rôles avec les entités peut être restreinte en fonction de règles métiers.

Les rôles dans les langages de modélisation

Les langages de modélisation sont le support de la modélisation conceptuelle. Les rôles ont montré leur intérêt en tant que concept de modélisation au sein des langages pour simplifier la structure des langages. De plus, la structure du langage est influencée par le choix des propriétés des rôles qui doivent y être intégrées. Il est ainsi nécessaire de bien identifier les propriétés à intégrer et leur impact sur la structure du langage de modélisation.

Les rôles sont aussi utilisés au sein même des langages de modélisation. Steiman a mis en perspective la notion de rôle par rapport à celle d'interface dans un langage de modélisation [89]. L'exemple du langage UML est utilisé pour montrer comment la notion de rôle peut simplifier la structure d'un langage modélisation et améliorer la qualité des modèles.

La notion d'interface dans le langage UML est proche de celle utilisée dans les langages de programmation. Une interface UML permet de décrire un contrat entre deux classes : la classe qui implémente l'interface et la classe qui l'utilise. La classe qui implémente l'interface doit

fournir le comportement défini par l'interface. Une même classe peut implémenter plusieurs interfaces. Les interfaces sont utilisées aux extrémités de relations et permettent d'obtenir un faible couplage entre les classes situées à chaque extrémité. Il n'est ainsi pas nécessaire de connaître la classe concrète qui est située à l'extrémité d'une relation.

Ces propriétés des interfaces sont très similaires à celles des rôles. De plus, le langage UML utilise déjà la notion de rôle dans des diagrammes comme les diagrammes de collaboration. C'est pourquoi, Steiman a proposé de remplacer la notion d'interface par celle de rôle dans le langage UML. Steiman a utilisé la définition de rôle comme des places aux extrémités des relations.

Cet exemple illustre les propriétés que les rôles doivent posséder lorsqu'ils sont utilisés au sein de langage de modélisation. Ainsi, les rôles joués par un joueur dépendent de ses relations avec les autres joueurs. Un joueur va occuper une place d'une extrémité et va ainsi jouer le rôle attribué à cette place. Un joueur peut jouer différents rôles simultanément. Un même joueur peut se trouver à plusieurs extrémités de relations en même temps. Un objet peut acquérir ou abandonner des rôles dynamiquement. Les relations entre joueurs ne sont pas fixes. Des objets de types différents peuvent jouer le même rôle. La relation est basée sur la notion de rôle. Les propriétés d'un joueur peuvent être spécifiques à un rôle. Le rôle comme l'interface définit un contrat entre les extrémités d'une relation. Par conséquent, le joueur qui joue un rôle doit satisfaire le contrat proposé par le rôle. Les rôles restreignent l'accès au joueurs. Un rôle ne définit pas entièrement un joueur, il s'agit d'un point de vue partiel sur le joueur. Rôles et joueurs partagent la même identité. Steiman fait le choix d'avoir une identité commune entre le rôle et le joueur pour éviter les problèmes de schizophrénie de l'objet. Il argumente son choix en soulevant le problème de savoir quelle identité entre celle du rôle et celle du joueur doit être choisie lorsque l'on manipule un joueur en fonction d'un rôle. Dans le cadre des langages orientés objets, l'objet manipulé est le joueur, même s'il est vu au travers d'un prisme qui est le rôle. Par conséquent, joueur et rôles partagent la même identité.

Le travail de Steiman sur l'utilisation des rôles dans les langages de modélisation a été complété par Kühn et al. Ces derniers ont proposé un *feature diagram* regroupant les propriétés des rôles identifiées par Steiman et ont ajouté des propriétés qui permettent de définir la notion de contexte pour les rôles [54]. L'objectif de ce *feature diagram* est de faire une sélection des propriétés des rôles désirées et de générer un squelette de langage de modélisation tenant compte de la notion de rôles. La définition des rôles comme place dans une relation est à nouveau utilisée pour générer le méta-modèle du nouveau langage. L'approche suggérée par Kühn et al. identifie les dépendances entre les propriétés des rôles. Le créateur du langage de modélisation est donc guidé dans la création du langage. Le problème des dépendances entre propriétés de rôles soulevé par Graversen [33] est donc évité.

Cependant, Graversen insiste sur le besoin de comprendre la sémantique du langage obtenu en choisissant les propriétés des rôles. En effet, il est simple de choisir les propriétés des rôles qui doivent être satisfaites dans le langage de modélisation. Cependant, les choix réalisés doivent satisfaire le sens que le langage de modélisation donne aux éléments qu'il décrit. Graversen a également proposé un *feature diagram* pour les propriétés des rôles. Les propriétés sont catégorisées dans les familles :

Classe propriétés associées aux classes qui peuvent jouer des rôles,

Contraintes propriétés qui contraignent les associations entre les rôles et les objets qui jouent les rôles,

Relations propriétés qui définissent la nature de la relation entre rôles et joueurs,

Threads propriétés qui définissent l'exécution d'un rôle,

Attributs et méthodes propriétés qui définissent caractéristiques des attributs et des méthodes d'un rôles,

Comportement propriétés qui définissent le comportement des appels de méthodes sur les rôles, l'échange d'information,

Identité propriété qui définit si un rôle a une identité propre ou non,

Cycle de vie propriétés qui définissent les aspects création, attachement d'un rôle et suppression,

Typage propriétés qui définissent le typage des rôles.

Pour chaque famille Graversen fournit une analyse des propriétés et de leurs impacts sur la signification des rôles dans le langage de modélisation créé.

Les travaux de Kühn et al. et de Graversen fournissent des outils pour créer un langage de modélisation possédant la notion de rôle dont les propriétés sont garanties et la sémantique précisée. Ces travaux permettent de faire le choix des caractéristiques des rôles en fonction de la signification que l'on souhaite leur donner au sein du langage de modélisation. Il est ensuite possible de créer le langage qui correspond.

Les rôles comme intermédiaires entre langages de modélisation

Les rôles peuvent être utilisés efficacement au sein des langages de modélisation. Mais ils peuvent aussi être utilisés pour permettre de créer des passerelles entre les langages comme dans une approche chaîne d'outils.

La notion d'interface dans les langages de programmation et de modélisation est centrale. Cette notion est au cœur de la modélisation et de la programmation par composants. Wende et al. utilisent des rôles pour définir des interfaces entre langages de modélisation et ainsi définir les langages de modélisation comme des composants qui peuvent être composés les uns avec les autres [102]. Wende et al. travaillent au niveau méta-modèle. Les méta-classes d'un langage sont regroupées sous la forme d'un composant et les points d'extension, c'est-à-dire les nouvelles fonctionnalités du langage, sont spécifiées. Ces points d'extension sont les interfaces entre les langages de modélisation et sont réalisés en utilisant des rôles. Wende et al. définissent deux langages de modélisation pour permettre l'utilisation de rôles afin de composer des langages de modélisation :

— Un langage de méta-modélisation qui étend le langage Ecore [25] pour définir les rôles et ajouter les points d'extension dans les méta-modèles des langages de modélisation à composer.

— Un langage pour définir la composition des méta-modèles des langages de modélisation à composer en utilisant les rôles.

Un outil de composition prend en entrée des modèles définis dans les langages précédents et créé un langage de modélisation par composition. Dans l'implémentation du langage obtenu, à la fois le rôle et la méta-classe qui joue le rôle sont transformés en interface. La relation *jouer un rôle* devient une relation d'implémentation d'interface entre l'interface du rôle et celle de la méta-classe qui joue le rôle. Des classes qui implémentent les interfaces sont créées. La classe qui implémente l'interface du rôle est abstraite. De plus, une classe adaptateur est associée à la classe qui implémente l'interface de la méta-classe qui joue le rôle. Cet adaptateur implémente concrètement le comportement du rôle. Il permet à la classe qui représente la méta-classe qui joue le rôle d'adapter son comportement à celui qui est attendu pour le rôle.

La définition de l'adaptateur est un point fort de l'approche proposée par Wende et al. car elle permet d'obtenir un couplage faible entre le rôle et les méta-classes qui jouent le rôle au niveau de l'implémentation du langage. Il est donc aisé de faire jouer le même rôle à des implémentations de méta-classes différentes. Dans un contexte de composition de langage cela ne pose pas de problème.

Cependant, dans un contexte impliquant de multiples équipes et pour lequel la sémantique des langages de modélisation ne peut pas être modifiée, l'approche suggérée se révèle problématique. En effet, l'utilisation de cet adaptateur implique que c'est à la classe, représentant la méta-classe qui joue le rôle, d'adapter son comportement par rapport au rôle. L'implémentation rajoute donc du comportement à un élément de modèle. Ce comportement peut ne pas correspondre à la définition initiale du comportement dans le méta-modèle d'origine. De plus, l'inclusion des rôles dans les langages de modélisation à composer implique de redéfinir ces langages de modélisation. La gestion de l'évolution des langages de modélisation et l'utilisation de modèles anciens est alors problématique. Il est nécessaire de recréer des méta-modèles pour gérer toutes les différentes versions des langages de modélisation.

Au contraire de Wende et al qui ont choisi une approche par composition pour créer des langages de modélisation, Seifert et al. ont utilisé les rôles dans un contexte fédération de modèles [85]. Seifert et al. ont montré l'utilisation des rôles pour faire communiquer des outils qui n'étaient pas conçus pour pouvoir échanger de l'information. L'hypothèse de départ est que chaque outil repose sur un méta-modèle qui définit les concepts manipulés par l'outil. Les objectifs des travaux de Seifert et al. sont de :

1. permettre à un outil d'utiliser les données modélisées dans les autres outils,
2. fournir cette interopérabilité sans modifier les méta-modèles d'origines.

Seifert et al. ont relevé que l'utilisation des rôles permet de définir des vues abstraites sur les modèles à utiliser en fonction de l'usage qui doit en être fait. De plus, les rôles créent un découplage entre les outils qui peuvent donc évoluer à leur rythme. La composition des rôles permet également d'extraire de l'information de plusieurs modèles en même temps pour constituer une vue cohérente ce qui permet l'intégration des outils entre eux. Les rôles permettent de faire une correspondance entre les concepts utilisés dans les différents outils. Les rôles définis sont ainsi très proche des concepts utilisés dans l'outil qui cherche à accéder aux données modélisées avec les autres outils. Une fois les rôles définis, un fichier de correspondance est établi. Il définit la correspondance entre les rôles et les concepts des modèles des outils à intégrer ainsi que la manière dont les rôles permettent d'interpréter les concepts auxquels ils sont liés. Le fichier de correspondance définit comment les attributs des concepts sont utilisés dans le cadre de l'outil qui requiert l'information. Ce fichier de correspondance est externe à tous les outils.

L'avantage de la solution proposée par Seifert et al. est de ne pas modifier les modèles des outils à intégrer du fait de la définition externe des rôles et des correspondances. Cependant, la définition des correspondances est statique. Par conséquent, le changement de la signification d'un élément ou de l'interprétation d'un élément requiert de redéfinir les correspondances associées à cet élément et de ré-exécuter le processus d'association des rôles et d'interprétation des modèles.

Les rôles permettent de créer des passerelles entre différents langages de modélisation soit en définissant des interfaces entre les langages soit en définissant des correspondances entre les concepts des langages. Pour les deux approches, la capacité des rôles à être joués par des

objets de types différents est un atout. Wende et al. ont montré l'intérêt de séparer l'adaptation d'un objet de la définition du rôle. Seifert et al. ont démontré les possibilités des rôles pour l'interopérabilité d'outils sans modifier les méta-modèles des outils. Cependant, aucune des deux approches ne fait usage des aspects dynamique des rôles. Par conséquent, l'adaptation des approches aux évolutions internes des modèles et des méta-modèles impliquent des efforts de gestion de configuration pour conserver la cohérence entre la définition des interfaces ou des correspondances et les nouvelles versions des langages de modélisation.

2.5. Positionnement

La complexité croissante des systèmes et leurs interdépendances font de l'ingénierie des systèmes et des systèmes de systèmes un enjeu économique et scientifique important. La réussite d'un projet de système ou de système de systèmes repose sur la collaboration entre des équipes de domaines, de responsabilités et d'organisations différents. Cette collaboration implique le partage d'information et d'une vision du système entre les différentes parties prenantes. Pour ce faire, la modélisation et la simulation sont des outils efficaces. Toutefois, des équipes de domaines et d'organisations différents implique que ces équipes n'utilisent pas les mêmes outils de modélisation et de simulation. D'autant plus qu'outils de modélisation et de simulation sont différents et ne reposent pas sur les mêmes concepts. Enfin, un système a une longue durée de vie. Il est essentiel de tenir compte de l'évolution des outils.

L'hétérogénéité et l'évolution des outils de modélisation et de simulation posent de nombreux problèmes pour passer de modèles système à des modèles de simulation. Elles imposent de nombreuses contraintes dont celles qui font l'objet de cette thèse et qui sont :

A d'être capable de manipuler des modèles systèmes de plusieurs natures :

- **1** sans modifier les méta-modèles qui définissent les langages de modélisation,
- **2** sans modifier les modèles à manipuler,
- **3** en travaillant de manière identique avec des concepts à sémantique équivalente dans des langages de modélisation différents,
- **4** en étant capable d'ajouter de l'information pour faire les liens entre les différents modèles.

B d'être capable de manipuler des spécifications de simulation différentes :

- **5** en travaillant avec différents outils de simulation utilisant un même formalisme de simulation ou des formalismes différents,

C de tenir compte de l'évolution des outils :

- **6** en gérant les dialectes des langages de modélisation et de simulation.
- **7** en permettant l'ajout de nouveaux langages de modélisation et de simulation.
- **8** en continuant à supporter des modèles dont les outils utilisés pour les créer sont obsolètes et retirés du service.

Les approches actuelles par transformation directe des modèles systèmes en modèles de simulation, par intégration de la sémantique d'exécution dans les langages de modélisation système ou par outils basés sur des flux de modèles sont typées et s'intéressent à la conservation des caractéristiques structurelles entre les différents langages. Il s'agit soit de conserver la structure même du modèle soit de conserver la signification de la structure.

Les approches par intégration sont cependant invasives sur les langages de modélisation puisqu'elles en changent la signification. Les approches par transformation utilisent des modèles pivot qui permettent de s'abstraire des détails des langages de modélisation et de simulation (ce sont des approches par unification). Cependant, les transformations sont fortement dépendantes de la structure des modèles et par conséquent très impactées lors de la prise en compte de nouveaux langages ou de modifications de langages existants. Les approches par outils basés sur des flux de modèles peuvent être divisées en deux groupes, celles utilisant l'unification de modèles et celles utilisant la fédération de modèles. Les approches par unification ont les mêmes inconvénients que celles utilisées pour la transformation directe. Les approches par fédération permettent de spécifier la sémantique des éléments au sein des langages de modélisation mais pas la sémantique que le modeleur donne à l'élément. Ces différentes approches sont sensibles aux modifications des définitions des langages de modélisations.

Contrairement aux approches précédentes, les rôles permettent de modéliser le point de vue du modeleur indépendamment des types des éléments de modèles et sans modifier les langages de modélisation.

L'utilisation d'un environnement externe aux outils de modélisation et de simulation et l'utilisation de rôles pour permettre de requêter cet environnement afin de récupérer les informations contenues dans les modèles pourrait permettre de répondre aux problématiques du passage de modèles systèmes à des modèles de simulation. Ce constat a guidé nos travaux qui sont détaillés dans les chapitres suivants de ce mémoire.

3

L'environnement Role4All

Sommaire

3.1. Introduction 43
3.2. Architecture globale de Role4All 45
3.2.1. Rappel du contexte 45
3.2.2. Présentation de l'architecture générale 46
3.2.3. Justification du choix du concept de rôle 48
3.2.4. Justification du choix des parsers combinateurs 48
3.3. Choix des caractéristiques des rôles 49
3.3.1. Description et sélection des caractéristiques des rôles dans le contexte ingénierie système 50
3.3.2. Utilisation des caractéristiques sélectionnées 55
3.4. Langage de rôle 65
3.4.1. Définitions préliminaires 66
3.4.2. Structure d'un rôle 67
3.4.3. Relation entre rôle et objet 68
3.4.4. Contexte pour les rôles 73
3.5. Utilisation d'importateurs modulaires à base de parsers combinateurs 75
3.5.1. Opérateurs de combinaison de parsers 76
3.5.2. Modularité apportée par les parsers combinateurs 77
3.5.3. Intégration des parsers combinateurs 78
3.6. Conclusion 80

3.1. Introduction

La conception d'un système ou d'un système de systèmes implique de faire collaborer toutes les équipes en charge d'une partie de la conception entre elles et toutes les parties prenantes. L'ingénierie système basée sur les modèles est mise en avant pour faciliter la communication entre les équipes car elle permet de partager facilement l'information sous la forme de modèles. Pour collaborer avec les parties prenantes, la simulation permet de visualiser le fonctionnement du système dans un environnement contrôlé. Les différentes parties prenantes partagent la même

vision du système. Cependant, il n'est pas possible de faire l'hypothèse que toutes les équipes de conception impliquées dans le système partagent le même outil et donc le même langage de modélisation. De plus, les outils de modélisation système et de simulation sont différents. Ces différences d'outils de modélisation et de simulation utilisés dans un contexte d'ingénierie système impliquent :

— de manipuler des modèles système de nature différentes,
— de manipuler des modèles de simulation de nature différentes,
— de suivre les évolutions des différents langages de modélisation.

Manipuler des modèles système et des modèles de simulation implique d'être capable de passer de l'un à l'autre. Les modèles systèmes pouvant être différents, cette hétérogénéité doit être gérée lors du passage du monde système au monde de la simulation. L'hétérogénéité des modèles de simulation doit aussi être gérée. Cette problématique est renforcée par les évolutions parallèles des outils et des langages de modélisation système et de simulation.

L'hétérogénéité des outils et des langages implique d'être capable d'utiliser les concepts d'un langage dans un second. Une barrière sémantique existe entre les domaines ce qui empêche l'utilisation directe des concepts d'un domaine dans un autre. L'intégration, l'unification et la fédération sont trois approches mises en avant pour franchir cette barrière sémantique[1]. L'intégration consiste à définir un unique langage de modélisation contenant tous les concepts de modélisation définissant les langages de modélisation qui sont intégrés. Le langage intégré sert à modéliser tous les aspects du système. L'unification consiste à définir un nouveau langage de modélisation qui contient des concepts permettant d'établir une correspondance entre les concepts de différents langages. Les modèles exprimés dans les langages à unifier (modèles d'entrées) sont transformés vers le langage de modélisation commun (modèle pivot). La fédération consiste à définir des liens sémantiques entre des éléments de modèles exprimés dans des langages différents qui représentent le même objet.

Les trois approches permettent de gérer des modèles de différentes nature. Cependant, le contexte système nécessite de prendre en compte les évolutions des langages et outils de modélisation et de simulation. Il faut avoir la capacité de gérer de nouveaux langages de modélisation et de simulation tout en conservant la capacité de travailler avec des modèles anciens.

Les approches par intégration aboutissent à un langage complexe et difficile à faire évoluer. En effet, le langage contient tous les concepts des langages qui sont intégrés. L'ajout d'un nouveau langage de modélisation implique de refaire toute l'analyse du langage intégré pour déterminer quels concepts doivent être ajoutés et comment les relier aux concepts existants. Ce travail est long et fastidieux.

Les approches par unification demandent de mettre au point un langage pivot. La difficulté de cette approche est de définir ce langage pivot avec le bon niveau d'abstraction. Un langage pivot trop générique fait perdre leur sens aux éléments des modèles sources. Un langage pivot trop spécifique est difficile à étendre pour accepter les concepts de nouveaux langages de modélisation.

Pour créer des correspondances entre des éléments de modèles, les approches par fédération s'intéressent à la signification intrinsèque des éléments de modèles. La plupart de ces approches posent comme condition à la création des correspondances entre éléments que la sémantique de ces éléments soit précise. Cependant, ces approches ne tiennent pas compte de l'intention du modeleur qui peut détourner l'utilisation d'un élément pour correspondre à son besoin

1. Nous rappelons brièvement les définitions de l'intégration, de l'unification et de la fédération. Plus d'informations sont données dans la sous-section 2.3.4

spécifique. Les correspondances associées à cet élément de modèles seront donc faussées.

Dans ce contexte de travail avec de multiples modèles, les rôles ont montré leur intérêt notamment pour prendre en compte des outils qui ne sont pas conçus pour travailler ensemble. Les rôles permettent de définir un point de vue sur les modèles et donc de conserver l'intention des modeleurs.

Nous proposons un environnement intégré de modélisation nommé Role4All utilisant les rôles pour permettre à des outils tiers d'accéder aux informations contenues dans des modèles créés par différents outils de modélisation. Ce chapitre a pour objectifs de :

— présenter l'architecture globale de notre proposition (section 3.2),
— présenter le choix des caractéristiques des rôles pour répondre à nos problématiques (section 3.3),
— présenter le langage de rôles défini à partir des caractéristiques sélectionnées (section 3.4),
— présenter le mécanisme d'importateurs modulaires de modèles utilisés dans notre environnement (section 3.5).

3.2. Architecture globale de Role4All

3.2.1. Rappel du contexte

La conception des systèmes actuels nécessite la collaboration d'équipes ayant des processus, des méthodes et des outils différents. Une collaboration basée sur les modèles implique la capacité à travailler avec les modèles réalisés par les différentes équipes. Parmi les écueils auxquels cette collaboration se heurte, il est possible de citer :

— la variété des modèles utilisés
— l'impossibilité de modifier les outils de modélisation utilisés,
— l'évolution au cours du temps des processus, méthodes et outils.

La variété des modèles en entrée est due aux multiples langages de modélisation qui peuvent être utilisés pour la conception et le développement d'un système. L'utilisation de langages de modélisation dédiés augmente encore cette diversité de l'offre de langages de modélisation. De plus, pour un même langage de modélisation, les outils de modélisation associés peuvent introduire des subtilités propres à l'outil. Ceci crée des dialectes pour les langages de modélisation.

L'utilisation des langages de modélisation et des outils associés est encadrée par le processus de conception et développement utilisé par une équipe. Cette intégration implique que tout changement d'outil cause un changement au niveau du processus. De plus, les outils utilisés pour la modélisation sont fréquemment des outils réalisés par des entités tierces. Par conséquent, il n'est pas possible de modifier l'outil pour l'adapter à un nouvel usage ou pour lui donner la capacité de collaborer avec d'autres outils de manière native.

Un autre problème pour la collaboration basée modèles est la durée du cycle de vie des systèmes actuellement développés. Deux grandes familles se distinguent : les systèmes à faible durée de vie et ceux ayant une longue durée de vie. Les premiers répondent à des besoins immédiats et requièrent une grande réactivité de la part des concepteurs / développeurs. Les seconds sont destinés à être pérennes souvent pour des raisons de coût. Dans les deux cas, les équipes de conception et développement peuvent adopter de nouveaux outils. De plus, au cours du temps, les langages de modélisation sont également amenés à évoluer. Enfin, dans le cas des systèmes pérennes, il est nécessaire de conserver la capacité à travailler avec des modèles hérités (voire historique) réalisés avec des outils qui ne sont plus disponibles. L'outillage doit

pouvoir gérer les évolutions des langages de modélisation en intégrant de nouveaux langages et les modifications des langages existants.

Comme indiqué dans le chapitre 2, ces problèmes d'hétérogénéité et d'évolutivité des modèles système et de simulation impliquent que nous nous devons :

A d'être capable de manipuler des modèles systèmes de plusieurs natures :
- **1** sans modifier les méta-modèles qui définissent les langages de modélisation,
- **2** sans modifier les modèles à manipuler,
- **3** en travaillant de manière identique avec des concepts à sémantique équivalente dans des langages de modélisation différents,
- **4** en étant capable d'ajouter de l'information pour faire les liens entre les différents modèles.

B d'être capable de manipuler des spécifications de simulation différentes :
- **5** en travaillant avec différents outils de simulation utilisant un même formalisme de simulation ou des formalismes différents,

C de tenir compte de l'évolution des outils :
- **6** en gérant les dialectes des langages de modélisation et de simulation.
- **7** en permettant l'ajout de nouveaux langages de modélisation et de simulation.
- **8** en continuant à supporter des modèles dont les outils utilisés pour les créer sont obsolètes et retirés du service.

Pour répondre à ces problématiques, nous proposons un environnement de modélisation intégré : Role4All.

3.2.2. Présentation de l'architecture générale

Pour combler l'écart entre les outils des différents domaines sans toucher aux outils existants, il est nécessaire de mettre en œuvre un environnement dédié à cette tâche. Un outil spécifique à un domaine d'expertise doit pouvoir utiliser des modèles réalisés par d'autres outils. Cependant, les outils ne doivent avoir à connaître ni les détails d'implémentation des modèles ni les autres outils. La Figure 3.1 présente l'architecture de l'environnement Role4All dédié à la manipulation de modèles hétérogènes que nous proposons.

Tout comme Platypus et OpenFlexo, le choix a été fait pour Role4All de stocker les modèles dans un environnement extérieur aux outils de modélisation et de simulation. Ce choix d'architecture renforce l'indépendance de Role4All par rapport aux outils de modélisation et de simulation. Par conséquent, l'obsolescence d'un outil de modélisation n'a pas pour effet la perte de la capacité à travailler avec les modèles créés à l'aide de cet outil. Le format du stockage des modèles ne fait pas partie du périmètre de cette thèse. Des suggestions intéressantes sont notamment proposées par l'environnement Platypus où le stockage de l'information est réalisé suivant la norme STEP. Cependant, notre proposition s'intéresse aux obligations issues de l'externalisation du stockage des modèles :

— la fourniture de points d'accès aux éléments de modèles stockés,
— la capacité à charger des modèles dans l'environnement.

Pour donner des points d'accès aux éléments de modèles, nous avons choisi d'utiliser un modèle de rôles. Ce modèle est défini par l'intermédiaire d'un méta-modèle et peut être créé en utilisant un Domain Specific Language. Pour réaliser les imports de modèles, nous avons choisi d'utiliser des parsers combinateurs afin d'implémenter des importateurs modulaires de modèles. Le choix de ces deux techniques est détaillé dans les sous-sections suivantes.

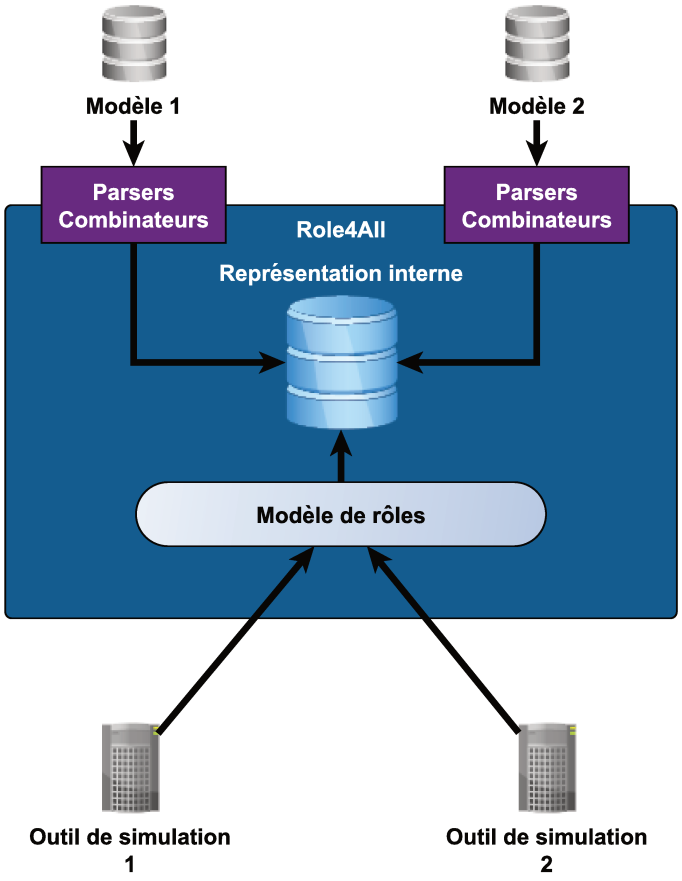


FIGURE 3.1. – Outillage pour créer un lien entre les modèles du système à faire et des systèmes support : les parsers combinateurs permettent d'importer les modèles du système à faire et les rôles permettent d'adapter ces modèles à ceux utilisés par les systèmes supports

3.2.3. Justification du choix du concept de rôle

De manière condensée, trois définitions conceptuelles ont été proposées pour la notion de rôle [88] :

- les rôles comme qualificatifs des extrémités d'une relation dans un modèle,
- les rôles comme une généralisation ou une spécialisation des types des langages de modélisation,
- les rôles comme des instances associées à des instances de types des langages de modélisation.

La première définition implique que des relations entre des éléments de modèles puissent être identifiées. Cette définition s'applique aux langages de modélisation. En effet, elle donne de la signification aux extrémités des relations modélisées. Cependant, l'interprétation que l'on souhaite donner à une relation peut évoluer au cours du temps. En effet, deux outils d'analyse de modèles n'envisagent pas les relations entre éléments de modèles de la même manière. La définition des rôles doit ainsi être flexible et non pas figée dans les langages de modélisation.

La seconde définition implique d'étendre les méta-modèles qui définissent les types utilisés dans les modèles. Or, il n'est pas possible de modifier la définition des langages de modélisation. En effet, la plupart de ces langages sont standards et il est important de conserver la compatibilité avec les standards. De plus, la modification des outils de modélisation pour prendre en compte les rôles n'est pas possible.

La troisième définition nous semble intéressante car elle permet de créer un modèle de rôle séparé des modèles à manipuler puis d'associer ce modèle de rôles aux éléments de modèles qui nous intéressent. Ainsi, nous ne modifions pas les éléments de modèles que nous manipulons. De plus, nous pouvons faire évoluer notre modèle de rôle à notre guise car nous en avons le contrôle.

Les rôles ont été utilisés dans différents domaines pour permettre de manipuler de l'information contenue soit dans des bases de données soit dans des modèles. Ces utilisations des rôles ont montré la possibilité de travailler avec des modèles de natures différentes, une des caractéristiques que nous recherchons pour le contexte ingénierie système.

Nous avons donc décidé d'utiliser des rôles pour permettre de définir des interfaces entre les outils de simulation et les modèles systèmes. Les outils de simulation accèdent aux éléments de modèles stockés dans notre environnement en passant par l'intermédiaire de rôles. Les rôles modélisent le point de vue que les outils de simulation ont besoin d'avoir sur les modèles. Les rôles font l'adaptation des éléments de modèles aux points de vue utilisés par les outils de simulation.

Pour faire le travail d'interface entre les outils de modélisation système et notre environnement, nous souhaitons posséder des importateurs modulaires. Pour cela, nous avons étudié l'utilisation de parsers combinateurs.

3.2.4. Justification du choix des parsers combinateurs

La conception de grands systèmes implique l'utilisation de nombreux langages de modélisation. De plus, l'évolution de ces langages nécessite d'avoir la capacité de tenir compte rapidement de ces évolutions. Une architecture modulaire des importateurs permet d'isoler les modifications et ainsi faciliter à la fois l'évolution et la maintenance. En effet, le lieu des modifications est clairement identifié. De plus, il est possible de sélectionner et de composer les modules d'import

en fonction des besoins. L'importateur obtenu par composition des modules répond aux besoins exprimés pour l'import de modèles.

Les différents outils de modélisation sérialisent les modèles réalisés par leur intermédiaire dans un format textuel. L'utilisation de parsers est ainsi adaptée pour réaliser l'import des modèles. En effet, ils permettent d'utiliser directement les modèles sérialisés sans avoir à écrire de programmes d'interface avec les différents outils de modélisation. L'écriture de programme d'interface est d'ailleurs impossible pour les outils de modélisation abandonnés. La capacité de travailler avec des modèles réalisés avec ces outils est donc conservée.

Dans le domaine de l'écriture de parsers, les parsers combinateurs fournissent une solution pour rendre modulaire l'écriture de parsers. La Définition 15 décrit la notion de parsers combinateur.

Définition 15

Un parsers combinateur est un opérateur qui prend en entrée deux parsers et les combine pour obtenir un troisième parser [45].

Les parsers combinateurs permettent de créer des parsers en composant d'autres parsers. Ainsi, des parsers pour une partie restreinte du modèle sérialisé qui correspond à un concept précis peuvent être définis. Les différents parsers peuvent alors être combinés pour obtenir le parser pour l'ensemble du modèle sérialisé. En cas de modifications de la sérialisation d'un concept ou de la définition du concept, seul le parser associé est impacté. Il n'est donc pas nécessaire de réécrire l'ensemble du parser. De plus, lorsque plusieurs modèles sont utilisés, les parsers pour chaque langage de modélisation peuvent être réutilisés. Ils peuvent être combinés pour obtenir un parser capable d'analyser tous les modèles dont l'écriture est ainsi facilitée. En effet, il n'est pas nécessaire d'écrire la totalité du parser, il s'agit uniquement de trouver les bons opérateurs de combinaison pour combiner les parsers des langages.

Nous avons décrit l'architecture globale de notre environnement Role4All qui repose sur :

— l'utilisation de rôles pour définir un modèle d'interface entre des modèles système et des modèles de simulation,
— l'utilisation de parsers combinateurs pour écrire des importateurs de modèles sérialisés.

Les rôles permettent de s'interfacer avec des modèles de nature différentes sans modifier les modèles d'origine. Les parsers combinateurs permettent d'écrire des importateurs de modèles modulaires et donc faciles à maintenir pour suivre les évolutions des langages de modélisation.

3.3. Choix des caractéristiques des rôles

Nous avons choisi d'utiliser les rôles pour permettre de définir des interfaces entre les outils de simulation et les modèles stockés au sein de l'environnement Role4All. La sous-section 3.3.1 présente les différentes caractéristiques des rôles et justifie leur sélection ou non pour la création d'un méta-modèle de rôles. La sous-section 3.3.2 illustre l'utilisation des caractéristiques sélectionnées dans un contexte ingénierie système impliquant des modèles hétérogènes et évolutifs.

3.3.1. Description et sélection des caractéristiques des rôles dans le contexte ingénierie système

Comme présenté chapitre 2, Steimann [88] et Kühn et al. [54] ont défini un ensemble de propriétés que les rôles peuvent posséder. L'implémentation d'un outillage pour permettre de modéliser les rôles implique de définir quelles propriétés doivent être possédées par les rôles pour l'implémentation.

Caractéristiques des rôles sélectionnées

Le rôle est indépendant du type des objets auxquels il est lié. Nous devons travailler avec des modèles hétérogènes. L'interprétation définie par les rôles ne dépend pas du type des éléments de modèles auxquels il doit être associé. Par conséquent, un rôle qui définit une interprétation doit pouvoir être associé à des éléments de modèles différents définis au sein du même langage de modélisation ou de langages de modélisation différents. Cette propriété est retenue et sera référée sous la dénomination de Propriété 1.

Propriété 1

Des objets de types différents peuvent jouer le même rôle.

Un rôle doit pouvoir posséder ses propres attributs et méthodes indépendamment des objets auxquels il est lié. Parmi les problèmes auxquels nous devons faire face, nous devons avoir la capacité d'ajouter de l'information manquante sans modifier les modèles originaux pour permettre de créer correctement les simulations. Les rôles sont le point d'entrée des outils pour accéder aux modèles. Ils doivent donc servir aux outils de simulation toute l'information nécessaire y compris celle qui ne se trouve pas dans les modèles système. Par conséquent, cette information est stockée dans la définition des rôles et est accessible par l'intermédiaire de propriétés propres aux rôles. Cette propriété est donc sélectionnée et sera référée sous la dénomination de Propriété 2.

Propriété 2

Un rôle possède ses propres caractéristiques et comportements.

Un même objet peut être associé à plusieurs rôles différents en même temps. Nous devons pouvoir travailler avec des outils de simulation qui ne sont pas du même type et avec des outils qui permettent de couvrir l'ensemble du cycle de simulation comme des outils de documentation. Par conséquent, nous devons avoir la capacité de définir des points de vue différents sur les mêmes éléments de modèles. Cette propriété est donc retenue et sera référée sous la dénomination de Propriété 3.

Propriété 3

Un objet doit pouvoir jouer des rôles différents.

Un objet peut être associé à plusieurs instances du même rôle. Les modèles fournis par les équipes de conception peuvent contenir une information modélisée une seule fois mais qui doit être utilisée plusieurs fois. Chaque utilisation doit se faire avec des paramètres différents. Or,

nous ne pouvons pas modifier les modèles pour dupliquer l'information qui nous intéresse. Nous devons avoir la capacité d'interpréter le même élément de modèle de la même façon mais avec des variations sur les paramètres de l'interprétation et donc les propriétés des rôles. En effet, une simulation doit pouvoir travailler avec un élément de modèle en l'interprétant plusieurs fois suivant le même point de vue mais chaque interprétation doit pouvoir être spécifique. Cette propriété est donc retenue et sera référée sous la dénomination de Propriété 4.

Propriété 4

Un objet peut jouer plusieurs fois le même rôle simultanément.

La relation entre un objet et un rôle peut être créée ou supprimée dynamiquement. Nous devons tenir compte de l'évolution des outils de simulation. Par conséquent l'interprétation des éléments de modèles évolue également au cours du temps. Dans notre approche, l'interprétation est portée par les rôles. Ainsi, les relations entre les rôles et les objets qui les jouent doivent pouvoir évoluer. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 5.

Propriété 5

Un objet peut acquérir ou abandonner des rôles dynamiquement.

Les rôles sont eux-mêmes des objets qui peuvent aussi être associés à d'autres rôles. Dans notre étude, nous souhaitons créer des modèles de simulation à partir de modèles système. Nous utilisons un certain point de vue pour interpréter les modèles système. Ce point de vue est modélisé à l'aide de rôles. Pour créer nos modèles de simulation, nous interprétons ce premier point de vue à l'aide d'un autre point de vue spécifique aux activités de simulation. Ce second point de vue est aussi modélisé sous la forme de rôles. Les rôles qui permettent d'interpréter les modèles système doivent donc pouvoir jouer les rôles spécifiques aux activités de simulation. Cette propriété est retenue et sera référée sous la dénomination Propriété 6

Propriété 6

Les rôles peuvent jouer des rôles.

En accédant à un objet au travers d'un rôle, il n'est pas possible d'avoir accès à toutes les caractéristiques de l'objet. Dans notre approche, les rôles doivent fournir un point de vue sur les éléments de modèles. Par conséquent, les rôles fournissent des propriétés d'accès sur les éléments auxquels ils sont liés. Le rôle fournit une interprétation des éléments de modèles auxquels il est lié et choisi donc quelles informations sont rendues disponibles. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 7.

Propriété 7

Les rôles restreignent l'accès aux objets.

Les rôles peuvent entretenir des relations de spécialisation et de généralisation. La définition

d'un rôle doit permettre de travailler à différents niveaux d'abstraction et permettre la réutilisation. Nous devons pouvoir travailler avec des outils de simulation différents mais du même type. Ainsi, nous pouvons vouloir réutiliser des parties des rôles déjà définis ou définir des rôles génériques qui permettent d'avoir un patron de travail pour définir de nouveaux rôles. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 8.

Propriété 8

Différents rôles peuvent partager des aspects structurels et comportementaux.

Les rôles et les objets ont des identifiants propres. Comme nous pouvons travailler avec des objets qui sont liés à plusieurs instances du même rôle et que nous souhaitons manipuler les rôles, nous devons pouvoir différencier chaque instance de rôle. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 9.

Propriété 9

Un objet et ses rôles ont des identités différentes.

La propriété *l'association d'un rôle et d'un objet peut être contrainte* peut être prise dans un sens large où la définition de l'association entre un rôle et un objet dépend des propriétés de l'objet. Nous utilisons les rôles pour permettre de définir des interfaces entre des outils de simulation et des modèles système. Par conséquent, un élément d'un modèle système peut avoir plusieurs significations dans le monde de la simulation. Le choix de la signification de l'élément peut dépendre des valeurs des attributs de l'objet. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 10.

Propriété 10

L'association d'un rôle et d'un objet peut être contrainte.

Les rôles peuvent être groupés sous la forme d'un rôle composite et un groupe de contraintes peut être appliqué à l'ensemble des rôles ainsi groupés. Dans le cadre de la simulation de modèles système, des concepts complexes peuvent être compris de plusieurs perspectives. Ils peuvent être compris comme un tout avec un haut niveau d'abstraction ou comme une série de sous-facettes à un niveau de compréhension du concept plus fin. Par conséquent,un rôle qui fournit une interprétation des éléments de modèles doit pouvoir fournir cette interface hiérarchique. Cependant, dans un premier temps, nous nous intéressons à l'aspect structurel de la définition des rôles. Par conséquent, nous retenons la partie de la propriété sur la définition de rôle composite. La partie de la propriété sur l'application de contraintes sur des rôles regroupés au sein d'un composite fait partie des perspectives. La propriété ainsi sélectionnée sera référée sous la dénomination Propriété 11

Propriété 11

Les rôles peuvent être groupés sous la forme de composite.

Les rôles utilisés avec des objets dépendent d'un contexte. Dans le cadre de la simulation de modèle système, différents outils de simulation peuvent être utilisés pour des simulations dont la nature est différente. L'outil de simulation et la nature de la simulation à réaliser permettent de définir quels rôles peuvent être utilisés pour créer l'interface entre les outils de modélisation système et les outils de simulation. Ce sont les contextes d'utilisation des rôles. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 12.

Propriété 12

Les rôles dépendent d'un contexte.

La relation entre un rôle et un contexte n'est pas exclusive. Nous travaillons avec différents outils de simulation qui peuvent être de même nature. Par conséquent, un rôle défini pour permettre l'interprétation d'un élément de modèle pour le contexte d'un outil de simulation peut être réutilisé dans le contexte d'un autre outil de simulation du même type. Cette réutilisation permet de factoriser les efforts de développement des rôles. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 13.

Propriété 13

Un rôle peut faire partie de plusieurs contextes.

Chaque contexte doit pouvoir être identifié. Nous souhaitons avoir la capacité de travailler avec des contextes précis notamment pour pouvoir leur associer des règles d'utilisation de rôles. Par conséquent, nous devons avoir un moyen d'identifier clairement chaque contexte. Cette propriété est donc retenue et sera référée sous la dénomination Propriété 14.

Propriété 14

Les contextes ont leur propre identité.

Caractéristiques qui font partie des perspectives

Les contextes sont des objets à part entière avec des propriétés et des comportements propres. Nous avons choisi de voir les contextes comme des moyens de grouper des rôles en fonction d'un usage et / ou d'un objectif et comme un moyen de définir des contraintes d'utilisation des rôles. La possession de contraintes d'utilisation des rôles n'est pas une caractéristique intrinsèque d'un contexte particulier mais de l'ensemble des contextes. Nous n'avons donc pas créer une description qui peut être instanciée d'un contexte particulier mais un contexte est une instance d'une description générale. Le traitement de cette caractéristique fait partie des perspectives.

La définition d'un contexte peut être complexe. Une vision hiérarchique permet de définir des contextes de manière modulaire et réutilisable. Nous reconnaissons les avantages fournis par cette propriété. Cependant, nous avons choisi, dans un premier temps, d'adopter une vision

non hiérarchique des contextes. Cette propriété fait partie des perspectives.

Un contexte peut être vu comme une spécialisation d'un autre contexte. Pour simuler des modèles système, il est possible de définir un contexte qui décrit le type de simulation à réaliser tandis qu'un second contexte héritant du premier spécialiserait la simulation pour un outil donné. Toutefois, nous nous sommes concentré sur les aspects de regroupement de rôles et de définition de règles d'utilisation des rôles au niveau des contextes. Cette propriété fait partie des perspectives.

Caractéristiques non retenues

Le choix d'attacher un rôle à un objet ou non dépend des associations que l'objet possède. Cette propriété est très liée à la définition des rôles comme place dans des associations. En effet, dans cette définition, les rôles définissent les extrémités des associations et un objet joue le rôle d'une extrémité lorsqu'il s'attache à l'extrémité. Dans notre approche, les rôles sont des instances à part entière qui s'attachent aux objets. Les critères pour qu'un rôle soit attaché à un objet dépend du point de vue de l'utilisateur des rôles. Par conséquent, attacher un rôle à un objet ne dépend pas uniquement des relations que l'objet entretient avec les autres mais plus du point de vue qui doit être adopté sur cet objet. Cette propriété n'est pas retenue en tant que telle.

Les objets peuvent être associés ou dissociés dynamiquement avec des rôles. Toutefois, ces actions d'associations et de dissociations peuvent être contraintes de suivre un ordre prédéfini. Dans le cadre de notre approche, les rôles fournissent une interprétation sur des éléments de modèles. Nous nous trouvons également dans un contexte évolutif incertain. Il n'est donc pas possible d'émettre des hypothèses sur les séquences d'interprétation des éléments de modèles. La propriété n'est pas retenue.

Les valeurs des attributs d'un objet dépendent des rôles qui sont associés à l'objet. Nous travaillons avec des modèles qui proviennent d'équipes tierces et que nous ne pouvons pas modifier. Par conséquent, l'ajout d'un rôle ne doit pas impacter l'élément de modèle auquel il est associé. Cette propriété n'est donc pas retenue.

La définition d'un objet peut dépendre des rôles auxquels il est associé. La définition des modèles systèmes est indépendante de notre outillage de rôle. Nous ne pouvons pas modifier la définition des éléments qui sont dans les modèles avec lesquels nous travaillons. Cette propriété n'est donc pas retenue.

Un objet et ses rôles doivent être vus comme un seul objet. Dans notre approche, nous créons un modèle de rôles dédiés et les outils de simulation accèdent aux rôles et non pas aux éléments de modèles. Par conséquent, les rôles doivent pouvoir être vus comme des objets à part entière. Cette propriété n'est donc pas retenue.

La nature de la relation d'association entre un rôle et un objet peut être contrainte. Dans un premier temps, nous avons choisi de nous concentrer sur les aspects dynamiques de la relation

entre un rôle et un objet. Cette propriété est donc rejetée.

Les contextes peuvent être vus comme des objets comme les autres et jouer des rôles. Notre vision des contextes comme des regroupements de rôles et de règles d'utilisation de rôles ne voit pas les contextes comme les éléments d'un modèle. Nous voyons les contextes comme des éléments qui peuvent être attachés à des éléments de modèles. Par conséquent, nous ne pouvons pas attacher de rôles à des contextes. La propriété n'est donc pas retenue.

Nous avons présenté les différentes caractéristiques des rôles et justifié le choix des caractéristiques choisies pour créer un méta-modèle de rôles. Dans la sous-section suivante, nous montrons l'utilisation des propriétés sélectionnées dans notre contexte de simulation de modèles système hétérogènes.

3.3.2. Utilisation des caractéristiques sélectionnées

Exemple d'illustration

Dans les sous-sections suivantes, nous utiliserons un exemple de système d'acquisition de données qui doit être simulé. Cet exemple sera repris et développé dans le chapitre 5. Cet exemple permettra d'illustrer les caractéristiques des rôles que nous avons sélectionnées. Le système a été modélisé à l'aide des langages de modélisation système SysML [29] et eFFBD (extended Functional Flow Block Diagrams) [55].

Le système modélisé est composé de deux sous-systèmes : un système d'acquisition qui est un hydrophone et un système de stockage et de traitement de l'information qui est vu comme un système d'information. Les données acquises par le système d'acquisition est un signal sonore qui est transmis par l'hydrophone au système de traitement. La modélisation SysML à l'aide de la version 1.2 du langage de ce système est donnée Figure 3.2.

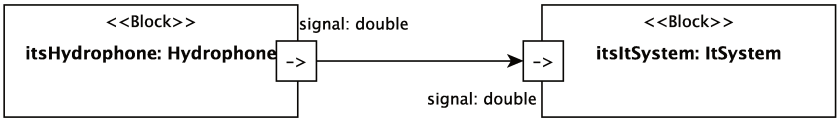


FIGURE 3.2. – Modèle SysML présentant le bloc itsHydrophone comme émetteur d'un flux signal et le bloc itsItSystem comme récepteur du flux

Le système est modélisé sous la forme de deux blocs Hydrophone et ItSystem. Le diagramme présenté est un diagramme de blocs interne (IBD [2]), il présente donc des instances des blocs Hydrophone et ItSystem nommée itsHydrophone et itsItSystem. Le bloc Hydrophone possède un port de sortie de type flowport qui lui permet de produire un flux, ce port porte le nom de signal et permet de produire des données de type double. Le bloc ItSystem possède un port d'entrée de type flowport qui lui permet de recevoir un flux. Ce port porte également le nom de signal et permet de recevoir des données de type double. Les deux ports affectés aux instances des blocs Hydrophone et ItSystem sont liés dans l'IBD.

La modélisation du système à l'aide du langage eFFBD est présenté Figure 3.3. Le langage

2. Contrairement à l'usage, nous conservons l'acronyme anglais.

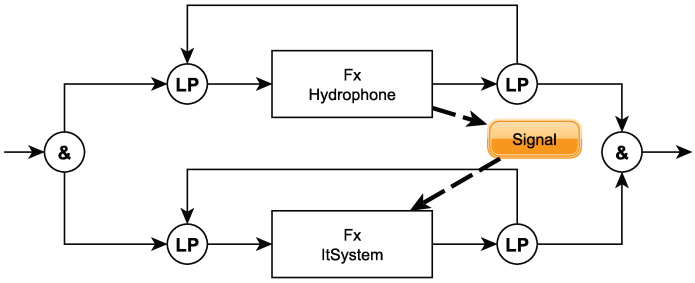


FIGURE 3.3. – Modèle eFFBD présentant la fonction Hydrophone comme émetteur d'un flux Signal et la fonction ItSystem comme récepteur du flux

de modélisation eFFBD modélise des fonctions et les flux échangés entre les fonctions. L'hydrophone et le système d'information ont été modélisés sous la forme de fonctions. Elles s'échangent un flux nommé signal émis par la fonction Hydrophone et reçus par la fonction ItSystem. Le langage eFFBD permet également de préciser le flux de contrôle des fonctions comme le fait que des fonctions fonctionnent en parallèle (représenté par un nœud &) ou bouclent (représenté par un nœud LP). Dans le cas de notre système, les deux fonctions Hydrophone et ItSystem bouclent. De plus, les fonctions bouclées fonctionnent en parallèle.

Nous allons utiliser ces deux modèles pour illustrer l'utilisation des rôles au niveau modèle.

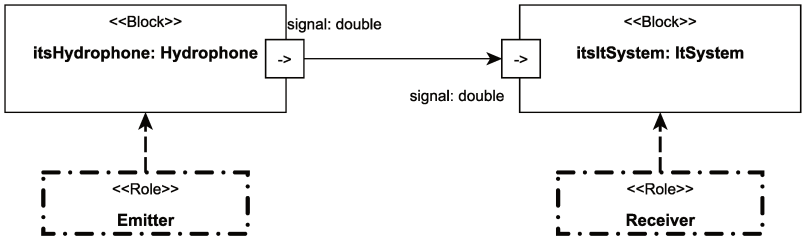
Des objets de types différents peuvent jouer le même rôle (Propriété 1)

Cette propriété nous permet de travailler avec des modèles réalisés dans des langages de modélisation différents sans se soucier des différents types des éléments modélisés dans les différents langages. Dans notre exemple repris dans la Figure 3.4, le bloc SysML itsHydrophone possède un flowport de sortie, il s'agit donc d'un émetteur pour le flux. Le bloc SysML itsItSystem possède un flowport d'entrée, il s'agit donc d'un récepteur pour le flux. Nous définissons des rôles Emitter et Receiver pour pouvoir interpréter les blocs SysML itsHydrophone et itsItSystem suivant l'explication précédente. Une instance du rôle Emitter est associée au bloc itsHydrophone et une instance du rôle Receiver est associée au bloc its ItSystem. Cette situation est illustrée Figure 3.4a[3]. Au sein du modèle eFFBD, la fonction Hydrophone émet un flux destiné à la fonction ItSystem. Par conséquent, la fonction eFFBD Hydrophone joue également le rôle Emitter tandis que la fonction eFFBD ItSystem joue également le rôle Receiver comme illustré Figure 3.4b. Les rôles Emitter et Receiver sont donc associés à des éléments de types différents.

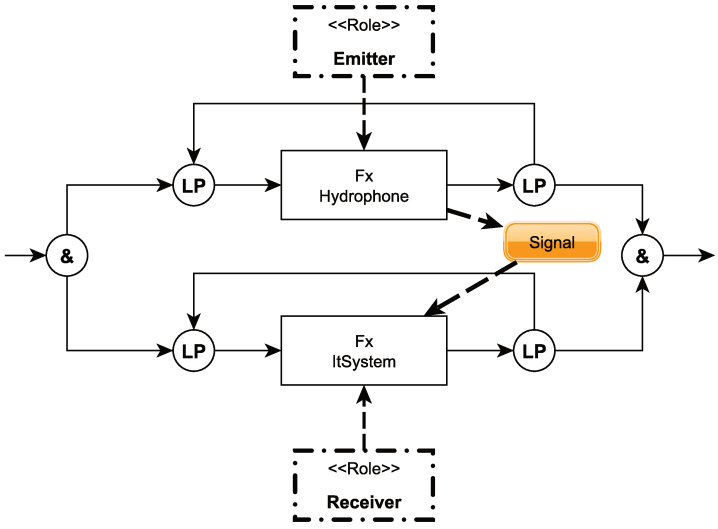
Un rôle possède ses propres caractéristiques et comportements (Propriété 2)

Un exemple d'utilisation de cette propriété est de permettre d'ajouter de l'information aux modèles système sans les modifier. Par exemple, nous souhaitons pouvoir simuler notre système modélisé à l'aide du langage SysML, repris Figure 3.5, sur une longue période temporelle simulée. Nous souhaitons faire varier son temps moyen entre deux pannes. Cette information n'a pas forcément été modélisée dans les modèles du système. Les rôles fournissent un moyen d'interpréter les éléments de modèles et nous pouvons définir leur structure alors que nous ne

3. Pour des raisons de facilité de représentation, les rôles ont été représentés sous la forme de boîtes avec le stéréotype «Role». Toutefois, nous n'avons pas utilisé le mécanisme de stéréotypage.



(a) Association des rôles Emitter et Receiver avec les blocs SysML.



(b) Association des rôles Emitter et Receiver avec les fonctions eFFBD.

FIGURE 3.4. – Association des rôles Emitter et Receiver sur des objets définis dans les langages SysML et eFFBD.

pouvons pas modifier les modèles systèmes qui sont sous la responsabilité d'équipe de conception externes. L'information sur le temps moyen entre deux pannes (MTBF[4]) peut donc être ajoutée aux rôles qui sont associés aux blocs SysML. Les outils accèdent au MTBF défini dans les rôles, qui agissent comme une interface vers les modèles, comme s'il s'agissait d'une propriété des éléments de modèles. Cette situation est illustrée Figure 3.5.

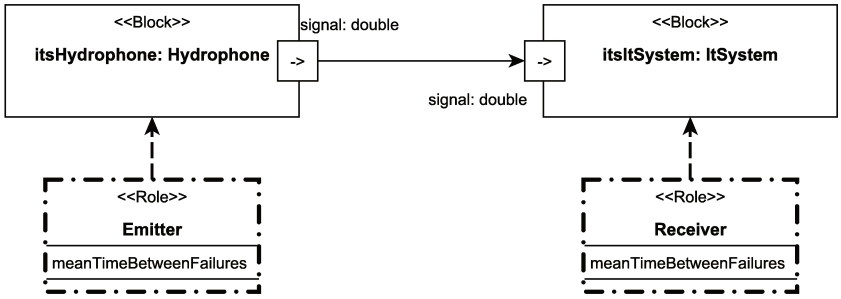


FIGURE 3.5. – Rôles Emetteur et Recepteur possèdant un attribut meanTimeBetweenFailures pour pallier à son absence dans les blocs modélisés dans le langage SysML de l'exemple

Un objet peut jouer différents rôles simultanément (Propriété 3)

Cette propriété permet d'interpréter les mêmes éléments de modèles de manières différentes. A partir de notre exemple précédent, repris Figure 3.6, le bloc itsHydrophone joue le rôle d'Emetteur et le bloc itsItSystem joue le rôle de Récepteur. Ces rôles sont adaptés à une analyse des flux produits et consommés dans le système. Ils permettent de s'assurer de la cohérence de la conception en s'assurant que tous les flux produits sont consommés et que tous les flux qui doivent être consommés sont bien produits. Dans le cas de la définition d'une structure de découpage du projet, plus communément appelée en anglais *Work Breakdown Structure (WBS[5])*, il est nécessaire de définir la traçabilité entre le modèle et la documentation de conception du système. De ce point de vue, les blocs itsHydrophone et itsItSystem sont des sous-systèmes constituants le SdS. Le rôle de sous-système doit également être affecté aux blocs. Ce rôle permettra de générer automatiquement la documentation associée au WBS.

La Figure 3.6 présente notre modèle SysML sur lequel des rôles Subsystem ont été affectés aux blocs en plus de leur rôle d'Emitter et de Receiver.

Un objet peut jouer plusieurs fois le même rôle simultanément (Propriété 4)

Cette propriété nous permet d'interpréter plusieurs fois et de la même manière le même élément de modèle. Dans notre exemple repris Figure 3.7, le diagramme IBD SysML fixe le nombre d'hydrophone à 1. Or, nous pouvons vouloir simuler un nombre supérieur d'instances d'hydrophone. Nous ne pouvons pas modifier directement le modèle SysML car il est sous la responsabilité d'une équipe tierce. De plus, la simulation peut être prospective, c'est-à-dire évaluer une alternative d'architecture qui n'a pas fait l'objet d'une étude préalable et ne sera peut être pas retenue. Pour permettre cela, le rôle Emitter est associé deux fois au bloc

4. Contrairement à l'usage, nous utilisons l'acronyme anglais.
5. Contrairement à l'usage, nous conserverons l'acronyme anglais.

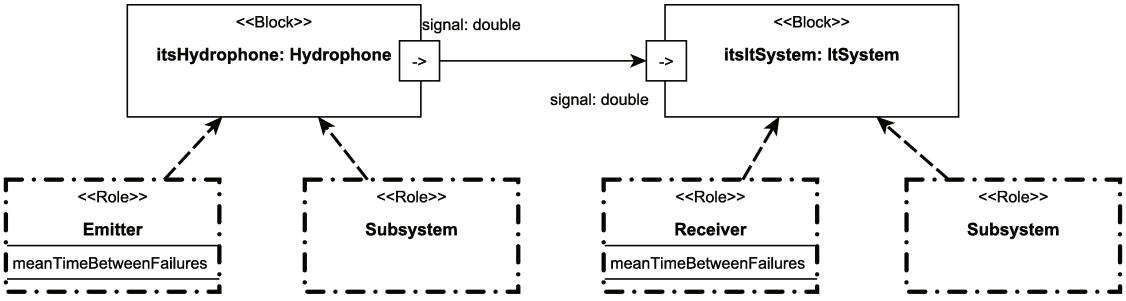


FIGURE 3.6. – Rôle Subsystem associé aux blocs SysML en plus des rôles Emiter et Receiver.

itsHydrophone. Ce dernier sera interprété deux fois sous la forme d'un Emitter. La Figure 3.7 illustre l'ajout de la deuxième instance du rôle Emitter.

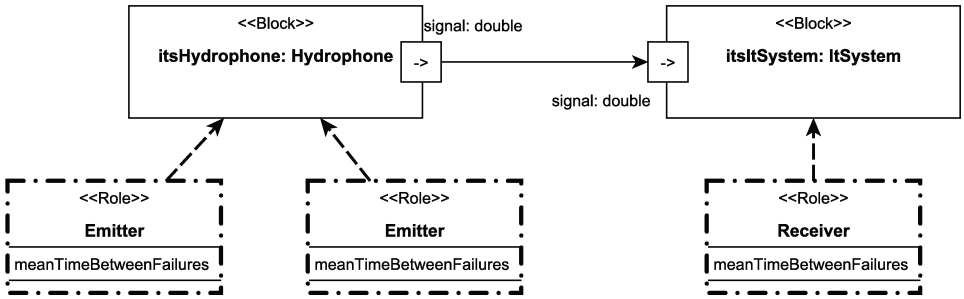


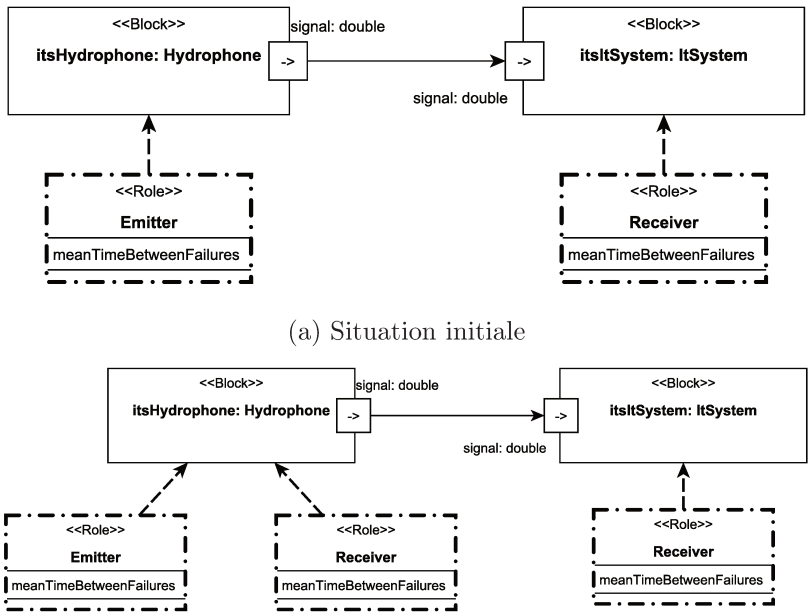
FIGURE 3.7. – L'objet itsHydrophone joue deux fois le rôle d'Emitter.

Un objet peut acquérir ou abandonner des rôles dynamiquement (Propriété 5)

Cette propriété nous permet d'ajouter des interprétations à des éléments de modèles dynamiquement. Dans notre exemple repris Figure 3.8, nous souhaitons ajouter temporairement une source sonore qui est générée par le moteur de simulation. Le bloc itsHydrophone doit alors devenir un récepteur de signal. Par conséquent, nous devons lui ajouter le rôle Receiver alors que cela n'était pas prévu au départ. De plus, nous devons pouvoir lui retirer ce rôle car l'expérimentation est limitée dans le temps. La Figure 3.8 illustre l'acquisition et l'abandon du rôle Receiver par le bloc itsHydrophone. L'association entre les objets et les rôles va pouvoir osciller entre la situation initiale Figure 3.8a qui est la situation de référence et la situation de l'expérimentation Figure 3.8b.

Les rôles peuvent jouer des rôles (Propriété 6)

Cette propriété permet d'enchaîner les interprétations des éléments de modèles. Elle permet d'interpréter les modèles d'une première manière et de réutiliser cette interprétation dans un autre contexte. Par exemple, la simulation basée agent permet de simuler des systèmes complexes constitués d'éléments en interactions [93]. Dans notre modèle SysML repris Figure 3.9, les éléments qui sont en interactions sont le bloc itsHydrophone et le bloc itsItSystem. Dans des modèles plus gros, de nombreux éléments peuvent exister qui n'ont pas d'intérêt à être simulés



(a) Situation initiale

(b) Acquisition dynamique du rôle Receiver par le bloc itsHydrophone

FIGURE 3.8. – Bloc SysML itsHydrophone acquérant et abandonnant le rôle Receiver

avec des agents car ils n'ont pas d'interactions. Les rôles Emitter et Receiver définissent des éléments qui sont en interaction. Nous pouvons leur associer le rôle d'Agent. La hiérarchie de rôles ainsi établie permet de simuler sous la forme d'agents uniquement les éléments du modèle SysML qui ont des interactions. Cette situation est illustrée Figure 3.9.

Les rôles restreignent l'accès aux objets (Propriété 7)

Les rôles servent d'intermédiaire entre les outils de simulation et les modèles. Ils fournissent une adaptation des modèles au point de vue des outils de simulation. Par conséquent, les rôles permettent de sélectionner les données dans les modèles tout en offrant un point d'accès aux propriétés des éléments de modèles. Par exemple dans notre modèle SysML, nous pouvons récupérer les flux émis par un émetteur peut importe le type de l'émetteur. Le rôle masque l'implémentation de la description des flux dans les modèles. Cette situation est illustrée Figure 3.10.

Différents rôles peuvent partager des aspects structurels et comportementaux (Propriété 8)

Cette propriété permet de définir des spécialisations ou des généralisations de rôles. Dans notre exemple repris Figure 3.11, nous pouvons préciser dans l'interprétation du bloc itsHydrophone que le comportement est celui d'un émetteur continu. Nous avons déjà défini le rôle Emitter qui permet de préciser le fait qu'un objet émet des informations. Emettre en continu est une spécialisation de ce comportement d'émetteur. Par conséquent, nous définissons le rôle ContinuousEmitter comme spécialisation du rôle Emitter tel qu'illustré Figure 3.11a et nous l'associons au bloc itsHydrophone comme illustré Figure 3.11b.

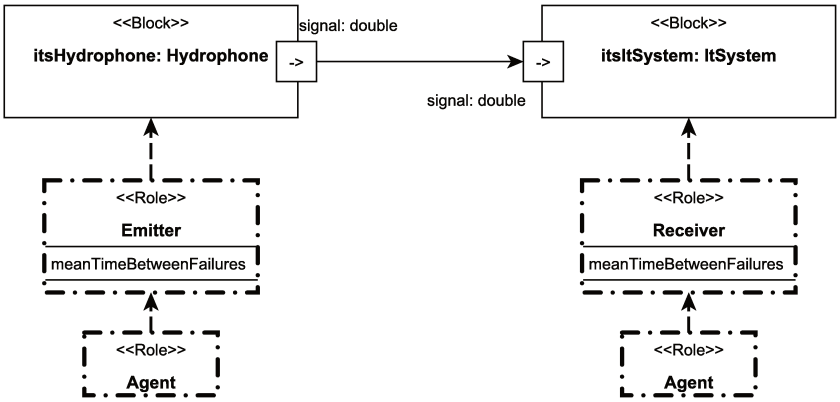


FIGURE 3.9. – Application du rôle Agent sur les rôles Emetteur et Récepteur appliqués à des blocs SysML

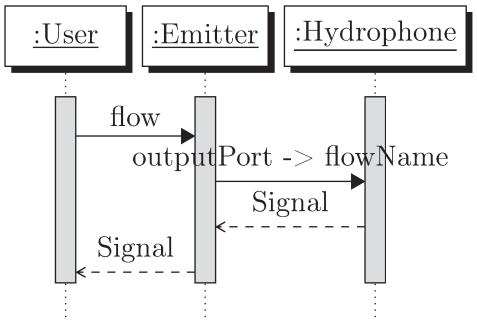
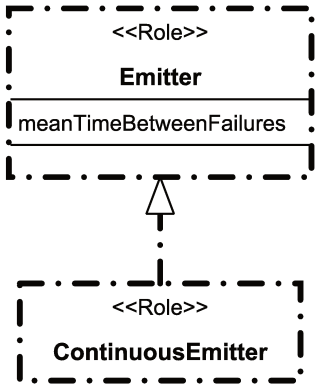
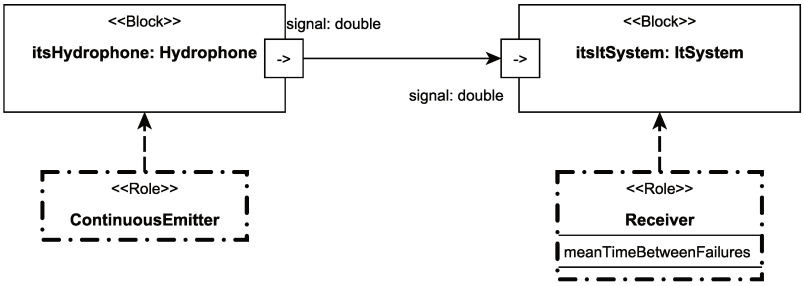


FIGURE 3.10. – Comportement d'un rôle comme interface d'accès pour un élément de modèle.



(a) Le rôle ContinuousEmitter spécialise le rôle Emitter



(b) Le rôle ContinuousEmitter est associé au bloc itsHydrophone

FIGURE 3.11. – Spécialisation du rôle Emitter par le rôle ContinuousEmitter.

Un objet et ses rôles ont des identités différentes (Propriété 9)

Cette propriété implique que les rôles ont leur propre identité. Il est donc possible de les identifier pour les manipuler. Cette propriété est utile lorsque le même rôle est associé plusieurs fois au même objet. Dans notre exemple, nous avons associé plusieurs fois le rôle Emitter au bloc itsHydrophone. L'objectif est d'émuler la présence de plusieurs sources de données pour le système d'information sans modifier le modèle. Chaque pseudo-source de données est identifiée de manière unique grâce à l'identité des rôles.

L'association d'un rôle et d'un objet peut être contrainte (Propriété 10)

Les rôles fournissent une interprétation des éléments de modèles. Le choix d'une interprétation pour un objet peut dépendre de ses caractéristiques. Dans notre exemple repris Figure 3.12, nous avons associé le rôle Emitter avec le bloc itsHydrophone. Nous avons fait ce choix parce que le bloc itsHydrophone possède un flowport de sortie. L'association du rôle Emitter avec le bloc itsHydrophone est bien conditionné par la possession par le bloc itsHydrophone d'un flowport de sortie.

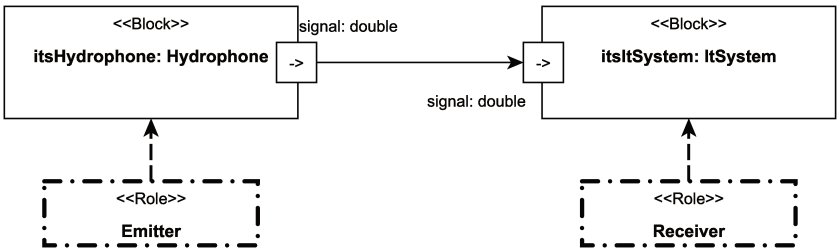


FIGURE 3.12. – Rôles Emitter et Receiver associés aux blocs SysML possèdant un port de sortie et d'entrée.

Les rôles peuvent être groupés (Propriété 11)

Suivant l'analyse d'un modèle SysML, les rôles d'Emetteur et de Récepteur ne doivent pas forcément être utilisés directement. L'analyse de dépendances permet de déterminer le nombre de relations qu'une entité modélisée entretien avec les autres entités. Durant cette analyse, le rôle de connexion est mis en avant. Ce rôle ne peut cependant pas exister sans les rôles d'Emetteur et de Récepteur illustrés Figure 3.13. Le rôle de Connexion contient les rôles d'Emetteur et de Récepteur. Les outils d'analyse pourront utiliser ce rôle sans avoir besoin de connaître les rôles d'Emetteur et de Récepteur qui seront utilisés au travers du rôle Connection. La Figure 3.13 illustre la définition du rôle Connection.

Les rôles dépendent d'un contexte (Propriété 12)

L'utilisation des rôles dépend d'un objectif et donc d'un contexte. Les informations qui sont spécifiques à un objectif sont :

— les rôles utilisés pour remplir l'objectif,
— les associations des rôles avec des éléments de modèles.

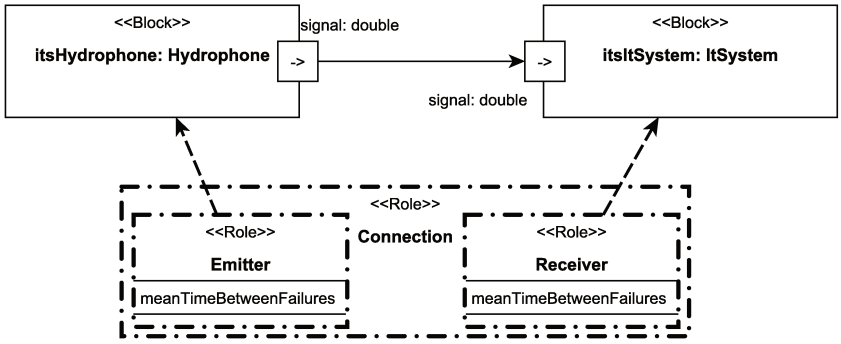


FIGURE 3.13. – Définition du rôle Connection qui regroupe les rôles Emitter et Receiver associés aux blocs itsHydrophone et itsItSystem.

Pour atteindre un objectif, il est nécessaire de définir un point de vue sur les modèles qui correspond à l'objectif. Les rôles permettent de définir ce point de vue. Le contexte doit donc identifier les rôles qui sont utilisés.

Tous les éléments de modèles ne sont pas associés aux mêmes rôles. Par conséquent, un contexte doit permettre d'identifier les objets qui sont associés aux rôles. Comme le même contexte peut être utilisé sur différents modèles, cette identification doit être générique par exemple sous la forme de règles d'association de rôles avec des éléments de modèles.

Dans notre exemple, nous souhaitons simuler les blocs SysML en adoptant un point de vue interaction en identifiant les émetteurs et les récepteurs d'information. Nous avons définis un contexte qui liste :

— les rôles Emitter et Receiver comme les rôles à utiliser dans ce contexte,
— la condition d'association du rôle Emitter avec les blocs SysML : le bloc doit posséder un port de sortie,
— la condition d'association du rôle Receiver avec les blocs SysML : le bloc doit posséder un port d'entrée.

La définition d'un contexte permet d'indiquer l'objectif suivi par l'utilisation des rôles. La définition de ce contexte permet également de réutiliser notre expérience avec la manipulation de modèles SysML avec d'autres modèles SysML que notre modèle d'exemple dans le cadre du même objectif.

Un rôle peut faire partie de plusieurs contextes (Propriété 13)

Les rôles peuvent être utilisés dans plusieurs contextes en même temps. Dans notre exemple, nous avons utilisé les rôles d'Emétteur et Récepteur pour faire une simulation basée agents du système. Cela constitue un premier contexte dont l'objectif est une analyse dynamique des modèles. Nous avons également défini un rôle Connexion qui groupe les rôles Emétteur et Récepteur afin de faire une analyse de la structure du modèle en vérifiant que tous les émetteurs d'informations sont liés à des récepteurs. Cette analyse constitue un second contexte qui lui concerne une analyse statique des modèles. Les deux contextes, qui se différencient sur les aspects dynamiques ou statiques de l'analyse, reposent pourtant tous les deux sur les mêmes rôles.

Les contextes ont leur propre identité (Propriété 14)

Les contextes doivent pouvoir être identifiés et réutilisés. Dans notre exemple, notre contexte pour la simulation basée agents possède un identifiant qui permet de le distinguer du contexte d'analyse statique des modèles.

Nous avons détaillé dans cette section les choix des caractéristiques des rôles qui nous intéressent pour notre contexte de manipulation de modèles système et de simulation hétérogènes sans modifier les modèles ou les langages de modélisation. Nous avons également illustré l'usage de ces propriétés dans ce contexte.

L'utilisation des rôles doit maintenant être intégrée dans une approche basée modèles de l'ingénierie système. Le concept de rôle et ses propriétés doivent être exprimés de sorte à pouvoir être utilisés avec des modèles système différents sans modifier ces modèles ou la définition des langages de modélisation utilisés pour les créer.

3.4. Langage de rôle [6]

Intégrer le concept de rôle dans les modèles système peut être réalisé de plusieurs manières :

— en complétant les méta-modèles (les définitions) des langages de modélisation avec le concept de rôle,
— en utilisant des concepts déjà existants dans les langages de modélisation, en les spécialisant (à l'image du mécanisme de stéréotypes) et en ajoutant les éléments nécessaires dans les modèles,
— en définissant un langage de modélisation spécifique (et donc un méta-modèle).

Compléter les langages de modélisation avec le concept de rôle implique de modifier les méta-modèles des langages de modélisation. Or, les langages de modélisation système sont souvent standardisés et ne peuvent pas être modifiés.

Utiliser des concepts existants dans les langages de modélisation pour ajouter les rôles directement dans les modèles système implique de modifier ces modèles. Or ces modèles sont réalisés par des équipes différentes de celle(s) qui utilisent les rôles. Ces modèles ne peuvent donc pas être modifiés.

Nous avons donc choisi la troisième manière en définissant un langage dédié pour la modélisation de rôles et leur association avec des éléments de modèles issus de modèles système. Nous avons donc créé un méta-modèle pour définir ce langage.

Dans cette section, nous présentons le méta-modèle en mettant en lumière les choix d'architecture du méta-modèle en fonction de chaque propriété. Les propriétés des rôles ont été classées en trois catégories en fonction du ou des concept(s) qu'elles concernent :

— structure d'un rôle,
— relation entre rôle et objet,
— contexte pour les rôles.

Le processus de construction du méta-modèle a abouti à celui présenté Figure 3.14 donné ici à titre informatif. Sa construction complète est détaillée dans les différentes sous-sections.

6. Ce langage a été présenté dans [80] publié dans les proceedings de la conférence IEEE SysCon 2015

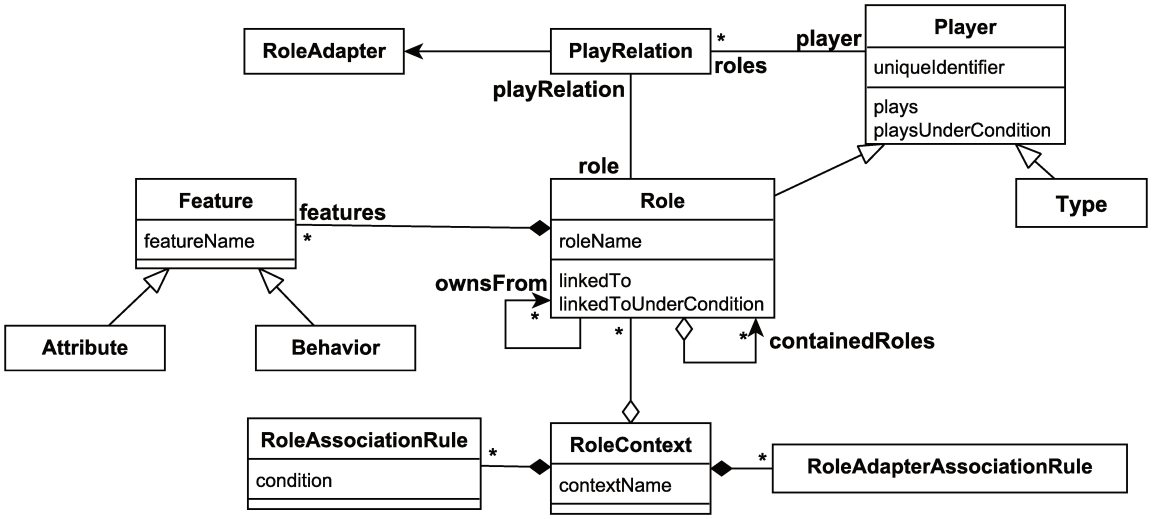


FIGURE 3.14. – Méta-modèle complet du langage de rôle.

3.4.1. Définitions préliminaires

Un rôle est associé à un objet. Nous associons des rôles avec des éléments de modèles. Les modèles sont définis à l'aide de méta-modèles. Les éléments de modèles sont des instances des méta-classes qui forment les méta-modèles. Nous utilisons donc la Définition 16 pour la notion d'objet dans le cadre de nos travaux.

Définition 16: Objet

Un objet est une instance d'une méta-classe.

Nous définissons un langage pour la modélisation et l'association de rôles à des modèles. Ce langage est défini par un méta-modèle dont le concept central, celui de rôle, sera donc modélisé sous la forme d'une méta-classe nommée Role. Dans le cadre de nos travaux nous utilisons donc la Définition 17 pour la notion de rôle.

Définition 17: Rôle

Un rôle est une instance de la méta-classe Role.

Objet et rôle sont des instances de méta-classes qui sont associées. Cette association est nommée dans la littérature sur les rôles *jouer un rôle*. Dans le cadre de nos travaux, nous utilisons la Définition 18.

Définition 18: Jouer un rôle

La relation jouer un rôle pour un objet signifie que l'objet est associé à un rôle.

Les rôles sont des instances d'une méta-classe. De plus, les associations entre les rôles et les objets sont également le sujet du langage de modélisation que nous avons créé et défini par un méta-modèle. Par conséquent, les rôles et leurs associations forment un modèle. Dans le cadre de nos travaux, nous utilisons la Définition 19.

Définition 19: Modèle de rôles

Un modèle de rôle est un ensemble d'instances de la méta-classe Role associées à des objets.

3.4.2. Structure d'un rôle

Les propriétés qui concernent la structure d'un rôle sont :

Propriété 2 Un rôle possède ses propres caractéristiques et comportements.

Propriété 8 Différents rôles peuvent partager des aspects structurels et comportementaux.

Propriété 11 Les rôles peuvent être groupés.

Notre langage se fonde sur la notion de rôle, nous avons donc créé une méta-classe Role pour permettre de représenter ce concept. Un rôle porte un nom. La méta-classe Role possède un attribut *roleName* pour stocker cette information.

D'après la Propriété 2, un rôle doit pouvoir posséder des caractéristiques et des comportements qui lui sont propres. Nous avons défini les méta-classes Attribute et Behavior pour représenter les concepts de caractéristiques et de comportements. Ces concepts forment les propriétés des rôles, nous avons donc défini la méta-classe Feature qui généralise les méta-classes Attribute et Behavior. La méta-classe Feature possède un attribut *featureName* qui permet d'identifier une propriété. La méta-classe Role possède une relation de composition nommée *features* avec la méta-classe Feature. Cette relation possède une multiplicité de aucun ou plusieurs notée * qui permet à un rôle de posséder un nombre indéterminé de propriétés. Le méta-modèle obtenu est présenté Figure 3.15.

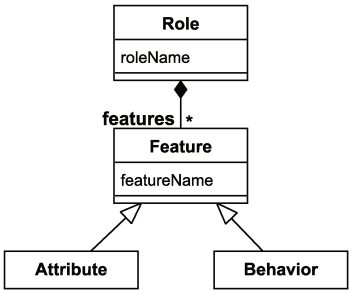


FIGURE 3.15. – Méta-classe Role possèdant des propriétés.

D'après la Propriété 8, les rôles peuvent partager des propriétés. Nous avons modélisé cette relation de partage en s'inspirant de la relation d'héritage qui existe dans les langages orientés objets. Nous avons créé une association nommée *ownsFrom* entre la méta-classe Role et elle-même tel qu'illustré Figure 3.16. Cette association permet à une instance de la méta-classe Role d'accéder aux propriétés d'une autre instance de la méta-classe Role.

D'après la Propriété 11, les rôles peuvent être groupés. Un rôle peut ainsi être constitué de sous-rôles à la manière d'un agrégat. En effet, un rôle peut appartenir à plusieurs groupes. Cette propriété implique la définition d'une relation d'agrégation entre la méta-classe Role et elle-même. La relation d'agrégation permet de définir des rôles agrégats. Un rôle agrégat possède les mêmes capacités qu'un rôle classique, il peut ainsi être associé à un élément de modèle et

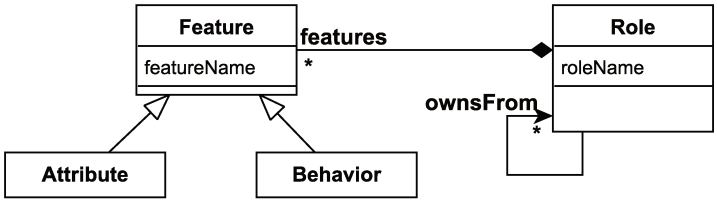


FIGURE 3.16. – Relation *ownsFrom* modélisant le partage de propriétés entre rôles.

fournir un accès vers cet élément. De plus, le rôle agrégat a la capacité de déléguer une partie de ses accès aux rôles qu'il agrège. Le rôle agrégat fournit une interface de plus haut niveau sur les agrégés. La Figure 3.17 présente cette relation.

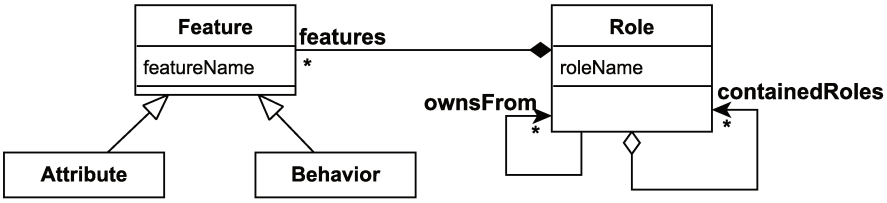


FIGURE 3.17. – Relation d'agrégation définie sur la méta-classe Role.

L'instanciation de la méta-classe Role permet de définir des rôles qui seront associés à des éléments de modèles. La sous-section suivante permet de préciser la nature de la relation entre les rôles et les éléments de modèles auxquels ils doivent être associés.

3.4.3. Relation entre rôle et objet

Les propriétés qui concernent la relation entre rôles et objets sont :

Propriété 1 Des objets de types différents peuvent jouer le même rôle.

Propriété 3 Un objet doit pouvoir jouer des rôles différents.

Propriété 4 Un objet peut jouer plusieurs fois le même rôle simultanément.

Propriété 5 Un objet peut acquérir ou abandonner des rôles dynamiquement.

Propriété 6 Les rôles peuvent jouer des rôles.

Propriété 7 Les rôles restreignent l'accès aux objets.

Propriété 9 Un objet et ses rôles ont des identités différentes.

Propriété 10 L'association d'un rôle et d'un objet peut être contrainte.

Nous utilisons les rôles pour pouvoir définir des interprétations sur des éléments de modèles afin de pouvoir utiliser les modèles avec des outils qui n'ont pas été prévus pour travailler avec ces modèles. Les modèles sont constitués d'éléments qui sont nos objets. Ces objets sont des instances de méta-classes qui sont définies dans les méta-modèles des langages de modélisation. Nos rôles sont des instances de la méta-classe Role définie dans notre langage. Dire qu'un objet joue un rôle revient à créer une relation entre l'objet et une instance de la méta-classe Role.

Cette relation s'exprime donc par la création d'une association entre la méta-classe Role et la méta-classe Type qui est une méta-classe générique et abstraite représentant toutes les méta-classe potentielles définies dans les méta-modèles des langages de modélisation. Cette situation est illustrée Figure 3.18.

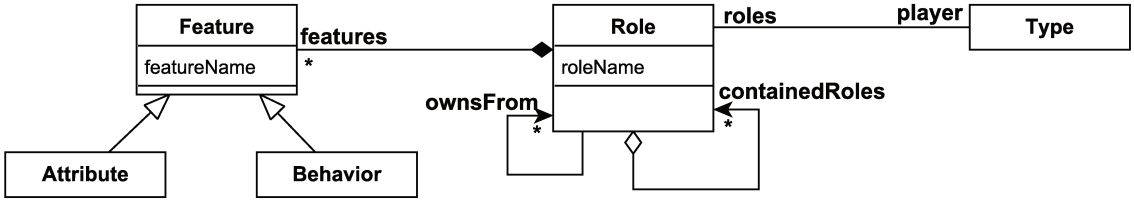


FIGURE 3.18. – Association entre les méta-classes Role et Type.

L'association est choisie bi-directionnelle pour permettre au rôle d'accéder aux propriétés de l'objet auquel il est associé et à l'objet de savoir quels rôles il joue.

La méta-classe Type présentée Figure 3.18 est une abstraction pour toutes les méta-classes définies dans les langages de modélisation. Créer une association entre la méta-classe Role et la méta-classe Type permet au rôle (c'est-à-dire aux instances de la méta-classe Role) d'être indépendants des méta-classes concrètes qui définissent les types des objets. Il est donc possible d'associer un rôle avec des éléments de types différents. En effet, la méta-classe Role n'est pas associée à une méta-classe spécifique. Par conséquent, ses instances peuvent être associées à n'importe quel objet.

Un rôle porte un nom qui est modélisé par l'attribut *roleName* dans la méta-classe Role. Dire qu'*un objet joue un rôle A* signifie que l'objet est associé à une instance de la méta-classe Role dont l'attribut *roleName* a pour valeur *A*.

Des objets, instances de méta-classes différentes doivent pouvoir jouer le même rôle. Ces objets sont donc associés à des instances de la méta-classe Role dont l'attribut *roleName* possède la même valeur.

Un objet peut être associé à plusieurs rôles

Un objet peut jouer des rôles différents ou plusieurs fois le même rôle. Ces propriétés sont ajoutées au méta-modèle sous la forme d'une multiplicité aucun ou plusieurs notée * sur la terminaison nommée *roles* entre la méta-classe Role et la méta-classe Type. Ceci est illustré Figure 3.19.

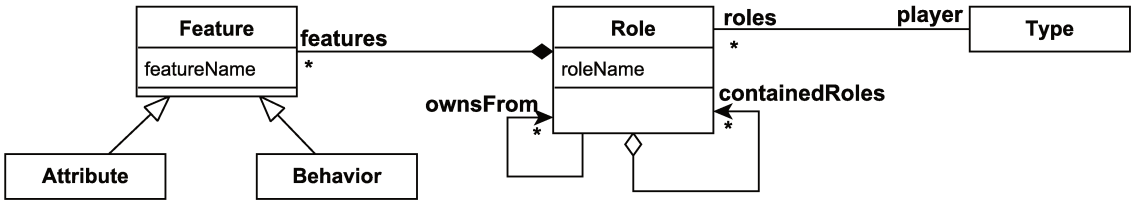


FIGURE 3.19. – Ajout de multiplicité pour permettre à un élément de modèle de jouer plusieurs rôles

Cette multiplicité permet à une instance de Type (c'est-à-dire un objet) d'être associée à plusieurs instances de la méta-classe Role. Nous avons sélectionné la multiplicité *0..** car elle ne force pas un objet à jouer un rôle. L'utilisation d'une multiplicité *de un à plusieurs* aurait eu pour effet de forcer tous les objets à jouer au moins un rôle. Or les rôles fournissent un point de vue sur les éléments de modèles, ainsi tous les éléments qui ne concernent pas le point de vue ne doivent pas être associés à des rôles.

Un objet peut jouer différents rôles implique que l'objet est associé à des instances de la méta-classe Role qui possèdent des valeurs différentes pour l'attribut *roleName*. Un objet peut jouer plusieurs fois le même rôle implique que l'objet est associé à des instances de la méta-classe Role qui possèdent la même valeur pour l'attribut *roleName*.

Les rôles peuvent jouer des rôles

Un rôle possède la propriété de pouvoir également jouer des rôles. Les instances de la méta-classe Type peuvent déjà jouer des rôles. Cependant, un rôle ne définit pas un type. Le type d'un élément définit sa nature immuable alors que le rôle permet de définir un point de vue sur un objet qui est dynamique [88, 54]. Par conséquent, nous ne pouvons pas créer un héritage entre les méta-classes Type et Role de notre méta-modèle pour permettre aux rôles de jouer des rôles.

Toutefois, si les rôles peuvent jouer des rôles, ils possèdent un comportement commun avec les instances de Type. Ce comportement peut être défini dans une méta-classe plus abstraite que les méta-classes Role et Type. Cette méta-classe abstraite possède des relations d'héritage avec les méta-classes Role et Type. Cette situation est illustrée Figure 3.20.

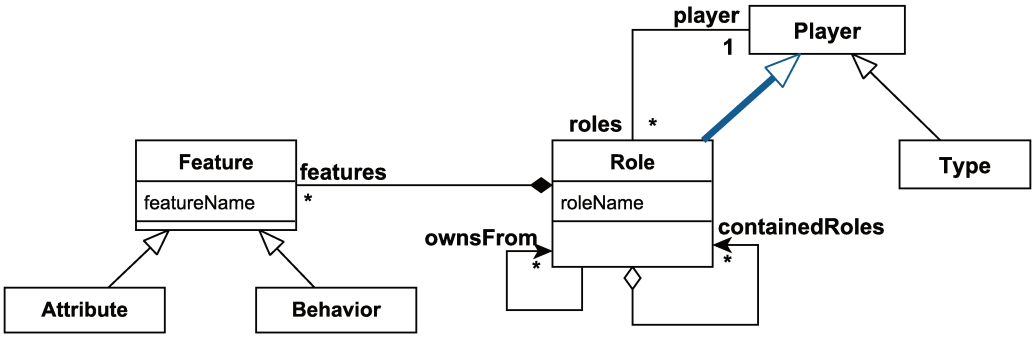


FIGURE 3.20. – Héritage entre les méta-classes Player et Role pour que des rôles puissent jouer des rôles.

La méta-classe Player est une abstraction pour toutes les méta-classes dont les instances vont pouvoir jouer des rôles. La méta-classe Player définit notamment les comportements qui autorisent les instances d'une méta-classe à jouer un rôle. Créer un héritage entre les méta-classes Player et Role (respectivement Type) donne aux instances de la méta-classe Role (resp. Type) la possibilité de jouer des rôles.

Propriétés des Player

La Propriété 9 implique que les rôles et les objets possèdent un identifiant qui permet de distinguer chaque instance. Comme à la fois rôles et éléments de modèles doivent posséder un

identifiant, celui-ci peut être placé dans une méta-classe commune. Un attribut uniqueIdentifier a ainsi été ajouté à la méta-classe Player. Cet attribut est hérité par toutes les sous-classes de la méta-classe Player. La Figure 3.21 illustre la modification du méta-modèle pour répondre à cette propriété.

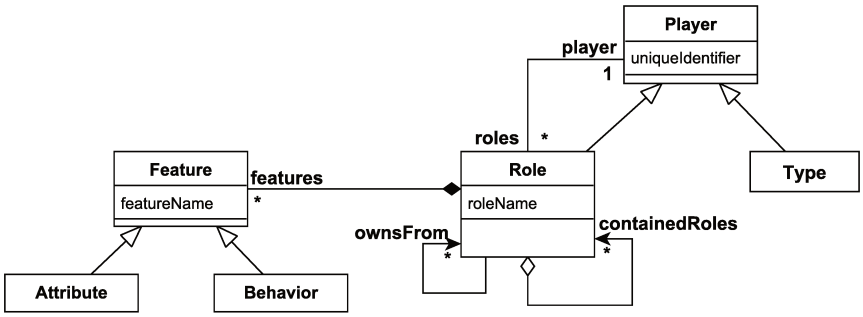


FIGURE 3.21. – Attribut uniqueIdentifier fournissant une identité propre à toutes les instances de rôles et d'éléments de modèles

Réification de la relation entre Role et Player

La relation entre un rôle et un objet peut être créée ou supprimée dynamiquement. Ainsi, nous devons être capable de créer des associations entre les instances de la méta-classe Role et de la méta-classe Player à la volée.

De plus, un rôle donne accès aux propriétés d'un objet selon un point de vue donné. Il faut donc adapter les propriétés de l'objet au point de vue associé au rôle. Cependant, le rôle doit rester indépendant de l'objet car le même rôle peut être associé à des objets de types différents et que les relations entre un rôle et un objet peuvent être créées dynamiquement. Le comportement d'adaptation de l'objet au rôle ne peut donc pas être associé au rôle.

La méta-classe Role doit :

— permettre de définir des comportements qui accèdent à une instance de Player,
— permettre de déléguer ces comportements à une instance externe pour garantir l'indépendance du rôle par rapport à l'instance de Player.

En réifiant la relation entre Role et Player sous la forme de la méta-classe PlayRelation, la relation peut prendre en charge la délégation du comportement du rôle. En effet, la relation connaît à la fois l'instance de Role et l'instance de Player. Cependant, ce comportement délégué est spécifique au couple (Role, Player).

Cepdendant, associer ce comportement à la méta-classe PlayRelation impliquerait de définir des méta-classes de relation spécifiques. Ces méta-classes devraient être spécifiées à chaque fois qu'une relation entre une instance de Role et une instance de Player est créée.

Pour conserver l'aspect générique de la relation, une méta-classe RoleAdapter est définie ainsi qu'une association entre les méta-classes PlayRelation et RoleAdapter. Les instances de la méta-classe RoleAdapter fournissent le comportement spécifique permettant de conserver les méta-classes Role et PlayRelation indépendantes du Player. Le méta-modèle ainsi modifié est présenté Figure 3.22.

La méta-classe RoleAdapter est associée aux comportements d'adaptation d'un objet à un rôle. Tous les comportements particuliers sont regroupés dans le RoleAdapter. Le rôle n'a

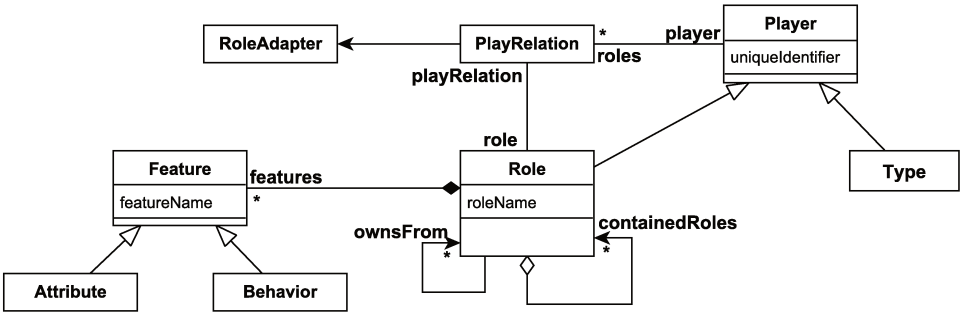


FIGURE 3.22. – Réification de la relation entre les méta-classes Player et Role et ajout de la méta-classe RoleAdapter fournissant les comportements d'interprétation des rôles.

donc pas besoin d'être modifié lors d'une association avec un objet et peut donc être utilisé indifféremment avec des objets de types différents.

L'ajout de la méta-classe RoleAdapter permet de mettre en place le patron chaîne de responsabilité [30] illustré Figure 3.23 qui reprend l'exemple fourni par la Figure 3.10. L'appel à

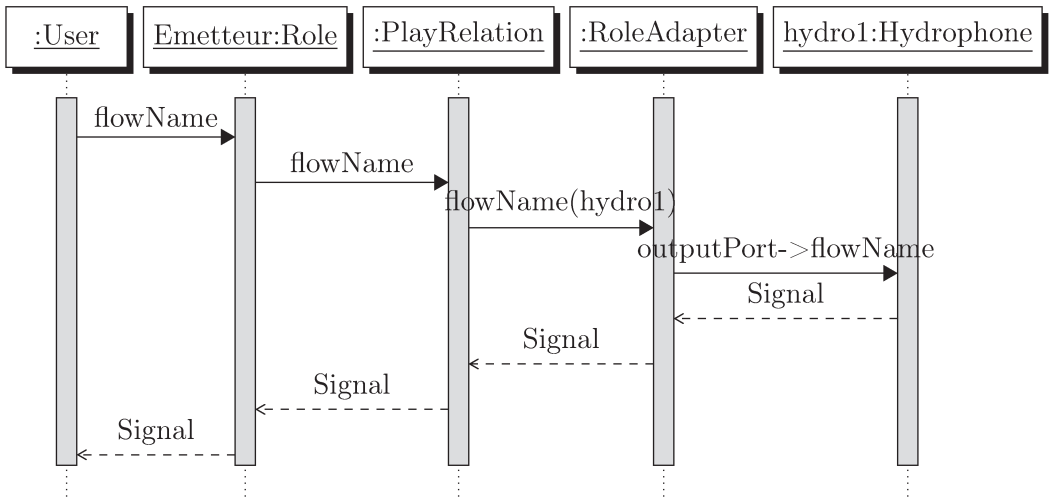


FIGURE 3.23. – Illustration du patron chaine de responsabilité.

flowName réalisé par le rôle Emetteur est cette fois réalisé sur l'instance de PlayRelation. Cet appel est relayé à l'instance de RoleAdapter qui peut réaliser l'appel nécessaire sur l'instance d'Hydrophone. Le rôle est donc indépendant du type de l'élément de modèle auquel il est lié. Seule l'instance de RoleAdapter est particulière au couple représenté par l'instance de Role et de Player.

De plus, le patron chaîne de responsabilité permet de changer l'instance de RoleAdapter et donc l'adaptation de l'élément de modèle au rôle. En effet, l'adaptation d'un objet à un rôle peut évoluer au cours du temps à l'image des langages de modélisation et des outils de simulation. Par conséquent, les instances de RoleAdapter doivent être modifiées sur les associations entre rôle et objet.

La Propriété 10 implique que la construction de l'association entre un rôle et un objet peut

être conditionnée. La définition de l'association peut être réalisée soit du point de vue de l'objet soit du point de vue du rôle. Cela se traduit par la mise à disposition par les méta-classes Role et Player de méthodes qui permettent de spécifier l'existence d'une relation entre un rôle et un objet. Ces méthodes comprennent :

— une méthode d'association sans condition,
— une méthode d'association sous condition.

La méta-classe Player (respectivement Role) se voit ajouter les méthodes *plays* (resp. *linkedTo*) pour spécifier une relation sans condition et *playsUnderCondition* (resp. *linkedToUnderCondition*) pour spécifier une relation sous condition. La Figure 3.24 représente le méta-modèle ainsi modifié.

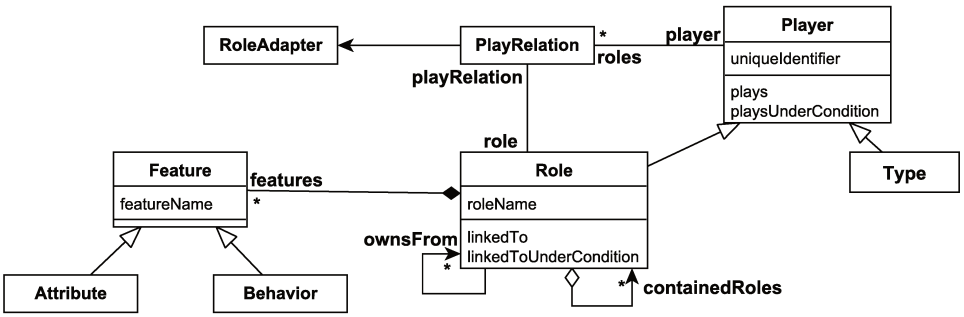


FIGURE 3.24. – Ajout de méthodes pour spécifier la relation entre Role et Player

3.4.4. Contexte pour les rôles

Les propriétés qui concernent les contextes pour les rôles sont :

Propriété 10 L'association d'un rôle et d'un objet peut être contrainte.

Propriété 12 Les rôles dépendent d'un contexte.

Propriété 13 Un rôle peut faire partie de plusieurs contextes.

Propriété 14 Les contextes ont leur propre identité.

D'après la Propriété 12, les rôles peuvent être associés à un contexte. D'après la Propriété 10, des contraintes peuvent être définies pour l'association entre un objet et un rôle. Ces contraintes peuvent dépendre du contexte. D'après la Propriété 13, l'appartenance d'un rôle à un contexte n'est pas exclusive. De plus, d'après la Propriété 14, chaque contexte doit pouvoir être identifié.

Un contexte définit un objectif. Les rôles qui définissent un point de vue spécifique au contexte doivent ainsi être identifiés. De plus, la manière d'associer ces rôles aux objets doit également être spécifiée. Comme nous souhaitons pouvoir utiliser le même contexte sur des modèles différents du même type, nous spécifions l'association entre rôles et objets sous la forme de règles. Dans notre méta-modèle nous avons défini des adaptateurs pour adapter les objets aux rôles qui leur sont associés. Le choix des adaptateurs est spécifique à un contexte. De manière identique à l'association entre rôles et objets, nous souhaitons capitaliser ces choix pour pouvoir les réutiliser avec des modèles différents. Nous avons également défini des règles pour associer des adaptateurs aux relations entre objets et rôles.

Dans notre approche, un contexte d'utilisation des rôles peut ainsi être vu comme la description :

— d'un ensemble de rôles qui peuvent être utilisés dans le cadre du contexte,
— d'un ensemble de règles pour associer les rôles à d'autres éléments de modèles ou d'autres rôles,
— d'un ensemble de règles pour associer les adaptateurs dynamiques aux relations entre des rôles et des éléments de modèles ou des rôles.

Nous avons ainsi défini la méta-classe RoleContext. La méta-classe RoleContext représente le contexte d'utilisation des rôles. Une relation d'agrégation est définie entre RoleContext et Role. Cette relation permet à un RoleContext d'indiquer quels rôles sont associés au contexte défini. Comme un rôle peut être associé à plusieurs contextes, nous avons choisi une relation d'agrégation plutôt qu'une relation de composition.

RoleContext possède deux relations de composition vers les méta-classes RoleAssociationRule et RoleAdapterAssociationRule qui représente les définitions des règles d'association. Le méta-modèle étendu est présenté Figure 3.25.

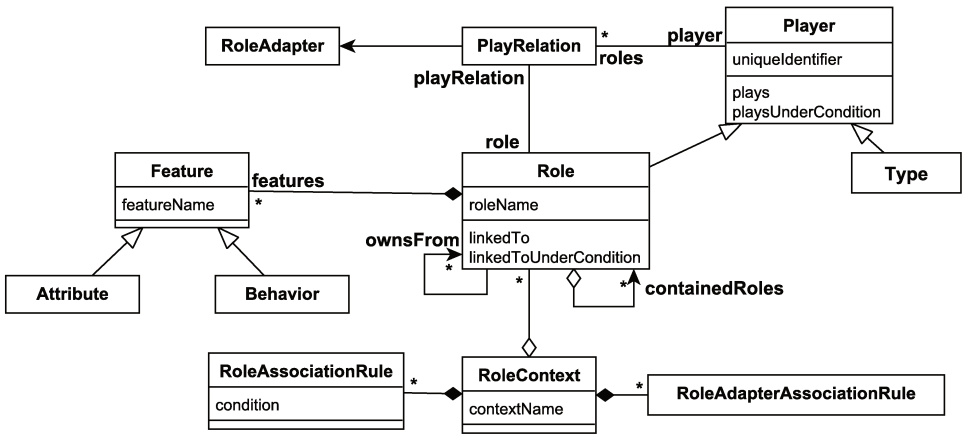


FIGURE 3.25. – Diagramme de classe définissant la structure d'un contexte de rôles

La méta-classe RoleAssociationRule représente une règle d'association d'un rôle avec des objets. La méta-classe RoleAssociationRule est associée aux méta-classes Role et Player. Ces relations permettent de définir une règle d'association entre un rôle et un élément de modèle ou un rôle. Cette relation pouvant être conditionnée, la classe RoleAssociationRule possède un attribut condition. Cet attribut permet de stocker la condition sous laquelle associer un rôle à une instance de Player.

La méta-classe RoleAdapterAssociationRule permet de représenter l'association d'un RoleAdapter à toutes les relations entre un rôle et un objet instance de la méta-classe Player. La méta-classe RoleAdapterAssociationRule possède trois associations vers Role, Player et RoleAdapter. Un RoleAdapter est spécifique à un couple (Role, Player). La combinaison des trois associations de RoleAdapterAssociationRule permet d'associer le RoleAdapter adéquat à un couple (Role, Player) spécifique. La structure de ces classes est présentée Figure 3.26.

Notre méta-modèle permet de définir des rôles et de les associer à des objets. Dans un contexte d'ingénierie de systèmes et de systèmes de systèmes, les objets sont des instances de méta-classes issues de méta-modèles différents. De plus, ces objets sont stockés dans des modèles différents. Ces modèles sont produits par des équipes et des outils différents. Pour

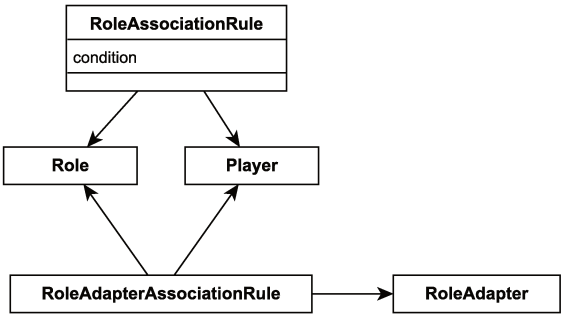


FIGURE 3.26. – Description de la structure des règles d'association des rôles et des adaptateurs dynamiques.

pouvoir manipuler ces modèles à l'aide de rôles, il est nécessaire de lire les différents modèles en gérant leur hétérogénéité en terme de langages de modélisation et d'outils.

3.5. Utilisation d'importateurs modulaires à base de parsers combinateurs [7]

Le méta-modèle de rôles permet de créer des modèles de rôles associés aux objets issus de modèles différents. Dans un contexte d'ingénierie de systèmes de systèmes, des équipes différentes sont impliquées avec chacune leurs méthodes et outils d'ingénierie. De plus, les systèmes conçus ont une longue durée de vie. Ainsi, les systèmes vont évoluer tout comme les méthodes et outils d'ingénierie utilisés pour les créer.

Les modèles utilisés pour la conception d'un système ou d'un système de systèmes sont réalisés dans des langages de modélisation et des outils différents. Des outils différents sont également utilisés pour créer des modèles dans le même langage de modélisation. De plus, les outils de modélisation peuvent devenir obsolètes et ne plus être utilisés et de nouveaux outils vont apparaître. Du fait de l'hétérogénéité et de l'évolution des outils, il n'est pas possible de lier l'environnement Role4All avec les outils de modélisation.

Les problématiques d'hétérogénéité et d'évolution se retrouvent dans les langages de modélisation. En effet, différents langages de modélisation sont utilisés par les équipes de conception et le choix des langages peut changer au cours du temps. Cependant, la définition des langages de modélisation est plus stable que celle des outils de modélisation. En effet, les langages de modélisation sont, pour la plupart standardisés. Les modèles réalisés dans les différents langages de modélisation ont un point commun. Ils sont stockés sous une forme sérialisée. L'accès aux données dans les modèles peut donc se faire en dé-sérialisant les modèles.

La dé-sérialisation, et par conséquent l'import des modèles dans l'environnement Role4All, peut se faire par l'intermédiaire de parsers. Ces derniers prennent en entrée les modèles sérialisés dans un format textuel et permettent de créer les objets auxquels des rôles pourront être associés. Cependant, l'hétérogénéité et l'évolution des modèles nécessitent de disposer d'importateurs modulaires qui permettent de gérer :

— les différents types de modèles qui doivent être importés,

7. L'utilisation des parsers combinateurs pour importer des modèles a été présentée dans [83] publié dans les proceedings du workshop SESoS 2014 et dans [79] publié dans les proceedings de la conférence WETICE 2015

— les évolutions des langages de modélisation,
— l'import de nouveaux types de modèles.

La sous-section 3.5.1 décrit l'utilisation des opérateurs de combinaison de parsers fournis par les parsers combinateurs pour définir le parser d'un langage de modélisation. La sous-section 3.5.2 décrit le gain de modularité apporté par l'utilisation des parsers combinateurs pour permettre d'importer des modèles définis dans des langages de modélisation variés et évolutifs. La sous-section 3.5.3 décrit l'intégration des parsers combinateurs à notre environnement Role4All.

3.5.1. Opérateurs de combinaison de parsers

Les parsers combinateurs définissent un ensemble d'opérateurs de combinaison de parsers. Les opérateurs de combinaison comprennent la séquence, le choix et le choix ordonné.

Posons *p1*, *p2* et *p3* trois parsers. Le parser *p3* est obtenu par combinaison des parsers *p1* et *p2*. Si *p3* est une séquence des parsers *p1* et *p2* alors *p3* est vérifié si une première partie de l'entrée vérifie *p1* puis la suite de l'entrée vérifie *p2*. Si *p3* est un choix entre *p1* et *p2* alors *p3* est vérifié si *p1* est vérifié mais pas *p2* ou si *p2* est vérifié mais pas *p1*. Les deux parsers sont testés. Si *p3* est un choix ordonné entre *p1* et *p2* alors *p3* est vérifié si *p1* est vérifié ou si *p2* est vérifié mais pas *p1*. Dans le cas d'un choix ordonné, si *p1* est vérifié alors *p2* n'est pas testé. Les parsers combinateurs permettent de se concentrer sur les détails des parsers les plus spécifiques pour obtenir les parsers généraux par composition.

Cette caractéristique est illustrée par l'exemple suivant où des parsers combinateurs sont utilisés pour écrire un parser générique de modèles SysML simplifiés. La Figure 3.27 reprend le diagramme de bloc interne utilisé comme exemple dans la section 3.3. L'exemple se limite ainsi à traiter le cas des diagrammes de blocs internes.

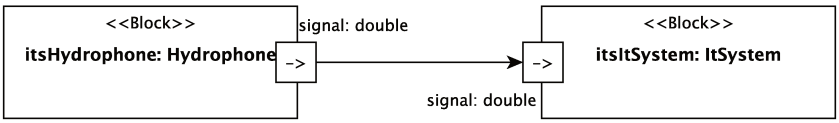


FIGURE 3.27. – Exemple simple de diagramme de bloc interne SysML.

Une forme sérialisée du modèle définie par ce diagramme est analysée en utilisant des parsers combinateurs. Le modèle d'entrée du parser contient des éléments de types *SysmlPart*, *SysmlFlowPort* et *SysmlConnection*. Le modèle a donc la forme d'un multiensemble tel que défini par l'équation 3.1.

$$\text{SysmlModel} = \{(\text{SysmlPart}, i), (\text{SysmlFlowPort}, j), (\text{SysmlConnection}, k)\} \text{ avec } i, j, k \in \mathbb{N} \tag{3.1}$$

Les termes i, j et k sont les multiplicités des occurrences de chaque terme dans le multiensemble. La forme sérialisée de ce modèle contiendra donc de multiples occurrences des formes sérialisées de *SysmlPart*, *SysmlFlowPort* et de *SysmlConnection*. Un parser est défini pour chacune de ces formes sérialisée conformément aux définitions suivantes :

$$\text{partParser} : \text{SerializedPart} \rightarrow \text{SysmlPart} \tag{3.2}$$

$$\text{flowPortParser} : \text{SerializedFlowPort} \rightarrow \text{SysmlFlowPort} \tag{3.3}$$

$$\text{connectionParser} : \text{SerializedConnection} \rightarrow \text{SysmlConnection} \tag{3.4}$$

partParser est un parser défini pour être vérifié lorsque son entrée est une forme sérialisée d'une *part* SysML. Lorsque ce parser est vérifié, il produit l'instance de *SysmlPart* qui correspond à la forme sérialisée. La structure de la forme sérialisée du modèle SysML en entrée implique que son parser peut être obtenu en composant les parsers *partParser*, *flowPortParser* et *connectionParser*.

Dans le format sérialisé des modèles SysML, les éléments SysML peuvent être sérialisés dans n'importe quel ordre. De plus, la forme sérialisée d'un élément de modèles ne valide pas les parsers associés aux types des éléments de modèles qui ne lui correspondent pas. De ce fait, nous pouvons utiliser l'opérateur de choix ordonné pour combiner les différents parsers.

En notant, / l'opérateur de choix ordonné et * l'opérateur d'occurrences multiples, le parser pour le modèle SysML sérialisé pour l'exemple est donné par l'équation 3.5.

$$sysmlParser = (partParser/flowPortParser/connectionParser)* \tag{3.5}$$

Cette écriture du parser pour le modèle SysML a permis de se concentrer sur l'écriture des parsers des sous-parties *SysmlPart*, *SysmlFlowPort* et *SysmlConnection* de la forme sérialisée de SysML. Les parsers peuvent être réutilisés dans une autre application. De plus, en cas de modification de la forme sérialisée d'une sous-partie, l'écriture du parser global n'est pas affectée. Seul le parser de la sous-partie doit être modifié. Ainsi, si la forme sérialisée de la sous-partie *SysmlPart* est modifiée, seul le parser *partParser* doit être modifié. Les parsers *flowPortParser*, *connectionParser* mais surtout *sysmlParser* n'ont pas à être redéfinis.

Cet exemple a permis d'illustrer l'utilisation des opérateurs de combinaison de parsers pour écrire un parser d'une forme sérialisée d'un modèle. L'utilisation des opérateurs de combinaison a permis d'isoler les sources de changement dans le format sérialisé du modèle et facilite la maintenance et l'évolution du parser. Ces propriétés des parsers combinateurs peuvent être utilisées pour parser des modèles réalisés dans plusieurs langages de modélisation et différents formats de sérialisation.

3.5.2. Modularité apportée par les parsers combinateurs

Les outils de modélisation et les formats de sérialisation des modèles vont évoluer au cours du temps. Pour assurer la pérennité de notre approche, il n'est pas possible de la reposer sur une technologie de modélisation ou de sérialisation particulière. De plus, l'acceptation de notre approche dépend de la capacité à travailler avec les outils de modélisation déjà existants en tenant compte de leur diversité.

L'utilisation des parsers combinateurs permet d'obtenir une indépendance par rapport aux outils de modélisation et leurs formats de sérialisation. En effet, l'écriture d'un parser se fait pour traiter un format de sérialisation particulier en tenant compte des spécificités inclues par l'outil de modélisation qui a généré le modèle sérialisé. Cependant, le parser est technologiquement indépendant de l'outil de modélisation et du format de sérialisation. Il continuera à pouvoir être utilisé même si l'outil de modélisation n'est plus utilisé. Il est ainsi possible de pouvoir continuer à manipuler des modèles anciens.

Les parsers combinateurs possèdent plusieurs propriétés permettant de gérer l'hétérogénéité et l'évolution des langages de modélisation :

— facilité de maintenance des parsers pour suivre les évolutions des langages,
— définition de parsers spécifiques qui peuvent être combinés pour gérer l'hétérogénéité,

— la combinaison de parsers sans avoir besoin de la concevoir a priori pour gérer l'ajout de nouveaux langages.

Les parsers combinateurs permettent de concevoir de manière modulaire des parsers. La maintenance des parsers est donc facilitée. Les modifications à apporter en cas de changement sont donc localisées au(x) parser(s) concerné(s). De plus, les parsers combinateurs combinent différents parsers. Il est donc aisé d'ajouter un nouveau parser pour gérer un nouveau langage ou un nouveau format de sérialisation en le combinant avec des parsers existants. La Figure 3.28 indique quels parsers sont impactés par les évolutions précédemment citées.

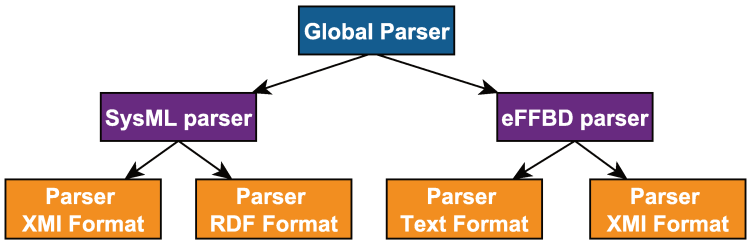


FIGURE 3.28. – Identification des parsers impactés lors de modification: en bleu ceux impactés par l'ajout d'un nouveau langage de modélisation, en violet ceux impactés par l'ajout d'un nouveau format de sérialisation, en orange ceux impactés par une évolution d'un langage de modélisation.

L'import des modèles doit pouvoir être réalisé sans devoir spécifier le format de sérialisation des modèles qui est d'ailleurs souvent inconnu des modeleurs. Pour importer les modèles dans les différents langages de modélisation, un parser global est donc défini. Ce parser est une combinaison des parsers spécifiques à chaque langage de modélisation. L'ajout d'un langage de modélisation se traduit par la définition d'un parser spécifique pour le nouveau langage et sa combinaison avec les parsers existants au sein du parser global.

Les parsers des formats de sérialisation sont combinés pour obtenir les parsers des langages de modélisation. L'ajout d'un nouveau format de sérialisation pour un langage de modélisation donné se traduit par l'écriture d'un parser pour ce nouveau format. Le parser du langage de modélisation est donc modifié lors de l'ajout d'un nouveau format.

Chaque concept d'un langage de modélisation est sérialisé d'une manière spécifique. Un parser par concept sérialisé est défini. Tous les parsers spécifiques aux concepts sont combinés à l'aide d'opérateurs de combinaison pour obtenir le parser pour le langage de modélisation sérialisé dans un format donné. Ainsi, si la définition du langage de modélisation ou d'une partie du format de sérialisation change alors le changement dans le parser est localisé et n'implique pas la redéfinition du parser complet.

La modularité apportée par les parsers combinateurs permet de suivre efficacement les évolutions des langages de modélisation. Il reste à rendre leurs définitions utilisables au sein de l'environnement Role4All.

3.5.3. Intégration des parsers combinateurs

Une fois les parsers combinateurs définis, il reste à les intégrer dans notre environnement de manipulation de modèles hétérogènes. Pour faciliter cette intégration, le mécanisme d'introspection est utilisé. L'introspection est la capacité d'un programme à connaître son état.

Parmi les utilisations de l'introspection, nous pouvons citer la détermination de la classe d'une instance ou des méthodes et des attributs d'une classe.

Utiliser l'introspection permet de séparer l'écriture des parsers de la définition de l'environnement de manipulation des modèles. La nature des modèles à utiliser est très variable dans le temps alors que la définition de l'outil de manipulation des modèles doit rester stable. L'introspection permet de tenir compte de la variabilité des modèles en permettant de charger dynamiquement les parsers associés aux langages de modélisation désirés. En poussant plus loin cette caractéristique et en utilisant les opérateurs de combinaison de parsers, il est possible de définir une bibliothèque de parsers. Seuls les parsers nécessaires à l'import des modèles désirés ont à être chargés dans l'environnement. L'introspection permet de les retrouver et de les utiliser sans autre déclaration que leur chargement dans l'environnement. L'utilisation des parsers est donc simplifiée par l'utilisation de l'introspection.

L'architecture de la solution d'import est illustrée Figure 3.29.

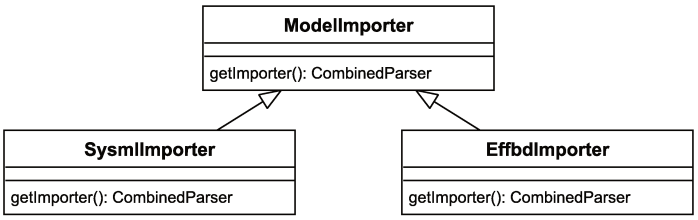


FIGURE 3.29. – Architecture pour l'utilisation de l'introspection pour faciliter l'intégration d'importateurs afin d'importer des modèles.

Un importateur est associé à tous les langages de modélisation. Cet importateur est bâti autour d'un parser obtenu en utilisant des parsers combinateurs. Tous les importateurs sont définis par une classe qui hérite de la classe ModelImporter. Cette dernière définit une méthode de classe abstraite *getImporter*. Dans les sous-classes, cette méthode est surchargée dans les sous-classes et renvoie le parser lié à l'importateur qui est associé à un langage de modélisation. L'introspection est utilisée pour trouver toutes les sous-classes de ModelImporter et appeler la méthode getImporter sur celles-ci. La méthode getImporter renvoie un parser. Les retours des différents appels à getImporter peuvent eux-mêmes être combinés par des parsers combinateurs. Un parser pour tous les langages de modélisation et leurs différents formats de sérialisation est ainsi obtenu.

Pour importer un modèle dans l'environnement Role4All, il est nécessaire de définir les parsers pour les différents formats de sérialisation désirés pour lire les modèles. Ces parsers sont combinés pour obtenir un parser pour le langage de modélisation. Ce parser doit ensuite être encapsulé dans une classe héritant de la classe ModelImporter fournie par l'environnement. Une fois cela réalisé, l'environnement est capable de travailler avec des modèles réalisés dans le langage de modélisation désiré et sérialisés dans les formats précisés par les parsers.

L'utilisation des parsers combinateurs associés à l'introspection permet d'adapter l'environnement Role4All aux évolutions des langages de modélisation et de leurs formats de sérialisation.

3.6. Conclusion

Les approches basées modèles apportent une aide à la communication et à la compréhension aux concepteurs de systèmes et de logiciels. Cependant, la complexité des systèmes et des logiciels augmente. La conception d'un système ou d'un système de systèmes nécessite d'utiliser un ensemble de langages de modélisation. Le passage de modèles système à des modèles de simulation doit tenir compte de cette hétérogénéité. Il faut également tenir compte de l'hétérogénéité des modèles de simulation. De plus, la longue durée de vie des systèmes implique une évolution de ces systèmes ainsi que des langages et outils de modélisation qui ont servi à leur conception. Le même constat peut être fait pour ce qui est des langages et outils de simulation. En effet, au cours de la conception d'un système, des simulations de différentes natures sont utilisées pour vérifier la conception du système. De plus, au cours du temps, les outils de simulation évoluent et de nouveaux outils peuvent être utilisés. Ainsi, pour permettre de créer des modèles de simulation à partir de modèles système, il faut à la fois gérer l'hétérogénéité et l'évolution des langages et outils de modélisation système et gérer l'hétérogénéité et l'évolution des langages et outils de simulation.

Le concept de rôle permet de définir un point de vue sur des objets indépendamment de leur type. Le concept de rôle est donc adapté à notre contexte de manipulation de modèles hétérogènes. Il possède également différentes propriétés. Nous avons sélectionné celles qui sont adaptées à notre contexte.

Un rôle possède des propriétés propres, il est donc possible d'ajouter de l'information aux modèles en fonction du point de vue défini par les rôles. Les rôles peuvent partager des aspects structurels et comportementaux ce qui permet de réutiliser des définitions de rôles. Les rôles peuvent être groupés pour définir des points de vue plus abstraits sur des éléments de modèles.

Des éléments de modèles différents définis dans plusieurs modèles peuvent jouer le même rôle. Ainsi, des éléments différents peuvent être interprétés de la même manière. Un même élément de modèle peut jouer des rôles différents. Ainsi, un même modèle peut être analysé ou simulé de différentes manières. Un même élément de modèle peut jouer plusieurs fois le même rôle. Cet élément de modèle pourra ainsi être interprété plusieurs fois. Les rôles possèdent leur propre identité, ce qui fait que l'interprétation est propre à chaque rôle. Les rôles restreignent l'accès aux objets en définissant des points d'accès aux propriétés des éléments de modèles. La création d'associations entre les éléments de modèle et des rôles est dynamique. Il est donc possible de changer le point de vue associé à des éléments de modèles. De plus, les associations entre les rôles et les éléments de modèles peuvent être contraintes ce qui permet de sélectionner les éléments de modèles d'intérêt par rapport au point de vue défini par les rôles. Les rôles peuvent jouer des rôles. Cette caractéristique donne la capacité de définir des points de vue sur des points de vue et donc de réutiliser des données extraites des modèles.

Les rôles dépendent d'un contexte qui est l'objectif pour lequel ils sont utilisés. Un même rôle peut donc faire partie de plusieurs contextes. Chaque contexte possède sa propre identité.

Le choix de ces différentes caractéristiques a permis l'élaboration d'un méta-modèle. Ce méta-modèle permet de définir des rôles et de les associer avec des éléments de modèles. Il est donc possible de manipuler des modèles système différents. L'élaboration de ce méta-modèle permet de ne pas modifier les langages et les outils de modélisation système.

Cependant, il est nécessaire d'être capable de lire les modèles système avant de pouvoir leur associer des rôles. Pour ce faire, nous avons défini des importateurs modulaires à l'aide de parsers combinateurs. Ces derniers permettent de faciliter la maintenance des parsers en écrivant des parsers spécifiques qui sont combinés. De plus, il est possible de combiner des

parsers à n'importe quel moment dans le temps et donc de prendre en compte l'utilisation de nouveaux langages de modélisation.

Ce chapitre a permis de poser les bases conceptuelles, les rôles et les parsers combinateurs pour des importateurs modulaires, de l'environnement Role4All facilitant l'analyse et la simulation de modèles hétérogènes. Le chapitre suivant décrit les détails d'implémentation de Role4All.

4

Réalisation de l'environnement Role4All

Sommaire

4.1. Introduction à la mise en œuvre 83
4.2. Analyse 84
4.2.1. Analyse fonctionnelle 86
4.2.2. Analyse physique 88
4.3. Définition à la volée de rôles 89
4.3.1. Description du processus de définition et d'allocation des rôles 90
4.3.2. Implémentation de la création des types de rôles 91
4.3.3. Implémentation de l'allocation des rôles 93
4.4. Définition à la volée des adaptateurs 96
4.4.1. Processus de définition des adaptateurs 96
4.4.2. Implémentation des adaptateurs dynamiques 97
4.5. Rôles et contextes 100
4.5.1. Définition d'un contexte 100
4.5.2. Mise en place du contexte 101
4.5.3. Sérialisation d'un contexte 102
4.6. Implémentation d'importateurs modulaires avec des parsers combinateurs 103
4.6.1. Processus de définition d'importateurs modulaires à base de parsers combinateurs 103
4.6.2. Le framework PetitParser 104
4.6.3. Implémentation des importateurs 107
4.7. Réalisation d'une interface dédiée 108
4.8. Conclusion 111

4.1. Introduction à la mise en œuvre

La conception de systèmes et de systèmes de systèmes implique l'utilisation de plusieurs langages de modélisation et de plusieurs outils de simulation. De plus, les systèmes conçus ont une longue durée de vie. Par conséquent, les langages de modélisation et les outils de simulation vont évoluer.

Nous avons identifié le concept de rôle comme un concept permettant de gérer l'hétérogénéité des modèles système et de simulation. L'utilisation d'importateurs modulaires permet de s'affranchir des spécificités techniques des outils de modélisation et leur obsolescence potentielle.

Les rôles et les importateurs modulaires doivent encore être mis en œuvre dans un outillage informatique dédié. La Figure 4.1 présente les objectifs de la mise en œuvre informatique de l'environnement.

D'après la Figure 4.1a, l'environnement Role4All doit permettre de définir des modèles de rôles et des importateurs modulaires utilisant des parsers combinateurs. Afin de prendre en compte de nouveaux outils de simulation, la mise en œuvre informatique de l'environnement Role4All doit permettre de définir des modèles de rôles complexes où les rôles peuvent eux-mêmes jouer des rôles (Figure 4.1b). Afin de prendre en compte l'utilisation de nouveaux langages de modélisation, les modèles de rôles doivent pouvoir être étendus par de nouveaux rôles sans modifier la structure des modèles de rôles déjà existants. De même, les importateurs modulaires doivent pouvoir être étendus ou modifiés pour tenir compte des évolutions des langages de modélisation et de l'utilisation de nouveaux langages (Figure 4.1c).

Les objectifs de ce chapitre sont :

— fournir une analyse des services que la mise en œuvre informatique de Role4All doit rendre,
— présenter les processus d'utilisation de chacun des services,
— présenter la réalisation de chacun des services.

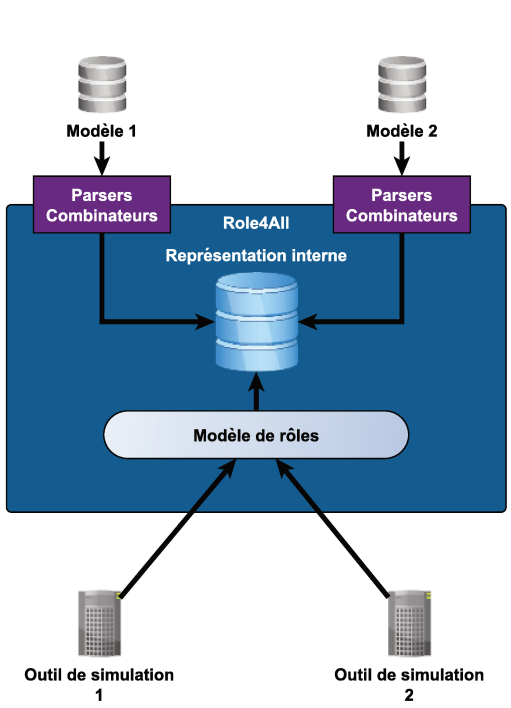
La section 4.2 fournit une analyse des besoins en terme de fonctionnalités de l'outillage et de son implémentation. La section 4.3 décrit le processus pour définir des rôles et les associer à des éléments de modèles ainsi que la réalisation informatique du processus. La section 4.4 expose le processus pour définir l'adaptation des éléments de modèles par les rôles et sa réalisation informatique. La section 4.5 décrit le processus de définition d'un contexte d'utilisation des rôles et sa réalisation informatique[1]. La section 4.6 décrit la création des importateurs modulaires à l'aide de combinateurs de parsers[2]. La section 4.7 décrit l'interface graphique réalisée pour l'environnement Role4All.

4.2. Analyse

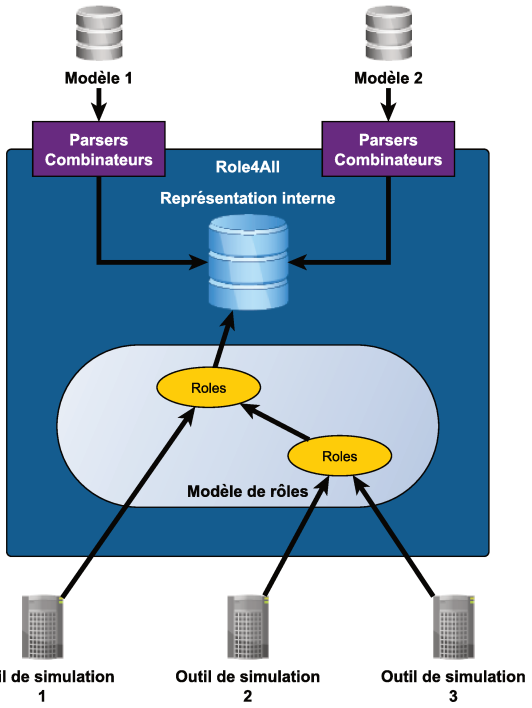
Le chapitre 3 a présenté les rôles et les importateurs modulaires qui sont les concepts qui soutiennent notre approche. Nous avons implémenté ces éléments au sein d'un environnement dédié que nous avons nommé Role4All. L'implémentation a été guidée par l'analyse des services qui doivent être rendus par l'environnement Role4All. Cette analyse explicite chaque partie prenante de l'environnement et les services à rendre à chacune des parties prenantes. Conformément à l'approche ingénierie système, nous avons décomposé l'analyse en deux parties :

1. l'analyse fonctionnelle pour déterminer les fonctions à réaliser pour chaque partie prenante de l'environnement Role4All (Sous-section 4.2.1),
2. l'analyse physique pour déterminer les aspects technologiques de l'implémentation de chaque fonction (Sous-section 4.2.2).

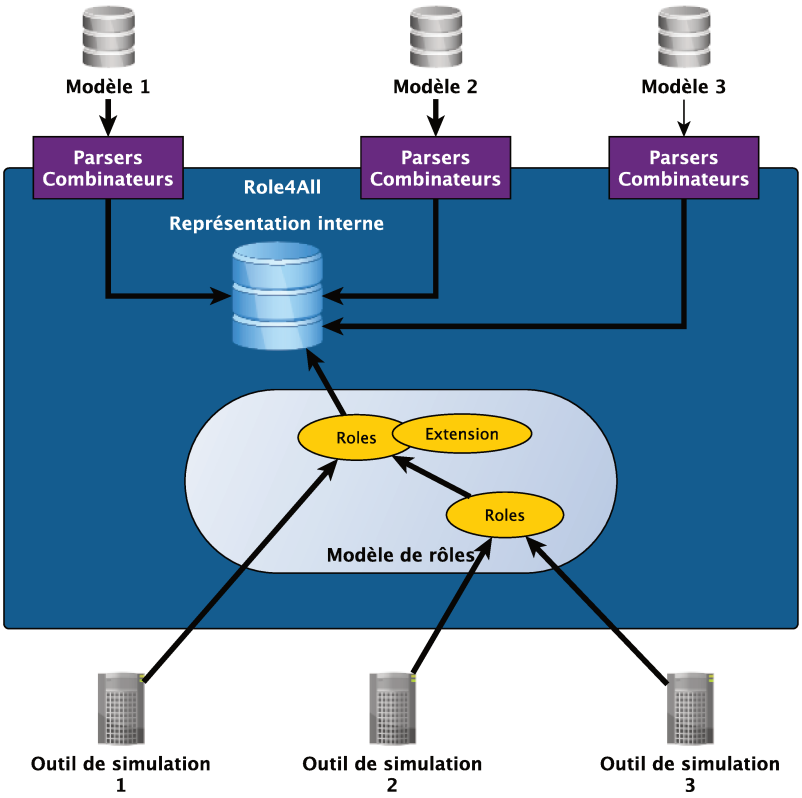
1. Ces deux sections ont fait l'objet de publication dans les conférences Syscon 2015 [80] et WETICE 2015 [79]
2. L'utilisation des parsers combinateurs a été présentée au workshop Software Engineering for Systems of Systems 2014 [83] et à la conférence WETICE 2015 [79]



(a) Architecture globale de l'environnement Role4All.



(b) Gestion de l'ajout d'un nouvel outil de simulation.



(c) Gestion de l'ajout d'un nouveau langage de modélisation.

FIGURE 4.1. – Objectifs de la réalisation informatique de l'environnement Role4All.

4.2.1. Analyse fonctionnelle

Les rôles permettent de définir une interface entre des outils de simulation et des modèles hétérogènes et évolutifs. Cette interface est un point de vue sur les modèles système à la manière d'un modèle pivot. De plus, les rôles permettent aussi de créer des liens sémantiques entre des éléments de modèles. Cependant, le modèle de rôle peut évoluer pour s'adapter aux évolutions des modèles système.

L'utilisation de rôles pour exécuter des modèles hétérogènes dans différents environnements d'exécution implique de distinguer quatre acteurs :

Utilisateur final Acteur en charge de la définition du besoin et de la réalisation de l'analyse et de l'exécution de modèles.

Modeleur des rôles Acteur en charge de la définition des rôles et des adaptateurs.

Modèles systèmes Les modèles des systèmes à analyser et/ou à exécuter en tant que partie de l'environnement de l'outil.

Modèles de simulation Les modèles utilisés par les outils d'analyse et d'exécution en tant que partie de l'environnement de l'outil.

L'activité de définition du besoin consiste à sélectionner les modèles à utiliser en entrée en fonction du niveau d'abstraction souhaité. Les outils d'analyse et d'exécution doivent également être choisis en fonction des objectifs à atteindre.

L'activité de définition des rôles consiste à définir les rôles qui seront utilisés, leurs associations avec des éléments de modèles ou d'autres rôles et les adaptateurs qui serviront à la génération des modèles d'analyse et d'exécution.

La Figure 4.2 identifie les acteurs liés à l'outillage d'exécution de modèles hétérogènes basés rôles ainsi que les fonctions principales fournies par l'outillage et les fonctions contraintes posées par les modèles.

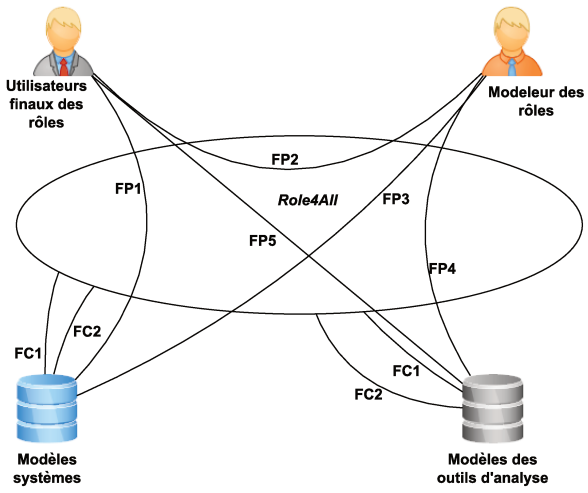


FIGURE 4.2. – Identification des acteurs et des fonctions d'un outillage basé rôle pour l'exécution de modèles hétérogènes

Les fonctions principales sont :

FP1 Importer des modèles hétérogènes.

FP2 Créer des rôles spécifiques.

FP3 Définir des associations entre rôles et éléments de modèles.

FP4 Définir l'adaptation des rôles.

FP5 Générer la simulation.

La fonction FP1 s'adresse aux utilisateurs finaux et leur permet d'importer dans l'outillage les modèles des systèmes à analyser et exécuter.

La fonction FP2 s'adresse aux modeleurs de rôles qui doivent définir un rôle en tant qu'entité par un nom / type, des attributs et des comportements. Ces rôles doivent être définis selon les besoins des utilisateurs finaux.

La fonction FP3 consiste à créer le modèle de rôles en définissant les rôles qui sont joués par les éléments de modèles et d'autres rôles.

La fonction FP4 consiste à définir les adaptateurs entre rôles et objets. Ces adaptateurs sont définis en fonction des outils visés pour l'analyse et l'exécution des modèles.

La fonction FP5 donne la possibilité aux utilisateurs finaux de créer les modèles d'analyse et d'exécution à partir des modèles importés et des modèles de rôles définis par les concepteurs de rôles.

Les fonctions contraintes, qui correspondent à des fonctions qui permettent au système de répondre à des contraintes de l'environnement, sont :

FC1 Gérer la diversité des modèles.

FC2 Gérer les évolutions des modèles.

Ces deux fonctions contraintes concernent aussi bien les modèles hétérogènes importés que les modèles de simulation des outils d'analyse et d'exécution. La fonction FC1 implique que l'outillage doit être capable d'importer des modèles système dans des formats et des langages divers. Il doit également être capable de s'adapter à plusieurs outils d'analyse et d'exécution.

La fonction FC2 implique que l'outillage doit être capable de s'adapter aux changements des formalismes de modélisation système et des outils utilisés pour analyser et exécuter les modèles.

La manipulation de modèles hétérogènes peut se faire par des approches par unification ou par fédération. L'unification utilise un formalisme de modélisation qui contient les concepts communs à différents modèles d'entrées pour créer un modèle pivot à partir des modèles d'entrée. La fédération crée des liens sémantiques entre des éléments de modèles différents.

La définition de modèles de rôles par les modeleurs de rôles permet de mettre en relation des éléments de modèles exprimés dans des langages différents à l'image de la fédération de modèles. Le modèle de rôle définit un point de vue sur les modèles système et permet aux utilisateurs finaux de créer des simulations à l'image de l'unification.

Cependant, en cas de changement dans les langages de modélisation ou les outils de simulation, les approches par unification et par fédération nécessitent de reprendre la définition du modèle pivot ou du modèle fédéré. Contrairement à l'unification et la fédération, les rôles et les modèles de rôles peuvent être adaptés aux changements des modèles système et des outils de simulation à la volée.

Les technologies choisies pour la réalisation de l'environnement Role4All doivent permettre de mettre au point les modèles de rôles et d'avoir la capacité de faire évoluer ces modèles à la volée.

4.2.2. Analyse physique

L'utilisation des rôles permet de répondre au besoin de s'adapter aux changements des langages de modélisation et des outils de modélisation. Les rôles possèdent des caractéristiques qui permettent de réaliser cette adaptation à la volée en étant associé à des objets et dissocié dynamiquement. L'environnement utilisé pour la modélisation et l'utilisation des rôles doit supporter l'association et la dissociation dynamique de rôles.

Les évolutions des langages de modélisation et des outils de simulation impose également d'être capable de définir de nouveaux rôles et de modifier des rôles existants. Les évolutions des langages de modélisation et des outils de simulation ne peuvent pas être prédites. Ainsi, l'environnement doit permettre de modifier à la volée la définition des rôles pour les adapter aux évolutions des langages de modélisation et de simulation. Cette propriété de l'environnement de modélisation et d'utilisation des rôles facilite également le processus de prototypage rapide. En effet, en ayant la capacité de modifier à la volée des rôles, il est possible de créer "en direct" des alternatives d'architectures à simuler.

De plus, nous nous imposons de réduire le nombre d'outils à utiliser pour permettre de définir des rôles et les associer aux éléments de modèles.

Le choix d'un langage de programmation pour la réalisation informatique de l'environnement Role4All doit ainsi répondre aux besoins suivants :

— associer et dissocier des rôles à la volée,
— modifier la définition de rôles à la volée,
— intégrer au maximum les outils nécessaires à la modélisation et l'utilisation des rôles et des modèles système au sein de Role4All.

L'association et la dissociation de rôles à la volée peut être réalisée dans n'importe quel langage de programmation à partir du moment où les objets et les rôles sont stockés dans un espace mémoire accessible à l'utilisateur. Ce dernier doit disposer d'outils lui permettant d'accéder aux objets en mémoire, de créer des rôles, d'associer ces rôles et de les dissocier.

La capacité de modifier la définition des rôles à la volée implique la possibilité de redéfinir leurs propriétés au sein d'un programme en cours d'exécution. Pour répondre à ce comportement dynamique, les langages à typage dynamique sont considérés comme une solution en fournissant des outils comme le chargement dynamique ou l'interception de comportement à l'exécution [62].

Smalltalk est un langage de programmation orienté objets à typage dynamique. Contrairement à d'autres langages de programmation à typage dynamique, Smalltalk dispose d'un environnement de débogage puissant. En cas d'erreur à l'exécution d'un programme, le débogueur Smalltalk se lance automatiquement. Ainsi, en cas d'évolutions d'un langage de modélisation ou d'un outil de simulation, les propriétés manquantes ou obsolètes des rôles vont provoquer une erreur et donc le lancement du débogueur Smalltalk. Ce dernier permet également de modifier le code des classes Smalltalk à la volée en ajoutant de nouvelles méthodes ou en modifiant la définition (paramètres ou code) de méthodes. Une fois les corrections effectuées, l'exécution du programme peut être reprise sans nécessiter le redémarrage du programme. Le débogueur Smalltalk facilite la mise au point d'un modèle de rôles et permet de s'adapter aux évolutions des langages de modélisation et des outils de simulation en permettant de corriger les erreurs dues aux évolutions.

Le débogueur Smalltalk intègre un inspecteur d'objets. Un inspecteur d'objets est un outil qui permet d'accéder aux propriétés d'un objet et de pouvoir exécuter du code sur cet objet. L'inspecteur permet de prototyper des actions sur un objet décrites en Smalltalk. Il facilite

donc la mise au point d'un programme Smalltalk. L'utilisation de l'inspecteur facilite la mise au point d'un modèle de rôle en permettant de prototyper les interactions entre les rôles et les objets avant leur implémentation.

Enfin, restreindre le nombre d'outils à utiliser pour modéliser et utiliser des rôles ainsi que des modèles système hétérogènes implique de définir le plus d'outils possible dans le même langage de programmation. En effet, le nombre de langage est réduit facilitant l'apprentissage de l'outillage et donc son acceptation. Pour importer les modèles système, nous avons choisi d'utiliser des parsers combinateurs. Smalltalk possède un framework pour l'écriture de parsers combinateurs [75]. L'écriture des importateurs modulaires peut donc être réalisée directement en Smalltalk.

Nous avons également défini un méta-modèle qui définit un langage spécifique (DSL [3]) pour la modélisation de rôles. Le besoin de limiter le nombre de langages de programmation utilisés, nous a fait choisir l'implémentation du DSL sous la forme d'un DSL interne. D'après [28], il existe deux types de DSL, les DSLs externes et les DSLs internes. Les DSLs externes doivent être analysé par un programme écrit dans un autre langage de programmation, ce dernier réalisant les actions décrites par le DSL. Les DSLs internes sont des APIs définies directement dans le langage de programmation qui réalisera les actions. Un DSL interne doit permettre de s'abstraire des détails techniques d'une implémentation dans le langage de programmation hôte du DSL ainsi que d'absorber les enchaînements de comportements dans le langage hôte [34]. Ces propriétés facilitent l'adoption du DSL par les utilisateurs. Le DSL interne fournit une interface métier pour les utilisateurs qui masquent la complexité du langage de programmation sous-jacent tout en s'intégrant avec les environnement de développement déjà existants.

Contrairement à des langages comme Java, Smalltalk possède la notion d'image. Une image Smalltalk est un instantané de l'état de la mémoire associée à Smalltalk. Elle contient l'ensemble des classes définies en Smalltalk ainsi que leur code et tous les objets instanciés. Grâce à la notion d'image, il est possible d'étendre directement le langage Smalltalk en ajoutant les concepts nécessaires à l'implémentation de notre DSL de rôles. De plus, les modifications apportées sont conservées entre deux redémarrages de l'environnement Smalltalk. L'ouverture de l'environnement Smalltalk fournie par l'image permet d'adapter cet environnement à des besoins spécifiques en l'étendant tout en étant capable d'utiliser tous les services fournis par l'environnement. Un DSL interne peut donc être créer en ajoutant les concepts nécessaires directement à l'image Smalltalk.

L'analyse fonctionnelle a permis d'identifier les utilisateurs finaux, les modeleurs de rôles, les modèles système et les outils d'analyse de modèles comme les parties prenantes de l'environnement Role4All. Role4All doit ainsi fournir des fonctions pour permettre de définir des rôles, de définir les adaptateurs entre les rôles et les éléments de modèles, d'utiliser les rôles mais aussi d'importer les modèles. Ces différentes fonctions ont été implémentées en Smalltalk.

4.3. Définition à la volée de rôles

Définir à la volée des rôles permet aux modeleurs de rôles de développer les modèles de rôles de manière itérative. Cela facilite la mise au point des modèles de rôles. De plus, l'utilisation dans un environnement de prototypage rapide est facilitée car la définition à la volée de rôles permet d'ajouter des rôles qui n'ont pas été identifiés lors de travaux préliminaires et de pouvoir

3. Domain Specific Language.

— créer un nouveau type de rôle,
— ajouter des attributs à une classe qui définit un type de rôle,
— retrouver toutes les instances d'une classe,
— retrouver une instance spécifique d'une classe,
— associer un rôle à une instance spécifique,
— associer un rôle à toutes les instances d'une classe sous une condition donnée.

4.3.2. Implémentation de la création des types de rôles

Le méta-modèle de rôle décrit dans la Section 3.4 et reproduit Figure 4.4 a été implémenté en Smalltalk. Les méta-classes définies dans le méta-modèle ont été réalisées sous la forme de classes Smalltalk qui forment un DSL interne que nous avons nommé R2DL (Role4All Role Description Language).

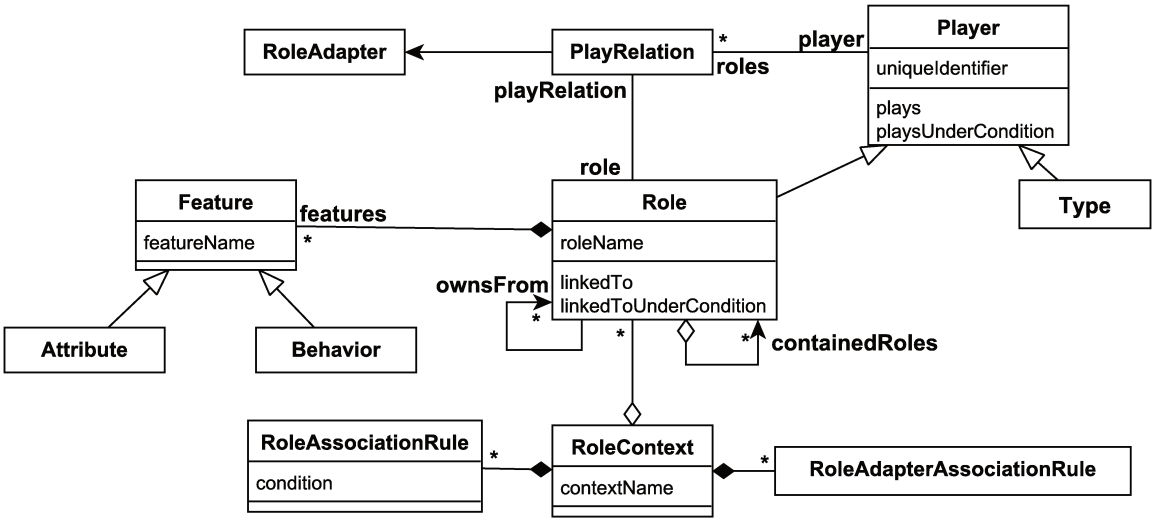


FIGURE 4.4. – Méta-modèle complet du langage de rôle de l'environnement Role4All

Un rôle possède des propriétés structurelles et comportementales. Il peut partager tout ou partie de ses propriétés avec d'autres rôles. Deux implémentations différentes peuvent être utilisées : soit la classe Role possède des attributs pour définir les propriétés structurelles et comportementales d'un rôle soit des types de rôles sont définis sous la forme de classes.

La première solution permet de ne définir qu'une seule classe pour la notion de rôle. Cependant, elle nécessite de s'assurer manuellement de la cohérence entre des propriétés partagées par différents rôles ainsi que celle des propriétés de rôles identiques.

Définir des types de rôles en utilisant des classes permet d'utiliser les mécanismes natifs de la programmation orientée objet. L'héritage entre les classes des types de rôles permet à leurs instances de partager la définition de leurs attributs et de leurs méthodes. De plus, la définition des types de rôles par des classes permet leur réutilisation par des équipes différentes.

L'utilisation de mécanismes natifs du langage Smalltalk et la capacité de réutilisation, nous ont fait choisir la définition de types de rôles sous la forme de classes.

Nous avons donc choisi d'utiliser une approche orientée objets pour la définition du DSL interne R2DL. Par conséquent, toutes les instructions du DSL R2DL sont des méthodes des

classes Smalltalk qui représentent les méta-classes de la définition du DSL.

Les types de rôles sont des définitions spécialisées de rôles. La classe Role est, quant à elle, une définition abstraite de rôles. Il existe donc une relation d'héritage entre la classe Role et les classes des types de rôles comme illustré Figure 4.5.

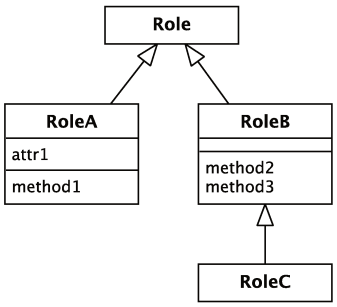


FIGURE 4.5. – Héritage entre classe de types de rôle et la classe Role

Les instances des classes des types de rôles sont donc des instances de la classe Role grâce à la relation d'héritage entre classes de types de rôle et la classe Role. Cette dernière représente la méta-classe Role de notre méta-modèle. Par conséquent, les instances des classes de types de rôles sont des rôles d'après notre définition du concept de rôle (Définition 17).

Les types de rôles définis par les modeleurs de rôles sont créés sous la forme de sous-classes de la classe Role. Le langage Smalltalk fournit des méthodes pour créer dynamiquement des classes. Notre DSL offre une interface de programmation dédiée à la création des types de rôles en appelant les méthodes fournies par le langages Smalltalk. La création d'un type de rôle se fait par l'intermédiaire du code présenté dans le Listing 4.1.

```
1 Role class >>named: aRoleName classifiedIn: aCategoryName
2   "Creates a role class with the name aRoleName and adds it to the category aCategoryName"
3   Role subclass: aRoleName asSymbol
4   instanceVariableNames: ''
5   classVariableNames: ''
6   poolDictionaries: ''
7   category: aCategoryName .
8 Role class >> named: aRoleName
9   self named: aRoleName classifiedIn: 'Role-Types-Definitions'.
```

Listing 4.1 – Code de création d'un type de rôle

La ligne 1 définit la méthode du DSL qui permet la création d'un type de rôle en fournissant son nom (*aRoleName*) et le nom d'un paquet logique (appelé catégorie en Smalltalk) où la classe du rôle sera classée (*aCategoryName*). La ligne 2 crée une sous-classe de la classe Role. Dans le dialecte de Smalltalk utilisé (Pharo), deux classes ne peuvent pas avoir le même nom. Le nom soumis par le modeleur de rôle est transformé en symbole pour la création de la classe. L'utilisation d'un symbole assure l'unicité de la classe à créer. L'ajout de la classe créée à la catégorie spécifiée est réalisé ligne 7. Utiliser des catégories permet d'organiser les éléments crées en utilisant R2DL pour faciliter la réutilisation du travail réalisé lors de la mise au point du modèle de rôles. En effet, les classes créées pour répondre à un besoin des utilisateurs finaux sont regroupées par familles de concepts et il est possible de les réutiliser ultérieurement.

La ligne 8 définit la méthode *named:* de la classe Role de R2DL. Cette méthode permet

de créer un type de rôle du nom *aRoleName* et de classer la classe qui définit le type de rôle par défaut dans la catégorie *Role-Types-Definitions*. Cette méthode réutilise la méthode *named:classifiedIn:* précédemment définie.

Le type d'un rôle décrit également les attributs que possède les rôles du type défini. R2DL encapsule les appels aux méthodes Smalltalk pour réaliser ces opérations comme illustré dans le Listing 4.2.

```
1 Role class >> addAttributeNamed: anAttributeName
2   "Adds an attribute with name anAttributeName to the role class and adds the accessor methods."
3   self addInstVarNamed: anAttributeName .
4   self createGetter: anAttributeName .
5   self createSetter: anAttributeName .
```

Listing 4.2 – Ajout d'un attribut à un type de rôle

La méthode *addAttributeNamed:* a été ajoutée à la classe Role. Cette méthode prend le nom de l'attribut à ajouter en paramètre sous la forme d'une chaîne de caractères. La ligne 3 illustre la création d'un attribut d'instance. Le mot-clé *self* fait référence à la classe sur laquelle la méthode *addAttributeNamed:* est appelée et non pas à la classe Role. L'attribut est ainsi ajouté dans la classe sur laquelle la méthode *addAttributeNamed:* est appelée. La ligne 4 (respectivement 5) montre la création du *getter* (resp. du *setter*) de l'attribut créé par l'appel à la méthode *createGetter:* (resp. *createSetter:*) de la classe Role.

Comme un rôle peut déléguer une partie de son comportement à l'adaptateur qui se trouve sur la relation entre le rôle et l'objet auquel il est attaché, nous avons ajouté la méthode *addDelegateMethod:* à la classe Role. Cette méthode prend en argument une chaîne de caractère qui décrit la signature de la méthode à créer. Cette signature est utilisée pour générer la méthode dans la sous-classe de Role sur laquelle la méthode *addDelegateMethod:* a été appelée. Le corps de la méthode créée contient un appel à la méthode ayant la même signature sur une instance de PlayRelation pour créer la délégation.

La création dynamique de sous-classes de la classe Role et l'ajout d'attributs permet la mise au point des types de rôles par les modeleurs de rôles. L'utilisation d'un DSL interne permet également de masquer la complexité du langage Smalltalk en offrant des services dédiés de haut niveau. Les types de rôles créés doivent maintenant être instanciés et les instances allouées à des objets conformément au processus détaillé précédemment.

4.3.3. Implémentation de l'allocation des rôles

L'allocation des rôles consiste à créer des instances d'une classe définissant un type de rôle et à associer ces instances avec des éléments de modèles. Les éléments de modèles sont des instances de classes Smalltalk qui représentent les méta-classes des méta-modèles des langages de modélisation. Les classes qui représentent les méta-classes des langages de modélisation héritent toutes de la classe Player représentant la méta-classe du même nom de notre méta-modèle. Par rapport à la Définition 16 du concept d'objet, un objet dans notre implémentation est une instance de la classe Player. Les rôles peuvent également être associés à des rôles, c'est-à-dire à d'autres instances de classes qui définissent des types de rôles.

L'allocation des rôles aux éléments de modèles ou à d'autres rôles peut se faire suivant deux stratégies :

1. pour une instance précise,

2. pour toutes les instances d'un type ou d'un rôle donné qui correspondent à une condition donnée.

Ces deux niveaux permettent d'avoir à la fois une approche grain fin du travail sur les modèles et une approche plus générique. La première approche permet d'avoir un contrôle fin sur les affectations de rôles et permet de faire de la mise au point du modèle de rôles sur des aspects pointus qui ne concernent que quelques instances des modèles système sources. La seconde approche est semblable aux définitions de règles de parcourt d'un modèle en fonction d'un méta-modèle telles les approches utilisées dans les outils de transformation de modèles.

Accès aux objets et aux rôles

Le langage Smalltalk fournit des mécanismes d'introspection. Le langage Smalltalk fournit notamment des méthodes pour récupérer toutes les instances d'une classe spécifique qui existent au sein de l'image Smalltalk sous la forme d'une collection. Cependant, lorsqu'une instance n'est plus utilisée et qu'aucune référence pointe vers cette instance, il n'y a aucune garantie que le ramasse-miettes de Smalltalk ait supprimé l'instance déréférencée de l'image. Ainsi, il est possible d'obtenir un résultat incohérent en récupérant des instances inutilisées qui sont toujours présentes dans l'image Smalltalk.

De plus, Smalltalk est un langage dynamique. Il est possible d'ajouter des attributs à une classe et de donner une valeur initiale à ses attributs dans le constructeur de la classe. Cependant, les instances créées avant la définition des attributs et la re-définition du constructeur de la classe possèdent des attributs avec des valeurs par défaut potentiellement différentes de celles attribuées dans le constructeur modifié.

Afin d'éliminer ces risques, tous les éléments de modèles et tous les rôles en cours d'utilisation sont stockées dans une collection dédiée. Cette collection peut être réinitialisée en fonction des besoins des modeleurs de rôles. Ces derniers ont ainsi un contrôle complet sur les instances stockées dans la collection.

Pour permettre de récupérer des instances spécifiques stockées dans la collection que nous avons définie, une interface à la collection est fournie par le DSL interne. Cette interface comprend la méthode *retrieveInstances* de la classe Player présentée Listing 4.3.

```
1 Player class >> retrieveInstances
2   "Enables to retrieve all the instances of the player in the collection of working players."
3   |players|
4   playersCollection := RoleOrganizer uniqueInstance players.
5   ^(playersCollection select: [ :each | each isKindOf: self ]).
```

Listing 4.3 – Récupération de toutes les instances d'une sous-classe de Player.

La classe RoleOrganizer fournit une encapsulation de la collection contenant toutes les instances de Player (éléments de modèles et rôles). RoleOrganizer est un singleton ainsi toutes les instances de la classe Player sont placées dans la même collection appelée *players*. A la ligne 4, le singleton RoleOrganizer est obtenu par l'appel à la méthode *uniqueInstance* de la classe RoleOrganizer. Il est ensuite possible de récupérer la collection *players* à partir du singleton.

Pour sélectionner des instances spécifiques dans une collection, le langage Smalltalk fournit la méthode *select:*. Cette méthode prend en argument un bloc de code qui est évalué sur tous les éléments de la collection et qui renvoie un booléen. Notre bloc de code, ligne 5, fait appel à la méthode *isKindOf:*. Cette méthode permet de savoir si un objet est une instance de la classe passée en paramètre de la méthode. Nous obtenons donc une collection contenant tous

les objets qui sont des instances de la classe sur laquelle la méthode *retrieveInstances* a été appelée.

Nous devons également avoir la capacité de retrouver une instance en fonction de son identifiant. Ce comportement est implémenté dans la méthode *retrieveInstanceId* présentée dans le Listing 4.4.

```
1 Player class >> retrieveInstanceId: aPlayerId
2   "Enables to retrieve the instance of Player having the identifier specified by aPlayerId."
3   |players|
4   players := self retrieveInstances .
5   ^(players detect: [ :each | each playerId = aPlayerId ] ifNone: nil).
```

Listing 4.4 – Récupération d'une instance de Player possédant un identifiant spécifique.

La méthode *retrieveInstanceId* présentée ligne 1 prend en paramètre l'identifiant d'une instance de Player. Cette méthode récupère toutes les instances de la classe par l'intermédiaire de la méthode *retrieveInstances* (ligne 4) et sélectionne l'unique instance qui porte l'identifiant passé en paramètre (ligne 5) en appelant la méthode *detect:ifNone:*. Si aucune instance ne porte l'identifiant désiré, la méthode retourne *nil*.

Associer un rôle à une instance spécifique

La première stratégie d'association d'un rôle à un objet nécessite de récupérer l'instance souhaitée dans la collection qui stocke toutes les instances de Player à l'aide de la méthode *retrieveInstanceId:*. Lorsque l'instance souhaitée est récupérée, la méthode d'instance *plays:* présentée Listing 4.5 peut être appelée sur cette instance.

```
1 Player >> plays: aRoleClass
2   "Creates the association between an instance of Player and an instance of Role creating the instance of role."
3   |playRelation roleInstance|
4   roleInstance := aRoleClass new.
5   playRelation := self createRelationWith: roleInstance.
6   RoleOrganizer uniqueInstance players add: roleInstance.
7   ^playRelation .
```

Listing 4.5 – Méthode d'instance permettant à une instance de Player de jouer un rôle

La méthode *plays* crée une instance de la classe *aRoleClass* passée en argument de la méthode (ligne 4). L'appel à la méthode *createRelationWith:* permet de créer l'instance de PlayRelation qui sert à associer l'instance de aRoleClass et l'objet sur lequel la méthode *plays:* a été appelée. Le rôle est ensuite ajouté à la collection d'instances nommée players dans le singleton de la classe RoleOrganizer.

Cette approche présente l'intérêt d'avoir un contrôle fin sur les instances de Player et les rôles qui leur sont associés. Cependant, il est également nécessaire d'avoir les moyens de mettre en place une approche permettant d'associer un rôle à un groupe d'instances.

Associer un rôle à plusieurs instances d'un Player

La seconde stratégie impose d'être capable de retrouver toutes les instances d'une classe. Elle repose ainsi sur l'utilisation de la méthode *retrieveInstances*.

Des méthodes de classe sont définies pour réaliser l'association de rôles avec un groupe d'instances. La méthode d'association d'un rôle avec les instances d'une classe donnée sous

une certaine condition est présentée Listing 4.6.

```
1 Player class >> plays: aRoleClass underCondition: aBlock
2   "Specifies for instances of a subclass of Player which match the condition specified in the aBlock parameter the
      played role."
3   |players|
4   players := self retrieveInstances select: [ :anInstance | aBlock value: anInstance ].
5   players do: [ :anInstance  | anInstance plays: aRoleClass.]
```

Listing 4.6 – Méthodes de classe pour faire jouer un rôle à toutes les instances d'un Player sous certaines conditions

La méthode *plays:underCondition:* prend en paramètre une sous-classe de Role ainsi qu'un bloc décrivant une condition qui doit être validée par toutes les instances qui joueront un rôle du type *aRoleClass*. Ligne 4, la méthode *retrieveInstances* est appelée. Les instances qui vérifient la condition définie dans le paramètre *aBlock* sont sélectionnées dans la collection retournée par *retrieveInstances*. La méthode *do:* du langage Smalltalk permet d'appliquer un ensemble d'opérations passé en paramètre sous la forme d'un bloc à tous les éléments d'une collection. Ligne 5, nous utilisons cette méthode pour appeler la méthode *plays:* définie au niveau instance avec *aRoleClass* comme paramètre sur toutes les instances sélectionnées.

Cependant, si toutes les instances d'un Player sans restriction doivent jouer le même rôle, la méthode *plays:* au niveau de la classe Player peut être appelée. Le code est donné dans le Listing 4.7.

```
1 Player class >> plays: aRoleClass
2   "Specifies for each instance of a given Player subclass the Role that will be played."
3   self plays: aRoleClass underCondition: [:each | true].
```

Listing 4.7 – Méthode de classe pour faire jouer un rôle à toutes les instances d'un Player

Cette méthode est un sucre syntaxique. Elle utilise la méthode *plays:underCondition:* en passant un bloc renvoyant vrai dans tous les cas comme condition (ligne 3).

L'ensemble de ces méthodes fournissent les moyens d'allouer dynamiquement des rôles à des instances de Player à la fois dans une approche gros grain et dans une approche grain fin. Ces deux niveaux facilitent la mise au point de l'allocation des rôles pour le modeleur de rôles. En effet, le modeleur de rôle peut travailler soit avec des ensembles d'instances soit avec une instance en particulier.

Dans notre approche, les rôles servent à définir un point de vue sur des modèles système. Ils doivent permettre d'adapter des modèles système à des outils d'analyse. Par conséquent, une fois les rôles définis et associés à des éléments de modèles, l'adaptation des éléments de modèles aux rôles doit être définie et implémentée.

4.4. Définition à la volée des adaptateurs

4.4.1. Processus de définition des adaptateurs

Un adaptateur permet d'adapter un objet au point de vue décrit par le rôle joué par l'objet.

La définition des adaptateurs est un processus collaboratif entre les utilisateurs finaux et les modeleurs de rôles. Ce processus permet de définir les adaptateurs à créer, les relations entre objets et rôles auxquels ils sont attachés et les opérations d'adaptation.

```
1 RoleAdapter class >> retrieveRoleAdapter: aRoleAdapterClassName orCreateAndClassifyIn: aCategoryName
2   "Retrieve the class which name is aRoleAdapterClassName otherwise creates a subclass of RoleAdapter with that
      name"
3   ^Smalltalk at: (aRoleAdapterClassName asSymbol) ifAbsent: [
4     self subclass: aRoleAdapterClassName asSymbol
5     instanceVariableNames: ''
6     classVariableNames: ''
7     poolDictionaries: ''
8     category: aCategoryName
9   ]
```

Listing 4.8 – Méthode de récupération d'un adaptateur dynamique

Le mécanisme d'introspection permet de récupérer les classes définies dans l'image Smalltalk. Différents objets sont stockés et accessibles par l'intermédiaire d'un dictionnaire nommé *Smalltalk* et dont les clés sont des symboles (ligne 3). Si le symbole associé à l'attribut *aRoleAdapterClassName* n'existe pas alors une sous-classe de RoleAdapter portant le nom *aRoleAdapterClassName* et rangée dans la catégorie *aCategoryName* est créée (lignes 4 à 8). L'appel à cette méthode est cependant réalisé lors de l'ajout d'un adaptateur sur une relation.

Ajout de l'adaptateur sur les relations

Des rôles du même type peuvent être associés à des objets de types différents. Un adaptateur permet de faire le lien entre l'objet et le point de vue défini par un rôle. Pour permettre de faire l'adaptation, l'adaptateur est associé à la relation entre un rôle et un objet. Il faut pouvoir associer des instances d'un adaptateur spécifique sur toutes les relations entre des rôles d'un type donné et des objets d'un type donné.

L'appel à la méthode *linkedToKind:withAdapter:* définie dans le Listing 4.9 permet l'ajout d'un adaptateur à des relations entre un rôle et un objet (au sens d'instance de la classe Player).

```
1 Role class >> linkedToKind: aPlayerClass withAdapter: aRoleAdapterClassName
2   "Associates an instance of dynamic adapter of type defined by aSymbol to all the roles associated to instances of
      type aPlayerClass."
3   |selectedInstances|
4   selectedInstances := self retrieveLinkedToKind: aPlayerClass .
5   selectedInstances do: [ :anInst |
6     anInst playRelation adapter:
7       (self retrieveRoleAdapter: aRoleAdapterClassName orCreateAndClassifyIn: self category) new.
8   ].
```

Listing 4.9 – Définition et ajout d'un adaptateur dynamique sur une relation

La méthode *linkedToKind:withInterpreter:* définie ligne 1 associe une instance de la classe *aRoleAdapterClassName* à toutes les relations entre un rôle du type de la sous-classe sur laquelle la méthode est appelée et une instance de la classe *aPlayerClass*.

La méthode retriveLinkedToKind est appelée sur la sous-classe de Role sur laquelle *linkedToKind:withInterpreter:* a été appelée (ligne 4). Toutes les instances de cette sous-classe liées à une instance de la classe *aPlayerClass* sont récupérées. Pour chaque instance récupérée, une instance de la classe *aRoleAdapterClassName* est ajoutée, en tant qu'adaptateur, sur la relation avec l'objet associé. L'instance de *aRoleAdapterClassName* est créée après un appel à la méthode *retriveRoleAdapter:orCreateInCategory:*.

Définition du comportement de l'adaptateur

Une instance de RoleAdapter est associée à une instance de PlayRelation qui fait le lien entre un rôle et un objet.

L'étape de mise au point du comportement, c'est-à-dire le code qui permet la réalisation concrète de l'adaptation, d'un adaptateur utilise les propriétés dynamiques du langage Smalltalk. Pour faciliter la mise au point de ces comportements, trois mécanismes sont utilisés :

— la réécriture de la méthode *doesNotUnderstand:* au sein de la classe PlayRelation,
— la création dynamique de méthodes dans la sous-classe de RoleAdapter de l'instance liée à celle de PlayRelation,
— l'utilisation du debugger Smalltalk pour écrire le code des adaptateurs.

Lorsqu'un rôle délègue son comportement à l'adaptateur, le rôle envoie un message correspondant à ce comportement à l'instance de PlayRelation faisant le lien avec l'instance de Player.

Pour éviter de spécialiser la relation, l'appel est intercepté par l'instance de PlayRelation en utilisant la méthode *doesNotUnderstand:* implémentée par tous les objets Smalltalk. La classe *PlayRelation* redéfinit cette méthode. Le Listing 4.10 montre la redéfinition de la méthode *doesNotUnderstand:* au sein de la classe PlayRelation.

```
1 PlayRelation >> doesNotUnderstand: aMessage
2   |newMessageDesc|
3   "Adding a new parameter to the message."
4   newMessageDesc := self forgeNewMessage: aMessage.
5   self createInterpreterMethod: (newMessageDesc at: 'messageSelector').
6   ^(newMessageDesc at: 'messageDesc') sendTo: adapter .
```

Listing 4.10 – Redéfinition de la méthode doesNotUnderstand: dans la classe PlayRelation

La méthode *doesNotUnderstand:* fait appel à la méthode *forgeNewMessage* pour forger un nouveau message à partir de celui reçu par l'instance de PlayRelation (ligne 3). Un nouveau paramètre qui correspond à l'instance de Player liée à la PlayRelation est ajouté au message reçu par l'instance de PlayRelation. La méthode *createInterpreterMethod* est ensuite appelée (ligne 4). En effet, l'instance de DynamicAdapter liée à l'instance de PlayRelation peut être incapable de répondre au message forgé par l'instance de PlayRelation. Dans ce cas, une méthode est créée avec un comportement par défaut dans la classe de l'adaptateur dynamique. Le message peut ensuite être envoyé (ligne 5).

La méthode créée dans la sous-classe de RoleAdapter possède comme corps un appel à la méthode *halt* implémentée par la classe *Object* de Smalltalk. Cette méthode permet de lancer un debugger interactivement. Smalltalk permet de redéfinir le code des méthodes des classes et de continuer l'exécution sans redémarrer l'application. Ces caractéristiques de l'environnement Smalltalk permettent de définir pas à pas et en fonction des besoins le comportement des adaptateurs.

La définition des rôles et des adaptateurs s'adressent principalement aux modeleurs de rôles. Les méthodes fournies permettent de mettre au point le modèle de rôles. Pour les utilisateurs finaux, il est nécessaire d'automatiser ces tâches.

4.5. Rôles et contextes

L'utilisation des rôles et des adaptateurs dans un environnement de production, c'est-à-dire où l'on ne cherche pas à mettre au point un modèle de rôles mais où l'on souhaite en réutiliser un, doit répondre à deux besoins :

— un besoin de capitaliser la connaissance de l'utilisation des rôles pour accélérer leur utilisation,

— un besoin de dynamisme pour pouvoir adapter les éléments à un point de vue particulier.

Il est possible de répondre à ces besoins en définissant une stratégie d'utilisation des rôles en deux temps. Le premier temps consiste à mettre en place un ensemble de règles génériques. Ces règles imposent un contexte d'utilisation des rôles. Le second temps consiste à faire évoluer ce contexte pour l'adapter aux attentes courantes de l'utilisation des rôles.

Le premier temps de la stratégie implique la mise en place de la notion de contexte pour les rôles. Le deuxième temps de la stratégie d'utilisation des rôles est couvert par les développements réalisés pour permettre la mise au point des rôles et des adaptateurs.

4.5.1. Définition d'un contexte

Un contexte correspond à la réalisation d'un objectif par l'utilisation de rôles. Pour remplir un objectif, il est nécessaire de définir quels rôles doivent être utilisés, comment les rôles sont associés aux objets et comment ces objets sont adaptés au point de vue décrit par les rôles.

La définition d'un contexte implique trois actions :

1. créer le contexte (méthode *named:*),
2. définir les règles d'association des rôles (méthode *addRole:to:underCondition:*)
3. définir les règles d'association des adaptateurs dynamiques (méthode *addAdapter:to:whenLinkedTo:*).

Initialisation d'un contexte

Le Listing 4.11 illustre l'implémentation de l'initialisation d'un contexte au sein de la classe RoleContext.

```
1 RoleContext class >> named: aContextName
2   "Creates a context with the name aContextName"
3   ^(self new contextName: aContextName)
```

Listing 4.11 – Code d'initialisation d'un contexte

La classe RoleContext définit une méthode de classe *named:* qui permet de créer une instance de RoleContexte et de donner la valeur *aContextName* à l'attribut *contextName* de l'instance.

Une fois le contexte créé, il est possible de lui associer des règles.

Définition des règles d'association des rôles

La définition de l'association entre des instances d'un rôle d'un type donné avec des objets sous une condition spécifique peut être réalisée sous la forme d'une règle d'association.

La classe RoleContext fournit des méthodes pour définir des règles d'association des rôles. Le Listing 4.12 donne le code de la méthode *addRole:to:underCondition:* qui permet de définir une de ces règles.

```
1 RoleContext >> addRole: aRoleClass to: aPlayerClass underCondition: aBlock
2   allocationRules add: (RoleAssociationRule player: aPlayerClass role: aRoleClass condition: aBlock) .
```

Listing 4.12 – Méthodes pour ajouter des règles d'association de rôles et d'adaptateurs dynamiques

La méthode *addRole:to:underCondition:* est définie pour permettre d'ajouter une règle d'association entre des rôles du type *aRoleClass* et des instances de la classe *aPlayerClass* sous la condition définie par *aBlock*. Dans le corps de la méthode, une nouvelle règle d'association d'un rôle est créée par l'appel à la méthode de classe *player:role:condition:* de la classe RoleAssociationRule. Cette règle est initialisée avec les paramètres de la méthode puis ajoutée à la collection des règles d'association de rôles nommée *allocationRules* du contexte.

Définition des règles d'association des adaptateurs dynamiques

L'association de rôles à des objets n'est pas suffisante pour permettre d'interpréter des modèles à l'aide de rôles. Des adaptateurs doivent être associés aux relations entre rôles et objets. Le choix des adaptateurs dépend de l'objectif recherché par l'utilisation des rôles et donc du contexte. Les adaptateurs à utiliser sur les différentes relations peuvent être spécifiés sous la forme de règles.

La classe RoleContext fournit une méthode pour définir des règles d'association d'un adaptateur aux relations entre des rôles et des objets. Cette méthode est présentée Listing 4.13.

```
1 RoleContext >> addAdapter: aRoleAdapterName to: aRoleClass whenLinkedTo: aPlayerClass
2   adaptationDefinitions add: (RoleAdapterAssociationRule adapter: aRoleAdapterName player: aPlayerClass role:
      aRoleClass) .
```

Listing 4.13 – Méthode pour ajouter des règles d'association d'adaptateurs dynamiques

La méthode *addAdapter:to:whenLinkedTo:* est définie pour permettre d'ajouter une règle d'association de l'adaptateur nommé *aRoleAdapterName* sur les relations entre des rôles du type *aRoleClass* et des instances de la classe *aPlayerClass*. Dans le corps de la méthode, une nouvelle règle d'association d'un adaptateur est créée par l'appel à la méthode de classe *adapter:player:role:* de la classe RoleAdapterAssociationRule. Cette règle est initialisée avec les paramètres de la méthode puis ajoutée à la collection des règles d'association d'adaptateurs nommée *adaptationDefinitions* du contexte.

Une fois un contexte défini et ses règles d'association ajoutées, il faut le rendre disponible dans l'environnement.

4.5.2. Mise en place du contexte

La définition d'un contexte n'implique pas sa mise en place dans l'outillage. En effet, la définition du contexte est séparée de son utilisation pour permettre la réutilisation du contexte. La mise en place du contexte correspond à l'application des règles définies pour le contexte sur les éléments de modèles et les rôles déjà crées au sein de l'outil. Les règles sont également sauvegardées dans l'outil. Le Listing 4.14 illustre le code servant à la mise en place du contexte.

```
1 RoleContext >> registerContext
2   self registerAllocationRules .
3   self registerAdaptationDefinition .
```

Listing 4.14 – Enregistrement des règles d'association des rôles et des adaptateurs dynamiques

La mise en place d'un contexte implique l'application des règles d'association des rôles et des adaptateurs. Il est de la responsabilité du contexte de mettre en place les règles au sein de l'outil (ligne 2 et 3). Pour chacune des règles, deux actions doivent être entreprises : exécuter la règle pour les modèles déjà existants et enregistrer la règle pour permettre son exécution lors de l'import de nouveaux modèles.

Une règle d'association de rôle définie par la classe RoleAssociationRule possède un attribut *player* qui représente le type des objets qui doivent jouer le rôle du type *aRoleClass*. Cette association peut être soumise à une condition. Cette condition est spécifiée par l'attribut *condition* dans la classe RoleAssociationRule. Pour chaque règle d'association de rôles, chaque instance de la classe définie par l'attribut *player* se voit associer une instance de la classe de l'attribut *aRoleClass* sous la condition de l'attribut *condition*.

La règle est enregistrée auprès de la classe RoleOrganizer. Cette classe est responsable du stockage des éléments de modèles et des instances de rôles. Lors de l'import de nouveaux modèles, les nouveaux éléments seront donc stockés dans une collection définies au sein de la classe RoleOrganizer. Il est donc de la responsabilité de la classe RoleOrganizer de réaliser les associations de rôles et d'éléments de modèles ainsi que celles des adaptateurs. Lors de l'ajout de nouveaux éléments de modèles ou de nouveaux rôles, la classe RoleOrganizer exécute les associations de rôles et d'adaptateurs enregistrées. L'enregistrement des règles d'association des adaptateurs suit le même schéma.

4.5.3. Sérialisation d'un contexte

Par principe, un contexte doit pouvoir être réutilisé. Les contextes sont définis à l'aide des instructions du DSL interne R2DL. Il est donc possible d'utiliser les mêmes instructions pour pouvoir réutiliser le contexte. La réutilisation peut se faire en deux étapes : la sérialisation du contexte sous la forme d'instructions R2DL et de code Smalltalk et l'exécution ultérieure de ces instructions.

La sérialisation d'un contexte en instructions R2DL et en code Smalltalk se fait par l'intermédiaire de la méthode *exportContext* définie dans la classe RoleContext et présentée Listing 4.15

```
1 RoleContext >> exportContext
2   self exportRoles.
3   self exportRolesAssociationRules.
4   self exportAdapterAssociationRules.
```

Listing 4.15 – Export d'un contexte sous la forme d'instruction R2DL

La structure de la classe RoleContext implique que l'export (sa sérialisation) se fait en trois étapes :

- la sérialisation des types de rôles associés au contexte,
- la sérialisation des règles d'association de rôles associées au contexte,
- la sérialisation des règles d'association d'adaptateur associées au contexte.

Comme un type de rôle peut être utilisé dans plusieurs contextes, la sérialisation d'un type de rôle doit permettre de tester l'existence préalable de la classe Smalltalk qui définit le type

des formats de sérialisation permet de définir la stratégie pour les traiter. Deux cas sont possibles :

1. le format de sérialisation est déjà connu pour le langage de modélisation,
2. le format de sérialisation ou le langage de modélisation n'a pas encore été traité.

Dans le premier cas, l'approche modulaire permet de réutiliser directement les importateurs déjà écrits. Dans le second cas, les utilisateurs finaux doivent faire appel à des équipes dédiées à la réalisation des importateurs pour chaque format de sérialisation inconnu. Les utilisateurs finaux restent maîtres de la conception des importateurs. Il est de leur responsabilité de définir leurs besoins pour l'importateur. Ils doivent notamment définir quel langage de modélisation est concerné ainsi que les actions qui doivent être réalisées par l'importateur. Ces informations sont fournies aux concepteurs de l'importateurs.

Ces derniers réalisent l'analyse du format de sérialisation à traiter pour repérer les concepts du langage de modélisation sous leur forme sérialisée et définir les actions associées à leur traitement.

Une fois réalisé, l'importateur est transmis aux utilisateurs finaux pour évaluation. Tant que cette évaluation n'est pas concluante, les étapes de conception et de réalisation sont bouclées. Si l'évaluation est concluante, les importateurs sont combinés pour obtenir un importateur global. La définition d'un importateur global permet de ne pas avoir à demander à l'utilisateur final de sélectionner le format du modèle sérialisé qu'il souhaite importer. De plus, les opérateurs de parsers combinateurs fournissent un mécanisme natif pour vérifier la correspondance d'un fichier à un format donné sans avoir à faire faire ce test par du code propre à l'environnement. La définition du parser global permet d'importer des modèles exprimés dans des langages différents.

4.6.2. Le framework PetitParser

Une implémentation des parsers combinateurs en Smalltalk existe sous la forme du framework PetitParser [75]. PetitParser implémente différents opérateurs de combinaison dont l'opérateur de séquence, de choix et de choix ordonné. Le Listing 4.16 présente l'utilisation de ces opérateurs.

```
1 p1 := 'Block' asParser.
2 p2 := 'Function' asParser.
3 p3 := p1, #blank asParser, p2.
4 p4 := p1 | p2.
5 p5 := p1 / p2.
```

Listing 4.16 – Exemples de code utilisant des parsers combinateurs avec le framework PetitParser

Le parser p1 (ligne 1) est satisfait si son entrée est constituée de la chaîne de caractères *Block*. Le parser p2 (ligne 2) est satisfait si son entrée est constituée de la chaîne de caractères *Function*.

La ligne 3 illustre l'utilisation de l'opérateur de combinaison en séquence qui est la virgule. Elle illustre également l'utilisation de parsers inclus nativement dans le framework PetitParser. Ici, il s'agit d'utiliser un parser pour les caractères d'espacement. Le parser fournit par PetitParser pour réaliser cette tâche est construit par le code *#blank asParser*. Le parser p3 (ligne 3) est construit comme la séquence des parsers p1, d'espacement et p2. Il sera donc vérifié si la chaîne en entrée est constituée de *Block Function*.

Le parser p4 (ligne 4) est un choix entre les parsers p1 et p2. L'opérateur de choix est le caractère barre verticale. Le parser p4 sera donc vérifié si le parser p1 est vérifié et p2 ne l'est pas ou si le parser p1 n'est pas vérifié et p2 l'est.

Le parser p5 (ligne 5) est construit comme un choix ordonné entre p1 et p2. Le parser p5 sera vérifié si le parser p1 est vérifié ou si le parser p1 n'est pas vérifié et le parser p2 est vérifié.

PetitParser est un framework orienté objet pour la construction de parsers. Les parsers sont définis dans des classes qui héritent des classes fournies par le framework. Un parser combiné hérite de la classe *PPCompositeParser* du framework PetitParser. Le parser est défini au sein de la méthode *start* qui doit être redéfinie dans tous les parsers héritant de *PPCompositeParser*.

Au sein d'une sous-classe de *PPCompositeParser*, il est possible d'utiliser des sous-parsers par combinaisons. Ces combinaisons seront définies dans la méthode *start*. Les parsers sont déclarés sous la forme d'un attribut d'instance et d'une méthode ayant le même nom et définissant leur comportement.

Dans la Sous-section 3.5.1, nous avions défini un parser pour une version simplifiée du langage SysML à l'aide de parsers pour les blocs, les flowports et les connexions. En reprenant cet exemple, la Figure 4.8 montre la structure du programme pour le parser pour la version simplifiée du langage SysML à l'aide du framework PetitParser.

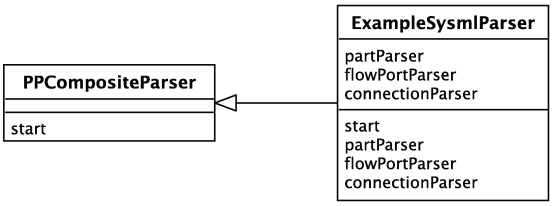


FIGURE 4.8. – Diagramme de classe pour l'implémentation du parser de la version allégée du langage SysML.

Le parser SysML est défini par la classe ExampleSysmlParser qui hérite de la classe PPCompositeParser. Nous avons identifié les parsers partParser, flowPortParser et connectionParser. Ces parsers sont implémentés sous la forme d'attributs et de méthodes dans la classe ExampleSysmlParser. La classe ExampleSysmlParser redéfinit également la méthode *start* de la classe PPCompositeParser. La combinaison des parsers précédents y est défini. Le Listing 4.17 présente l'utilisation des parsers combinateurs du framework PetitParser sur notre exemple de modèle SysML simplifié.

```
1 ExampleSysmlParser >> start
2   ^(partParser / flowPortParser / connectionParser) star.
```

Listing 4.17 – Implémentation de la méthode start dans la classe ExampleSysmlParser

La méthode *start* définit la totalité du parser comme un choix ordonné entre les parsers partParser, flowPortParser et connectionParser. Le symbole / représente l'opérateur de choix ordonné dans le framework PetitParser.

Le parser peut être validé par des entrées qui contiennent plusieurs part, flowport ou connexion. Ceci est illustré par l'utilisation de la méthode *star* sur le parser obtenu par combinaison des parsers partParser, flowPortParser et connectionParser.

PetitParser offre également la possibilité d'ajouter aux parsers des opérations à effectuer lorsqu'ils sont vérifiés. La syntaxe pour ajouter des opérations est *parser* ==> *action* où ==> est l'opérateur ajoutant une action à un parser.

L'orientation objet de PetitParser permet d'utiliser l'héritage pour séparer les parties parsers et actions. Les parsers sont tous définis dans une classe-mère et les classes-filles spécifient les actions associées à chaque parser. Les parsers sont définis dans la classe-mère sous la forme de méthodes. L'ajout d'action consiste à redéfinir les méthodes de la classe-mère dans les classes-filles en appelant la méthode de la classe-mère et en utilisant l'opérateur ==>. Cette approche orientée objet permet de définir une seule fois les parsers et de définir plusieurs actions en fonction des besoins.

En reprenant l'exemple précédent, nous souhaitons écrire un parser pour notre version simplifiée de SysML qui affiche des messages sur la console en fonction de ce qui est analysé. La classe ExampleSysmlParser définit déjà l'analyse de chaînes de caractères décrivant le modèle SysML. L'héritage permet de réutiliser le code déjà écrit. La structure du programme obtenu est présenté Figure 4.9.

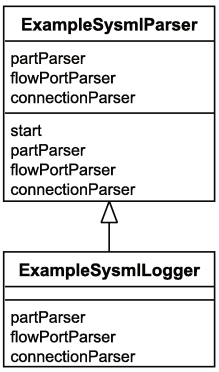


FIGURE 4.9. – Diagramme de classe de l'implémentation de parsers avec actions pour la version allégée du langage SysML.

Nous limitons notre présentation au cas du parser partParser. La classe ExampleSysmlParser définit le parser partParser comme un parser pour la chaîne de caractère *part*. Le Listing 4.18 donne le code de cette définition.

```
1 ExampleSysmlParser >> partParser
2   ^('part' asParser).
```

Listing 4.18 – Implémentation du parser partParser dans la classe ExampleSysmlParser

Le parser est défini en appelant la méthode *asParser* sur la chaîne de caractères *part*. Nous réutilisons le parser ainsi défini. Le mécanisme d'héritage permet à une sous-classe d'appeler les méthodes définies dans sa classe-mère. Ce mécanisme est utilisé pour définir le parser qui écrit des messages sur la console comme illustré dans le Listing 4.19.

```
1 ExampleSysmlLogger >> partParser
2   ^(super partParser ==> [ :nodes | Transcript show: 'Part parsee'.].
```

Listing 4.19 – Implémentation du parser partParser dans la classe ExampleSysmlLogger

Le parser partParser défini par la classe ExampleSysmlParser est appélé via l'appel de méthode *super partParser*. Une action sous la forme d'un bloc est ajouté via l'opérateur ==>. Grâce à cet approche orientée objets des parsers, nous pouvons réutiliser des définitions de parsers et réaliser des actions en fonction du besoin. La séparation entre la partie analyse et la partie traitement permet de s'adapter aisément aux évolutions des langages de modélisation. Ces évolutions peuvent être de deux natures : soit le format de sérialisation est modifié soit la structure du langage de modélisation est modifiée. Les évolutions du format de sérialisation ont des impacts sur la partie analyse tandis que les évolutions de la structure du langage de modélisation ont des impacts sur la partie traitement. La séparation de ces deux aspects permet de circonscrire les modifications à réaliser sur les importateurs pour les adapter aux évolutions des langages de modélisation. La maintenance des importateurs en est facilitée.

4.6.3. Implémentation des importateurs

Les facilités offertes par le framework PetitParser sont utilisées pour écrire les importateurs de modèles. L'import de modèle repose sur deux classes dans Role4All : ModelImporter et ModelLoader.

La classe ModelImporter définit une méthode de classe abstraite appelée *importer*. Cette méthode doit être redéfinie par toutes les sous-classes de ModelImporter. Cette méthode renvoie une instance de PPCompositeParser qui correspond à l'association de tous les parsers définis pour un formalisme donné. L'implémentation de la méthode *importer* dans les sous-classes utilise les opérateurs de combinaison fournis par PetitParser pour composer les parsers des différents formats de sérialisation utilisés pour le formalisme à parser.

La classe ModelLoader est responsable de la création d'un parser global à partir des parsers définis par les sous-classes de ModelImporter. Elle utilise les mécanismes d'introspection pour remplir cette tâche comme illustré dans le Listing 4.20.

```
1 ModelLoader class >> combinedParserCreation
2   "Retrieves all the instances of the subclass of ModelImporter and creates a parser for all possible inputs."
3   |combinedParser|
4   combinedParser := nil.
5   ModelImporter subclassesDo: [ :aModelImporter |
6     combinedParser := self combineParser: combinedParser.
7   ].
8   ^combinedParser .
```

Listing 4.20 – Création d'un parser global en utilisant les mécanismes d'introspection de Smalltalk

La méthode de classe *combinedParserCreation* sélectionne toutes les sous-classes de ModelImporter (ligne 5). La méthode *importer* est appelée sur chaque sous-classe de ModelImporter. Un parser global est construit en combinant les retours de ces différents appels (ligne 6). Le parser ainsi créé est utilisé pour analyser les différents modèles sérialisés qui doivent être importés dans l'outillage.

L'utilisation de l'introspection permet de créer un importateur global à partir de tous les importateurs existant dans l'image Smalltalk. Ainsi, lorsqu'un importateur a été créé pour un langage de modélisation, son intégration dans l'image Smalltalk est suffisante pour permettre de l'intégrer à l'importateur global. Cette caractéristique permet de gérer l'import de modèles créés avec des outils de modélisation qui ne sont plus utilisés. Le même mécanisme permet de gérer l'import de modèles exprimés dans de nouveaux langages de modélisation. En effet,

l'importateur (et donc le parser) pour le nouveau langage de modélisation doit être réalisé et intégré dans l'image Smalltalk. Il sera alors disponible pour la création de l'importateur global.

L'utilisation des parsers combinateurs apporte de la modularité dans la construction de parsers. L'approche orientée objet de PetitParser ajoute de la modularité dans l'ajout d'actions aux parsers. L'utilisation de l'introspection facilite la création d'un parser global à partir de parsers qui ont pu être créés par des équipes différentes. La modularité apportée à ces différents niveaux permet d'importer des modèles existants, de s'adapter aux évolutions des langages de modélisation et prendre en compte de nouveaux langages de modélisation.

En plus de l'implémentation du langage de rôle et des importateurs modulaires à l'aide du framework PetitParser, nous avons également implémenté une interface graphique pour l'environnement Role4All.

4.7. Réalisation d'une interface dédiée

Afin de permettre l'utilisation de l'environnement Role4All, une interface utilisateur dédiée a été implémentée. Cet outillage repose sur l'utilisation des frameworks Smalltalk :

Glamour [13] est un DSL interne en Smalltalk pour la création d'interface du même style que le navigateur de classes natif de Smalltalk. Nous l'avons utilisé pour la réalisation d'une interface de navigation dans les modèles de rôles à l'image du *System Browser* Smalltalk.

Roassal pour la visualisation graphique des relations entre rôles et éléments de modèles.

Mustache pour l'écriture de patron pour l'export de la visualisation en GraphViz.

GraphViz [4] est un outil de visualisation de graphes. Le langage dot est utilisé pour définir un graphe. GraphViz lit un fichier dot et est capable de générer des fichiers image du graphe décrit dans le fichier dot.

L'interface graphique réalisée est présentée Figure 4.10.

L'interface développée est définie dans une classe nommée RoleGui. L'interface développée est constituée de trois parties :

— un navigateur,
— une représentation des modèles et des rôles,
— un éditeur.

Le navigateur permet de visualiser les contextes, les rôles associés aux contextes ainsi que les propriétés des rôles. De nombreux contextes peuvent être définis mais seulement une partie d'entre eux doivent être utilisés. Les contextes à utiliser sont répertoriés dans le singleton de la classe RoleOrganizer (introduite sous-section 4.3.3) après avoir appelé la méthode *registerContext* de la classe RoleContext (présentée sous-section 4.5.2). La Figure 4.11 présente les dépendances entre classes de l'environnement Role4All qui permettent à l'interface graphique de présenter la liste des contextes et des rôles associés.

La classe RoleGui représente l'interface graphique. Elle utilise le singleton de la classe RoleOrganizer. Ce dernier possède une collection de tous les contextes enregistrés et donc utilisables. L'instance de RoleGui fait appel à la méthode *contexts* de la classe RoleOrganizer qui permet de récupérer tous les contextes qui sont répertoriés par le singleton.

Les contextes possèdent une collection de références sur les classes qui définissent les types de rôles qui sont utilisés dans le contexte. Une fois la collection de contextes récupérée par

4. http://www.graphviz.org

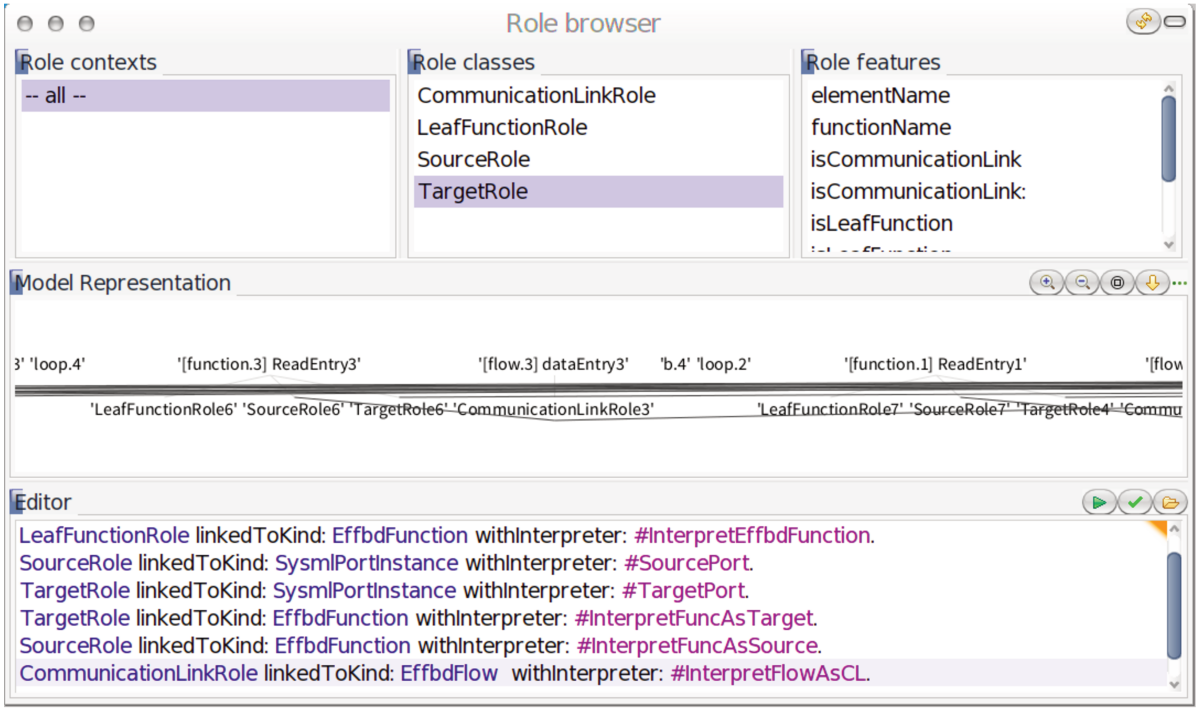


FIGURE 4.10. – Interface graphique de l'outil dédié implémenté

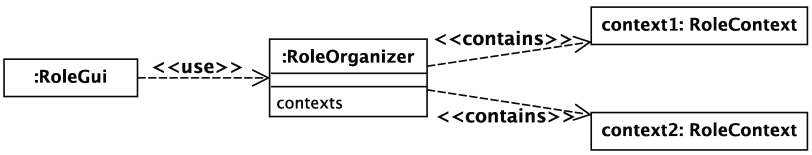


FIGURE 4.11. – Relations de dépendances entre les classes RoleGui, RoleOrganizer et RoleContext

l'interface graphique, il est possible d'accéder pour chaque contexte à la collection des types de rôles. Enfin, pour chaque type de rôles, il est possible d'accéder aux propriétés définies qui sont réalisées sous la forme de méthodes.

Le contexte – *all* – est un contexte par défaut qui permet de lister tous les types de rôles.

La seconde partie de l'interface graphique permet de représenter les modèles système et les modèles de rôles. La représentation des modèles et des rôles permet de visualiser les éléments de modèles avec lesquels il est possible de travailler ainsi que les rôles qui leurs sont associés. Cependant, cette visualisation reste un prototype. Sa lisibilité pour les grands modèles doit être améliorée. Le choix des informations des éléments de modèles et des rôles doit également être étoffé. A l'heure actuelle, seul l'identifiant associé à l'élément de modèle ou au rôle est affiché. Cette information permet à un utilisateur d'accéder à l'élément de modèle ou au rôle désiré en utilisant les méthodes d'accès fournies par le DSL interne R2DL.

L'utilisation de R2DL se fait grâce à la troisième partie de l'interface graphique : l'éditeur. La partie éditeur permet d'utiliser les instructions définies par R2DL qui sont en réalité des portions de code Smalltalk. Les contextes peuvent être sérialisés sous la forme d'instructions R2DL. Cette sérialisation comprend la sérialisation des classes qui définissent les types de rôles incluant leurs attributs et leurs méthodes. Elle comprend également les règles d'association de rôles et d'adaptateurs qui sont définies au sein du contexte. R2DL étant un DSL interne, la sérialisation peut contenir des instructions R2DL et du code Smalltalk. Le résultat de la sérialisation peut être stocké dans un fichier. Le contenu de ce fichier peut être réutilisé soit par l'intermédiaire de l'éditeur soit directement par l'environnement Smalltalk. Ce mécanisme permet de réutiliser des contextes déjà définis et ce directement à partir de l'interface de Role4All.

Afin d'aider l'utilisateur, différentes actions sont disponibles via des boutons :

— le chargement de modèles dans l'environnement Role4All,
— l'exécution de code,
— l'export de la visualisation au format GraphViz.

Le bouton de chargement de modèles permet de faire appel à la méthode load de la classe ModelLoader. Cette action a pour effet d'ouvrir un navigateur de fichier pour trouver le fichier contenant la description sérialisée du modèle à charger et de créer l'importateur global. La Figure 4.12 illustre les dépendances nécessaires au sein de l'environnement Role4All pour la création de l'importateur global.

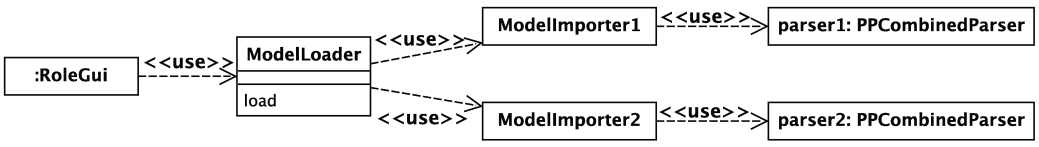


FIGURE 4.12. – Dépendances entre la définition de l'interface Role4All et les parsers servant à l'import de modèles.

L'instance de la classe RoleGui appelle la méthode *load* de la classe ModelLoader à chaque fois que le bouton de chargement de modèle est pressé. L'appel à cette méthode permet de retrouver, en utilisant l'introspection, toutes les sous-classes de la classe ModelImporter. Ces classes définissent un parser pour des modèles sérialisés. Il est donc possible de récupérer chaque parser associé à une sous-classe de ModelImpoter et de les combiner en utilisant des opérateurs de combinaison. Un importateur global est ainsi obtenu. La création de l'importateur global à

chaque pression sur le bouton de chargement permet de prendre en compte des importateurs qui auraient été définis entre deux appuis successifs sur le bouton. Ainsi, il n'est pas nécessaire de redéfinir ou de redémarrer l'environnement Role4All pour prendre en compte les nouveaux importateurs.

Le bouton d'exécution de code permet d'exécuter le code écrit dans la partie éditeur de l'interface.

Le bouton d'export de la visualisation permet de créer un fichier dot pour permettre la représentation de modèles comportant un grand nombre d'éléments. Pour les modèles de grande taille et ayant de multiples relations avec des rôles, la visualisation basique fournie par l'interface peut se révéler insuffisante. Pour remédier à ce problème, il est possible d'exporter l'ensemble du modèle et les relations avec les rôles au format graphviz. Cet export permet de générer une image représentant la hiérarchie entre éléments de modèles et rôles.

L'outillage créé est exécuté au sein d'une image Smalltalk. Tous les outils natifs Smalltalk comme :

l'explorateur système qui permet d'accéder à la structure des classes définies dans l'image Smalltalk,

l'inspecteur qui permet d'accéder a la valeurs des attributs d'une instance,

le debogueur qui permet de corriger du code.

L'annexe A illustre cette possibilité. Dans la partie représentation du modèle de rôles de l'interface de Role4All, nous avons repéré l'objet d'identifiant *pI12* qui joue le rôle d'identifiant *SourceRole0*. Nous souhaitons connaître ses attributs et accéder à la définition de sa classe. Cependant, nous ignorons la classe de l'objet d'identifiant *pI12*. Nous avons donc utilisé la partie éditeur pour exécuter une instruction R2DL qui permet de récupérer l'instance de l'objet d'identifiant *pI12*. Nous avons utilisé l'inspecteur pour connaître la classe de l'objet récupéré qui est SysmlPortInstance. Enfin, nous avons pu utiliser l'explorateur système pour accéder à la définition de la classe SysmlPortInstance. Cet enchaînement d'actions est typiquement celui utilisé pour l'écriture des adaptateurs.

4.8. Conclusion

Nous avons défini un environnement, nommé Role4All, pourvu d'un DSL interne, nommé R2DL (Role4All Role Description Language), pour manipuler des modèles système hétérogènes. Nous pouvons interfacer ces modèles avec différents outils d'analyse. Cet environnement utilise le concept de rôle pour interfacer les outils d'analyse avec les modèles importés dans Role4All à l'aide d'importateurs modulaires utilisant des parsers combinateurs.

Afin de rendre Role4All opérationnel, nous avons conduit une analyse des services que l'environnement doit rendre. Nous avons identifié deux populations cibles :

— Les modeleurs de rôles qui doivent créer les modèles de rôles pour interfacer modèles système et outils d'analyse.

— Les utilisateurs finaux qui utiliseront les rôles dans un contexte opérationnel.

Les modeleurs de rôles doivent pouvoir définir les rôles, leurs associations avec des éléments de modèles et les actions d'adaptation des éléments de modèles aux rôles. Les utilisateurs finaux des rôles doivent pouvoir réutiliser le travail des modeleurs de rôle directement dans leurs contextes opérationnels.

Pour réaliser ces fonctions, nous avons implémenté Role4All en Smalltalk. Smalltalk est un langage typé dynamiquement. Smalltalk permet également de redéfinir des classes et des

méthodes à la volée sans nécessiter de redémarrer l'exécution d'un programme. Il s'agit donc d'un environnement intéressant pour la mise au point puis l'utilisation ultérieure des modèles de rôles. L'ensemble des opérations a fait l'objet de la conception d'un DSL interne qui fournit une abstraction de la syntaxe du langage Smalltalk orientée définition et utilisation d'un modèle de rôles.

Pour être indépendant des outils de modélisation système, les modèles système réalisés doivent être importés au sein de l'environnement Role4All. Pour cela, nous avons défini des importateurs modulaires à base de parsers combinateurs. Le framework PetitParser, disponible dans le langage Smalltalk, permet d'écrire des parsers en utilisant une approche orientée objets. Il est donc possible d'écrire des parsers modulaires et réutilisables.

En utilisant les aspects dynamiques du langages Smalltalk, nous permettons de mettre au point de manière incrémentale des types de rôles et des modèles de rôles. En effet, il est possible de créer de nouvelles classes au sein de l'environnement Smalltalk qui correspondent aux types de rôles. Cette création peut se faire à la volée sans nécessiter le redémarrage de l'environnement Smalltalk. De plus, les propriétés des rôles et les méthodes des adaptateurs peuvent être définies lors de l'exécution de requêtes sur le modèle de rôles. Des erreurs de conception du modèle de rôles peuvent être corrigées à l'exécution des requêtes sur le modèle de rôles sans nécessiter de reprendre l'ensemble des requêtes précédentes. Ce comportement facilite le prototypage rapide.

Notre implémentation a permis de mettre en place l'ensemble de l'architecture nécessaire à l'utilisation de rôles pour manipuler des modèles système hétérogènes et évolutifs. Cependant, il est nécessaire de rechercher une combinaison performante de parsers pour réaliser l'import des modèles sérialisés. Un nouveau découpage des importateurs en fonction des formats de sérialisation puis des langages de modélisation et non pas l'inverse pourrait répondre en partie à cette limitation. De plus, notre implémentation se limite à récupérer de l'information depuis les modèles système. Afin de pouvoir utiliser pleinement les principes de fédération de modèles, nous devons implémenter une relation bi-directionnelle entre les rôles et les objets, instanciés dans la représentation interne de Role4All, auxquels ils sont liés.

Nous avons défini les concepts de base de l'environnement Role4All et fourni une implémentation en Smalltalk. L'approche peut maintenant être validée sur un exemple de système de systèmes (SdS).

5

Validation dans le contexte Système de Systèmes

Sommaire

5.1. Introduction 114
5.2. Contexte du cas d'étude 114
5.2.1. Description de l'observatoire sous-marin MeDON 116
5.2.2. Modélisation des sous-systèmes d'un observatoire 119
5.2.3. Problèmes soulevés 122
5.3. Import de modèles hétérogènes 125
5.3.1. Définition des représentations internes des méta-modèles 125
5.3.2. Analyse des formats de sérialisation des modèles 128
5.3.3. Définition des parsers pour importer les modèles sérialisés 129
5.3.4. Intégration des parsers à l'environnement Role4All 133
5.4. Manipulation de modèles hétérogènes 134
5.4.1. Analyse de l'outillage d'exécution de modèles 135
5.4.2. Définition des types de rôles 138
5.4.3. Définition des associations des rôles avec les éléments de modèles . . . 140
5.4.4. Définition des adaptateurs 141
5.4.5. Définition d'un contexte 143
5.4.6. Exécution du modèle 144
5.5. Adaptation du modèle de rôles pour permettre l'utilisation d'un nouvel environnement de simulation 146
5.5.1. Analyse de l'outil de simulation de réseau OMNeT++ 146
5.5.2. Modèle de rôles associé à des rôles 148
5.5.3. Association de rôles avec des rôles 149
5.5.4. Ecriture du visiteur 150
5.5.5. Simulation obtenue 151
5.6. Adaptation à un remplacement d'outil de simulation 153
5.6.1. Présentation du simulateur ADEVS 153
5.6.2. Ecriture d'un visiteur pour la création des éléments de modèle ADEVS 154
5.6.3. Ecriture d'adaptateurs dynamiques 155
5.7. Adaptation à un changement dans les langages de modélisation . 156
5.7.1. Modification du méta-modèle du langage SysML et du format de sérialisation 157

5.7.2. Impact sur le parser pour le langage de modélisation 158
5.7.3. Impact sur le modèle de rôles 159
5.8. Génération de documentation 161
5.8.1. Analyse du générateur de documentation 161
5.8.2. Définition du modèle de rôle et des adaptateurs dynamiques 162
5.9. Conclusion 163

5.1. Introduction

Nous avons défini un environnement nommé Role4All associé à un DSL nommé R2DL (Role4All Role Description Language) pour la manipulation de modèles hétérogènes. Nous sommes capables d'unifier plusieurs modèles chaque modèle pouvant être réalisé dans un langage de modélisation différent. Cet environnement utilise la notion de rôles pour interfacer des modèles système avec des outils de simulation. Les modèles sont importés dans l'environnement Role4All à l'aide d'importateurs modulaires écrits en utilisant des parsers combinateurs.

Un rôle permet de définir un point de vue sur des éléments de modèles indépendamment de leur type. Les parsers combinateurs sont des opérateurs qui permettent de combiner des parsers pour définir un parser plus général.

L'environnement Role4All a été réalisé en Smalltalk.

Nous illustrons l'utilisation de cet environnement sur un système de systèmes réel : l'observatoire sous-marin MeDON.

Les objectifs de ce chapitre sont :

— de présenter le système de systèmes étudié qui est l'observatoire sous-marin MeDON (section 5.2),
— de montrer la modularité, la réutilisabilité et l'extensibilité des importateurs modulaires à base de parsers combinateurs (section 5.3) [1],
— de montrer la capacité à unifier des modèles hétérogènes en n'impactant ni les modèles ni la définition des langages de modélisation (section 5.4) [2],
— de montrer la capacité de l'environnement à prendre en compte un changement d'outil de simulation (section 5.6),
— de montrer la capacité de l'environnement à permettre l'utilisation de plusieurs outils de simulation du même type (section 5.5),
— de montrer l'adaptabilité de l'environnement aux évolutions des langages de modélisation (section 5.7),
— de montrer la capacité de l'environnement à être utilisé dans des contextes différents comme la documentation basée modèles de système (section 5.8).

5.2. Contexte du cas d'étude

Pendant deux ans, nous avons travaillé sur le projet européen d'observatoire sous-marin MeDON (Marine eData Observatory Network) [61]. Ce projet impliquait l'Ifremer, le Plymouth

1. L'utilisation des parsers combinateurs dans ce contexte a fait l'objet de publications dans le workshop SESoS 2014 [83] et dans la conférence WETICE 2015 [79]
2. Cette partie du travail a fait l'objet de publication dans les conférences SysCON 2015 [80] et WETICE 2015 [79]

Marine Laboratory (institut de recherche sur l'environnement marin), le Marine Institute de l'université de Plymouth, l'ENSTA Bretagne, Océanopolis et le National Marine Aquarium de Plymouth. Ce projet a permis la mise au point et le déploiement au sein du parc marin d'Iroise au large de Brest d'un observatoire sous-marin côtier. Cet observatoire avait pour objectif d'étudier les habitudes de la population de dauphins y résidant ainsi que de relever différents paramètres marins comme le courant ou la salinité sur une longue période. Même si ce projet a atteint son objectif, différents problèmes ont émaillé son déroulement.

L'organisation du projet a donné la gestion de projet à l'institut anglais tandis que les aspects réalisations techniques étaient sous la responsabilité de l'institut français. Cette organisation transfrontalière et multi-organisations du travail nécessitait de mettre en place des moyens d'organisation du travail et de communications entre les différents organismes.

Nous avions notamment besoin de communiquer autour des besoins scientifiques du projet. Ces besoins étaient de pouvoir observer la faune sous-marine notamment les dauphins et de surveiller sur le long terme différents paramètres marins. Ces besoins ont été traduits sous la forme d'exigences de haut niveau pour l'observatoire. Ces exigences ont été classiquement diffusées sous la forme d'un document à tous les partenaires. Ces derniers avaient la charge de décliner ces exigences dans leur domaine d'expertise et de réaliser la conception de la partie de l'observatoire à leur charge. Cette séparation des tâches a posé le problème de la vérification de chaque partie conçue avec les exigences globales de l'observatoire en tenant compte des autres parties de l'observatoire.

Pour réaliser la conception des systèmes à leur charge, les différents partenaires ont utilisé des méthodes différentes. La partie mécanique de l'observatoire a été modélisée sous la forme de modèle SADT par Ifremer. La partie réalisation informatique du système d'information a été modélisée sous la forme de service à l'aide du langage de modélisation UML par l'équipe de l'ENSTA Bretagne en charge du système d'information dont nous faisions partie. Les spécificités de la partie matérielle du système d'information ont été décrites dans un document papier par Ifremer. Enfin, les algorithmes de traitement du signal nécessaires pour l'étude des dauphins et la description des hydrophones sous la responsabilité de l'équipe d'acousticiens de l'ENSTA Bretagne ainsi que la description des capteurs de paramètres marins sous la responsabilité d'Ifremer ont également été définis dans des documents papiers. L'architecture fonctionnelle globale des capteurs a été réalisée sous la forme d'un modèle SysML tandis que l'architecture fonctionnelle des algorithmes de traitement a été réalisée avec le langage de modélisation d'architecture fonctionnelle eFFBD.

L'utilisation des modèles pour les parties mécaniques et la réalisation logicielle ont permis de s'accorder sur les interfaces matérielles et sur la structure des données informatiques. Cependant, des spécifications manquaient aux interfaces avec les aspects de l'observatoire définis en version papier. Ces manques étaient notamment causés par des incompréhensions de la documentation par les différentes parties ou par des évolutions des spécifications passées inaperçues. Ces manques ont provoqués des problèmes lors de l'intégration des différents systèmes les uns avec les autres. Ces problèmes ont pu être corrigés en reprenant les réalisations informatiques du système d'information qui étaient les plus simples à modifier. Cependant, ces problèmes auraient pu être anticipés si une vision globale du système avait été partagée. Cette vision globale pouvait être obtenue par l'utilisation de simulations. De plus, les conceptions des différents systèmes auraient pu être validées dès les premières phases de conception. Les corrections des problèmes d'interfaces auraient ainsi été plus rapides.

La simulation implique l'utilisation de modèles. Tous les partenaires auraient donc dû utiliser des modèles. Cependant, même si cela avait été le cas, tous les partenaires auraient utilisé des

langages de modélisation proches de leur domaine d'expertise donc différents les uns des autres. La capacité de travailler avec des modèles différents pour en faire des simulations nous aurait donc manqué.

Dans cette section, nous utilisons l'exemple de l'observatoire MeDON pour illustrer comment l'utilisation de Role4All nous aurait permis de créer des simulations dans les phases amonts du cycle de conception à partir de modèles exprimés dans des langages différents. La Sous-section 5.2.1 décrit l'observatoire sous-marin MeDON. La Sous-section 5.2.2 propose une modélisation fonctionnelle de MeDON. La Sous-section 5.2.3 décrit les problèmes qui sont soulevés par l'exemple.

5.2.1. Description de l'observatoire sous-marin MeDON

Un observatoire sous-marin est un système de systèmes destiné à l'observation de phénomènes physiques et biologiques. La Figure 5.1 présente la structure de l'observatoire sous-marin MeDON.

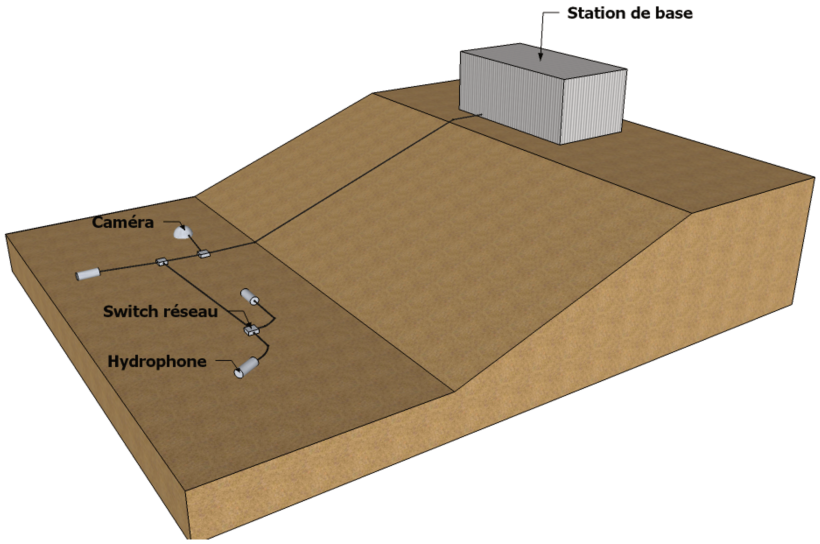


FIGURE 5.1. – Structure générale d'un observatoire sous-marin.

Un observatoire sous-marin est constitué d'un ensemble de capteurs (hydrophones, caméras,...), d'une infrastructure réseau (switch réseau, câble, station de base) et d'un ensemble de programmes informatiques distribués.

L'infrastructure réseau comprend les éléments déployés sous l'eau tels que les câbles et les switch réseau dans le cadre d'un réseau câblé ainsi que les moyens physiques mis à disposition pour l'exécution des programmes informatiques.

La partie programme informatique comprend les programmes de traitements de données, les bases de données, les serveurs d'applications, la gestion des accès aux données stockées,...

La conception et le développement d'un observatoire sous-marin requiert une expertise approfondie dans les domaines de la mécanique, de l'électronique, de l'informatique et du monde marin.

Afin de permettre la réalisation d'un observatoire, il est important d'identifier clairement quels sont tous les acteurs qui interviennent dans la spécification, la conception, le dévelop-

pement et le déploiement d'un observatoire. Cette identification est un préalable à la phase d'élicitation des exigences du système [38]. Les acteurs peuvent être directement impliqués dans le processus de conception ou poser des contraintes sur le système. La Figure 5.2 identifie les acteurs majeurs impliqués dans le processus de conception d'un observatoire sous-marin que ce soit du point de vue besoin ou de la solution technique.

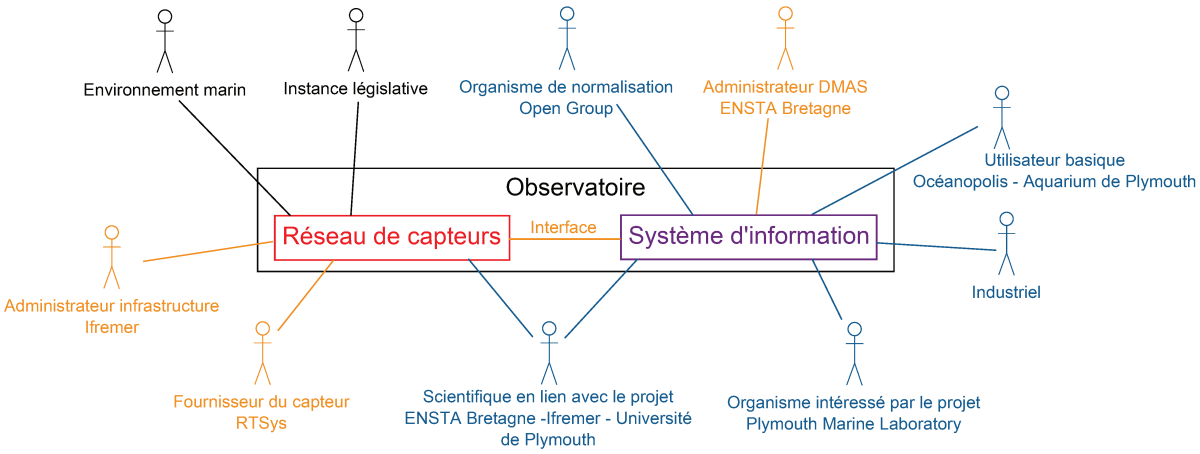


FIGURE 5.2. – Identification des acteurs impliqués dans la réalisation d'un observatoire sous-marin.

L'installation d'un observatoire sous-marin doit tenir compte des contraintes liées à l'environnement marin pour permettre de réaliser les meilleures observations possibles du phénomène étudié. Ces contraintes sont dictées par les scientifiques qui financent le projet. L'un des objectifs du projet MeDON était d'observer la faune sous-marine et notamment les dauphins. Il s'agissait d'observer le comportement des dauphins sur le long terme pour tenter d'y déceler des comportements répétitifs en fonction des moments de la journée ou du cycle lunaire qui pouvait influencer le comportement des dauphins. Il fallait donc installer l'observatoire dans une zone où la présence de dauphins était avérée.

Cependant, l'installation d'un observatoire ne peut pas se faire dans certaines zones réglementées. Il est nécessaire de tenir compte des obligations légales qui définissent les actions autorisées dans chaque zone. Ces contraintes sont encore plus forte dans les zones côtières qui sont considérées comme protégées comme la parc marin d'Iroise où l'observatoire MeDON a été déployé. L'obtention des autorisations auprès des autorités en charge de la gestion de l'espace maritime est obligatoire avant tous travaux. Ces deux points ont une grande influence sur l'installation géographique des différents éléments de l'observatoire. Le choix de la zone géographique d'installation ayant notamment des implications sur le choix des solutions technologiques à mettre en œuvre.

Pour pouvoir étudier les populations de dauphins dans le parc marin d'Iroise une solution de détection par acoustique passive, sans émission sonore pour la détection, a été choisie. Cette solution implique l'utilisation d'hydrophones sélectionnés par l'équipe d'acousticiens de l'ENSTA Bretagne. MeDON devait également permettre de surveiller sur le long terme les courants marins dans le parc marins d'Iroise à l'aide d'ADCPs, des capteurs permettant de mesurer l'intensité d'un courant marin, utilisés par l'Ifremer. Une caméra opérée par Ifremer devait également être utilisés pour observer la flore et la faune marine. Tous ces capteurs peuvent être achetés sur étagère auprès de fournisseurs ou bien conçus et développés par les scientifiques

associés au projet d'observatoire. Il est ainsi possible d'avoir des capteurs provenant de diverses parties qui devront être intégrés au réseau par l'administrateur de l'infrastructure (Ifremer). Ce dernier doit imposer les standards de connectiques à utiliser et s'assurer de la bonne intégration physique des capteurs au sein du réseau.

L'intégration des capteurs au système informatique est réalisée par l'administrateur du système d'information également appelé en anglais Data Management and Archiving System (DMAS). Nous étions en charge de l'administration du système d'information. Nous étions responsables de la gestion de l'ensemble des programmes informatiques. Ces programmes permettent la communication entre les systèmes distribués constituants l'infrastructure informatique. Nous travaillions en coopération avec l'administrateur de l'infrastructure réseau qui est responsable de l'aspect matériel.

Le système d'information fournit une interface aux utilisateurs des données produites comme Océanopolis ou l'aquarium de Plymouth qui pouvaient utiliser les données acquises par l'observatoire pour des opérations de sensibilisation. Les scientifiques de l'ENSTA Bretagne et de l'université de Plymouth en lien avec le projet sont un cas particulier d'utilisateurs. Ces derniers étaient responsables de la définition des algorithmes de traitement du signal qui permettaient la détection des dauphins dans les signaux sonores acquis par les hydrophones.

Maier a défini cinq critères pour déterminer si un système est un système de systèmes [57] :

— l'indépendance managériale,
— l'indépendance fonctionnelle,
— l'évolution,
— la coopération,
— la distribution géographique.

La conception et le développement ou l'acquisition des capteurs étaient sous la responsabilité de l'Ifremer, du Plymouth Marine Laboratory et de l'équipe d'acousticiens de l'ENSTA Bretagne. La conception et le développement du système d'information était sous la responsabilité de l'équipe de développement logiciel de l'ENSTA Bretagne. Enfin la conception et le développement de l'infrastructure mécanique et réseau était sous la responsabilité d'Ifremer. Chaque système constituant l'observatoire MeDON est donc géré de manière indépendante.

Les hydrophones doivent pouvoir fonctionner indépendamment des caméras et des ADCPs. La panne de l'un des capteurs ne doit pas affecter les autres capteurs ni le système d'information. De même, la panne du système d'information ne doit pas empêcher les hydrophones et les ADCPs de réaliser des mesures. Les différents capteurs et le système d'information sont donc fonctionnellement indépendants.

MeDON était destiné à fonctionner plusieurs années. Le nombre des capteurs pouvait augmenter tout comme de nouveaux hydrophones devaient pouvoir être intégrés au réseau de capteurs pour remplacer les hydrophones déployés à l'origine. De même, le logiciel devait pouvoir évoluer pour répondre à de nouveaux besoins en terme d'accès aux données. Ainsi, les constituants de l'observatoire MeDON pouvaient évoluer.

Pour permettre aux utilisateurs extérieurs comme Océanopolis ou l'aquarium de Plymouth, les données recueillies par les hydrophones et les ADCPs ainsi que les données issues des traitements pour la détection des dauphins devaient être échangées au sein de l'observatoire. Par conséquent, les constituants de l'observatoire MeDON devaient collaborer.

Afin de permettre de faire des mesures valides, les hydrophones ne sont pas placés au même endroit. De plus, ils doivent être éloignés des ADCPs qui peuvent générer des interférences étant donné qu'ils émettent des sons. Enfin, les capteurs sont immergés et reliés à une station de base situé sur la côte. Les constituants de l'observatoire MeDON sont donc distribués

géographiquement.

L'observatoire MeDON peut donc être considéré comme un système de systèmes. Malgré les problèmes découverts lors de la phase d'intégration du système, l'observatoire MeDON a été déployé au début de l'année 2012. MeDON a été déployé avec deux hydrophones pour permettre la détection de dauphins dans le parc marin d'Iroise. Un ADCP a également été déployé pour mesurer les courants. Une caméra est également installée et gérée par Ifremer. Les données récupérées par les hydrophones sont traitées et le résultat de ces traitements est accessible sur le site internet dédié aux données MeDON [60].

5.2.2. Modélisation des sous-systèmes d'un observatoire

Un des objectifs du projet MeDON était de détecter les dauphins. Une évolution du projet est donc de permettre de localiser les dauphins détectés. Lors de la conception de l'observatoire MeDON, l'utilisation de modèles avait permis, pour les domaines où ils avaient été utilisés, de définir les interfaces facilitant le travail de développement et d'intégration du système réalisé. Au vu de ces avantages, l'utilisation de modèles pour les premières phases de la conception des systèmes constituants l'observatoire a été étendue à tous les constituants du système. La simulation est également utilisée pour vérifier les modèles dans ces premières phases de la conception de la fonctionnalité de localisation des dauphins.

Pour localiser des dauphins, il est nécessaire de disposer :

— d'un ensemble d'hydrophones,

— d'un système de traitement de données.

L'ensemble d'hydrophones est sous la responsabilité des acousticiens de l'ENSTA Bretagne tandis que le système de traitement de données reste sous notre responsabilité. En plus de la différence des équipes responsables de la conception et du développement des systèmes, ces derniers concernent des domaines d'expertise différents. La modélisation de ces systèmes implique donc des concepts différents. Par conséquent, les outils et les langages de modélisation associés sont différents. Nous avons donc modélisé les hydrophones et le système de traitement de données dans deux langages de modélisation différents.

Les capteurs ont été modélisés à l'aide du langage SysML [29]. Dans le cadre du projet MeDON, les capteurs utilisés sont des hydrophones permettant d'acquérir des données sur l'environnement sonore sous-marin. La Figure 5.3 représente un diagramme de blocs réalisé pour définir la structure fonctionnelle d'un hydrophone déployé au sein de l'observatoire MeDON.

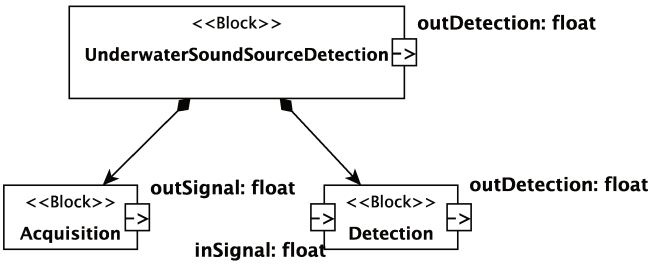


FIGURE 5.3. – Block Definition Diagram correspondant à un extrait de l'analyse fonctionnelle d'un hydrophone déployé dans le cadre du projet MeDON.

Une approche descendante a été utilisée pour définir l'architecture fonctionnelle d'un hydro-

phone dans le cadre du projet MeDON. La fonction principale d'un hydrophone est d'acquérir les signaux sonores sous-marins. Dans le cas du projet MeDON, les hydrophones devaient posséder une intelligence minimale leur donnant la capacité d'analyser le signal pour y détecter la présence d'émetteurs de signaux particuliers comme les dauphins. De ce fait, la fonction principale d'un hydrophone du projet MeDON est la détection de sources sonores sous-marine. Cette fonction est modélisée par le bloc SysML UnderwaterSoundSourceDetection. Cette fonction est constituée de deux sous-blocs : un bloc d'acquisition et un bloc de détection. Le bloc d'acquisition représente la fonction responsable de l'acquisition du signal brut. Le bloc de détection représente la fonction qui permet de tester la présence d'une source sonore dans le signal acquis. Les deux blocs possèdent des flowports par lesquels les échantillons de signaux peuvent être échangés.

La trilatération est un algorithme qui repose sur l'utilisation de la différence des distances estimées entre une source sonore et des capteurs. Dans le cadre d'un observatoire sous-marin comme MeDON, un hydrophone est choisi comme référence et c'est la différence des temps d'arrivée d'une détection entre l'hydrophone de référence et chaque hydrophone qui est utilisée pour calculer la position de la source. Une détection correspond à un signal dont l'amplitude est plus élevée que celle du bruit ambiant estimé. Pour notre problème, quatre hydrophones permettent de déterminer une solution exacte. En pratique, seulement trois peuvent être utilisés. En n'utilisant que trois hydrophones, l'algorithme donne deux solutions possibles. Or, les hydrophones sont posés sur le fond. Par conséquent, il est possible d'éliminer la solution qui est aberrante par rapport à la profondeur de la source [105].

Le choix de cet algorithme par les acousticiens de l'ENSTA Bretagne implique la modélisation du déploiement de trois instances de la fonction UnderwaterSoundSourceDetection tel que décrit dans le diagramme de blocs internes Figure 5.4.

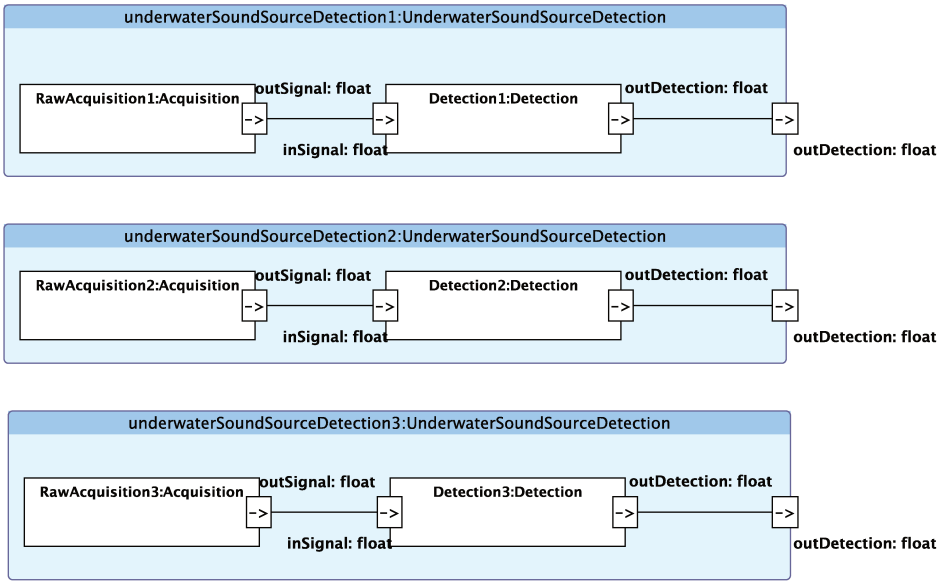


FIGURE 5.4. – Internal Block Diagram correspondant à l'analyse fonctionnelle des hydrophones déployés dans le cadre du projet MeDON.

Chaque instance de la fonction UnderwaterSoundSourceDetection est constituée d'une ins-

tance de la fonction Acquisition et d'une instance de la fonction Detection. Un diagramme de bloc interne SysML sert à décrire l'instanciation des blocs SysML mais aussi à définir les relations qui existent entre ces instances notamment en définissant les liens entre les ports associés à chaque bloc. Le flowport de sortie d'une instance Acquisition est lié au flowport d'entrée d'une instance Detection. Le flowport de sortie d'une instance Detection est lié au flowport de sortie d'une instance UnderwaterSoundSourceDetection englobante. Ce dernier port représente la production d'information par les fonctions de détections de source sonores sous-marines.

Les approches de conception basée composants sont très utilisées dans le monde de l'informatique embarquée. Cette utilisation fait du langage SysML qui est dérivé du langage UML un choix naturel. La conception fonctionnelle d'un système d'information s'intéresse plus aux flux d'informations qui sont échangés entre les fonctions du système d'information. Le langage eFFBD [55] permet de modéliser des architectures fonctionnelles en se concentrant sur les flux entre fonctions. Nous avons donc choisi ce langage pour modéliser le système d'information plutôt que le langage SysML. Le résultat de la modélisation est présenté Figure 5.5.

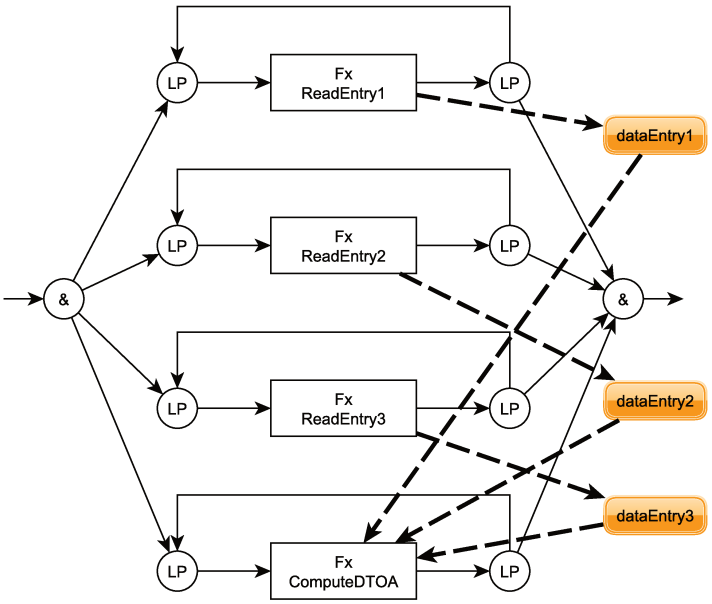


FIGURE 5.5. – Modèle eFFBD illustrant l'architecture fonctionnelle du système d'information de l'observatoire sous-marin MeDON.

Le système d'information est composé de quatre fonctions principales : trois fonctions permettant de recevoir les données des capteurs (ReadEntry1, ReadEntry2 et ReadEntry3) et une fonction réalisant la localisation de sources sonores à l'aide d'un calcul sur la différence des temps de détection des signaux sonores (ComputeDTOA). Ces différentes fonctions s'exécutent en parallèle. Les fonctions s'échangent des flux (dataEntry1, dataEntry2, dataEntry3) qui correspondent aux données reçues par les fonctions de réception et qui servent d'entrée aux calculs effectués par la fonction de localisation.

5.2.3. Problèmes soulevés

Les moyens humains, matériels et financiers impliqués dans le déploiement de l'observatoire sous-marin MeDON sont conséquents du fait de l'environnement hostile dans lequel le système est déployé. En effet, les capteurs sont déployés sous l'eau, ils doivent donc être étanches au risque de ne pas fonctionner. De plus, les câbles sous-marins en zone côtière peuvent être endommagés par les activités de pêche. Les opérations sous-marines de maintenance sont également rendues difficiles par le manque de visibilité et le besoin de personnels et de matériels adaptés à ces activités.

Cela impose que le système déployé doit être opérationnel dès sa mise en route. Pour rendre cela possible, des prototypes doivent être réalisés pour permettre de tester et comparer les choix d'architectures.

Cependant, le coût du matériel est prohibitif. En effet, il s'agit de matériels durcis pour résister à l'environnement marin. Ce sont également des éléments qui ne sont pas produits en grande série ce qui augmente leur coût.

L'utilisation de matériels réels n'est donc pas possible, le prototypage doit être virtuel. L'utilisation d'un prototype virtuel permet de vérifier au plus tôt la conception des systèmes constituants l'observatoire sous-marin. En effet, en ne nécessitant pas de produire les constituants de l'observatoire, l'utilisation de prototypes virtuels permet de tester des architectures où des choix matériels n'ont pas encore été faits. De plus, comme il n'y a pas de réalisation de prototypes physique couteux, il est possible de tester un plus grand nombre d'alternatives d'architecture sans induire de dépassement de budget.

La construction des prototypes virtuels passe par une étape de modélisation des systèmes constituants l'observatoire sous-marin. Ces modèles servent de base pour la création du prototype virtuel de l'observatoire sous-marin. Cependant, un observatoire sous-marin implique des experts issus de domaines différents. Les modèles pour chacun de ces domaines font intervenir des concepts différents. Il existe donc une variété de langages de modélisation utilisés pour la conception d'un observatoire sous-marin comme MeDON.

Pour permettre de concevoir les prototypes virtuels à partir de modèles différents, il faut être capable d'en extraire les informations d'intérêt malgré les différences de formalismes de modélisation. De plus, l'extraction de l'information des modèles ne doit pas nécessiter pour un utilisateur de connaître les détails des méta-modèles de tous les langages de modélisation.

Différents outils d'exécution sont utilisés tout au long du cycle de conception de l'observatoire. Dans les premières étapes, le niveau de détail des modèles et le besoin en exécution sont de haut niveau. Plus tard dans le cycle, l'exécution devra être plus fine pour permettre de faire des calculs de dimensionnement. Les outils d'exécution sont ainsi amenés à évoluer au cours du temps. Il est nécessaire d'avoir la capacité d'accompagner cette évolution.

En plus de l'exécution, des outils d'analyse sur les modèles peuvent être utilisés pour vérifier leur qualité. Il est donc nécessaire d'être capable d'adapter les modèles à des outils de natures différentes.

L'exemple de l'observatoire sous-marin MeDON implique :

— deux formalismes de modélisation SysML et eFFBD utilisés pour modéliser des sous-systèmes de MeDON.

— plusieurs outils d'exécution des modèles : un simulateur d'architecture fonctionnelle réalisé en interne, un simulateur réseau (OMNeT++) et un simulateur de composants basé événements discrets (ADEVS) ces deux derniers disponibles sur étagère.

— en plus de l'exécution des modèles, le besoin de faciliter la documentation du système par

l'utilisation des modèles.

La Figure 5.6 illustre la situation problématique que les points précédents provoquent.

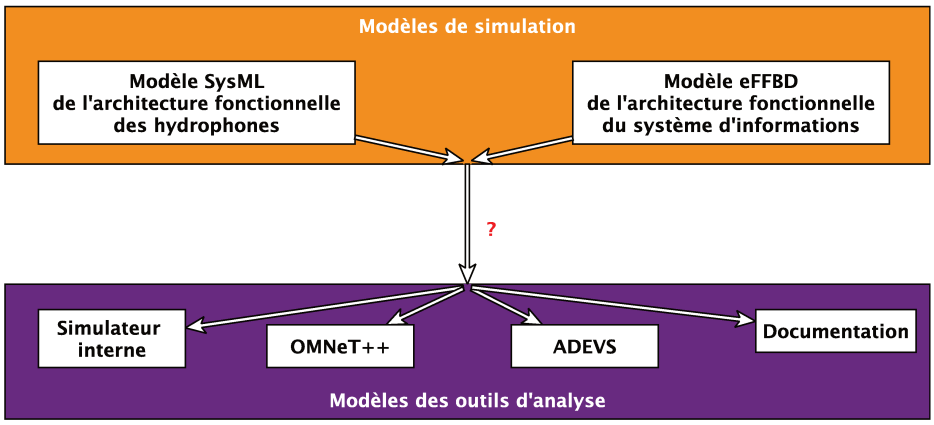


FIGURE 5.6. – Situation problématique du projet MeDON

Tout d'abord, l'observatoire n'est pas modélisé dans un seul langage de modélisation. Plusieurs modèles exprimés dans les langages SysML et eFFBD sont utilisés. Il faut donc avoir la capacité de lire ces différents modèles et d'en extraire de l'information pour pouvoir créer les prototypes virtuels (modèles de simulation). Ce problème nécessite d'utiliser des approches d'unification ou de fédération de modèles.

Ensuite, les prototypes virtuels (modèles de simulation) sont de nature différentes. La construction des modèles de simulation nécessite d'adopter des points de vue différents sur les modèles système qui servent d'entrée au processus de construction des prototypes virtuels. Par conséquent, l'approche d'unification ou de fédération utilisée dans le cadre de MeDON doit être guidée par la modélisation d'un point de vue.

Enfin, nous ne devons pas oublier que nous travaillons dans un contexte système de systèmes. L'observatoire MeDON est prévu pour avoir une durée de vie de plusieurs années et est amené à évoluer. Cette évolution se traduit, pour nous, par l'utilisation de nouveaux langages de modélisation, de nouveaux outils de simulation et une évolution de la structure des langages de modélisation et des outils de simulation déjà utilisés.

Pour résoudre ces problèmes, nous utilisons notre environnement Role4All tel qu'illustré Figure 5.7.

Pour pouvoir lire les modèles SysML et eFFBD, nous avons réalisé des importateurs dédiés à ces langages. La section 5.3 illustre l'écriture de ces importateurs à l'aide de parsers combinateurs et leur intégration au sein de l'environnement Role4All grâce à des mécanismes d'introspection. Cette section illustre les capacités de Role4All de lire des modèles différents et d'accepter de nouveaux langages de modélisation au fil des évolutions de MeDON.

Pour créer les prototypes virtuels à partir des modèles SysML et eFFBD, nous avons choisi une approche par unification. En effet, nous faisons le choix d'utiliser uniquement les modèles système comme référence sans pouvoir les modifier. Il n'y a donc pas besoin d'établir de relations bi-directionnelle entre les modèles système et les modèles de simulation qui sont re-générés à chaque fois. L'unification des modèles SysML et eFFBD étant guidée par le point de vue de l'outil de simulation, nous utilisons les rôles pour modéliser ce point de vue. La section 5.4 illustre la création du modèle de rôle à partir de l'analyse d'un outil de simulation et la création

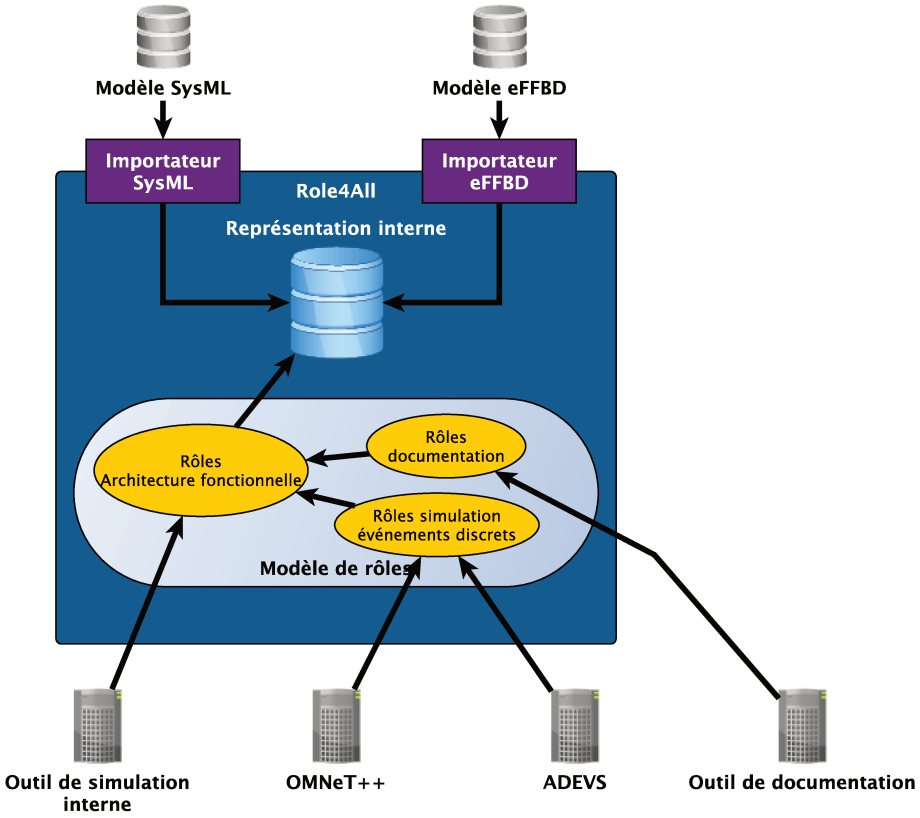


FIGURE 5.7. – Utilisation de Role4All pour l'observatoire MeDON

de la simulation proprement dite.

Un simulateur interne et le simulateur de réseau OMNeT++ sont utilisés. La section 5.5 illustre l'utilisation de plusieurs modèles de rôles pour modéliser différents points de vue. La propriété des rôles qui jouent des rôles est utilisée pour permettre d'associer des points de vue sur les modèles SysML et eFFBD pour les adapter à la simulation par événements discrets qui est le point de vue de l'outil OMNeT++.

Dans le cadre du projet MeDON, deux simulateurs par événements discrets OMNeT++ et ADEVS sont utilisés. La section 5.6 illustre l'utilisation des adaptateurs définis dans Role4All entre les rôles et les objets qui jouent des rôles. Les adaptateurs permettent d'avoir des modèles de rôles indépendants de l'utilisation qui en est faite. Il est ainsi possible d'utiliser le même modèle de rôles pour différents simulateurs qui ont le même point de vue sur les modèles mais une structure différente. Cette caractéristique permet de s'adapter aux évolutions des outils de simulation.

La durée de vie du projet MeDON implique que la définition des langages de modélisation va évoluer également. La section 5.7 illustre ces changements en illustrant la modularité des parsers combinateurs qui permettent de tenir compte de modifications dans la structure des langages de modélisation sans perdre la capacité de lire des modèles réalisés dans des versions du langage de modélisation qui ne contiennent pas ces modifications. Nous montrons également comment les aspects dynamiques des rôles permettent d'adapter le modèle de rôles à la nouvelle structure du langage de modélisation.

Enfin, les modèles SysML et eFFBD ne servent pas uniquement à créer des simulations mais servent également à la documentation du système. La section 5.8 illustre la propriété des objets qui peuvent jouer plusieurs rôles et qui peuvent donc être utilisés en fonction de plusieurs point de vues.

5.3. Import de modèles hétérogènes

Le choix d'un outil dédié aux activités de création de prototypes virtuels à partir de modèles système permet d'être technologiquement indépendant des outils de modélisation système. Ce choix nécessite cependant d'être capable d'importer des modèles définis par des méta-modèles différents. Pour ce faire, nous avons suivi quatre étapes :

1. définir la représentation interne à Role4All des méta-modèles des langages de modélisation (Sous-section 5.3.1),
2. analyser le format de sérialisation des modèles à importer (Sous-section 5.3.2),
3. écrire les parsers pour les formats de sérialisation (Sous-section 5.3.3),
4. intégrer les parsers à l'environnement Role4All (Sous-section 5.3.4).

5.3.1. Définition des représentations internes des méta-modèles

Dans le cadre du projet MeDON, nous avons des modèles définis dans les langages SysML et eFFBD. Les méta-classes définissant les méta-modèles de ces deux langages ont été ajoutées dans l'environnement Role4All sous la forme de classes Smalltalk.

Méta-modèle pour le langage SysML

SysML est un langage standardisé par l'Object Management Group [67]. Les éléments du méta-modèle SysML peuvent être groupés en neuf catégories en fonction des diagrammes qu'ils concernent:

— diagramme d'exigences,
— diagramme de définition de blocs,
— diagramme de bloc interne,
— diagramme paramétrique,
— diagramme de package,
— diagramme d'activité,
— diagramme de séquence,
— diagramme d'états,
— diagramme de cas d'utilisation.

Les modèles SysML utilisés dans le cadre de notre exemple de l'observatoire MeDON sont des diagrammes de définition de blocs et des diagrammes de blocs internes. Nous avons donc restreint la traduction du méta-modèle SysML en classes Smalltalk aux méta-classes qui concernent les diagrammes de définition de blocs et les diagrammes de blocs internes.

De plus, afin de pouvoir nous concentrer sur l'élaboration du modèle de rôles, nous avons volontairement simplifié la structure du méta-modèle SysML retranscrite. Tout d'abord, SysML repose sur l'utilisation des concepts du langages UML pour la représentation de structures composites. Le méta-modèle SysML définit de nombreuses abstractions pour permettre aux structures composites d'être génériques. Nous nous sommes concentrés sur des structures composites uniquement composées de blocs et de ports.

Une seconde différence concerne la notion d'instance de blocs nommée *part* dans le vocabulaire SysML. Dans un modèle SysML une part est définie au sein d'un diagramme de bloc interne et sert ainsi à représenter la structure interne d'un bloc. Une part est une instance de bloc et peut contenir d'autres part qui constituent la structure interne du bloc. Par conséquent, les parts à l'intérieur d'une autre part ne sont pas rattachées à la part englobante mais au bloc dont la part englobante est une instance. Afin de faciliter, dans un premier temps, l'élaboration du modèle de rôles, lorsqu'un bloc contient des sous-blocs, chacune de ses instances possèdera référence vers des instances des sous-blocs.

Les classes écrites pour le langage SysML concernent donc :

— la description des blocs qui constituent le système (diagramme de définition de blocs),
— la description de l'instanciation des blocs (diagramme de blocs internes).

La partie concernant la définition de blocs est présentée Figure 5.8.

La méta-classe SysmlNamedElement est une classe-mère pour toutes les méta-classes du méta-modèle du langage SysML. Elle implique que tous les éléments sont nommés. Cette méta-classe hérite de la méta-classe Player. Cette relation d'héritage implique que toutes instances des méta-classes du méta-modèle SysML sont capables de jouer des rôles.

La méta-classe SysmlBlock permet de représenter les blocs qui composent le système. La relation de composition subBlocks modélise les relations hiérarchiques entre blocs. La méta-classe SysmlBlock possède une relation de composition avec la méta-classe SysmlPort qui modélise la possession par les blocs de points d'entrée-sortie de données.

La méta-classe SysmlPort possède un attribut isBehavioral. Cet attribut permet de distinguer un port forçant l'implémentation d'un comportement par le bloc qui le possède d'un port qui sert de relai vers un port d'un sous-bloc.

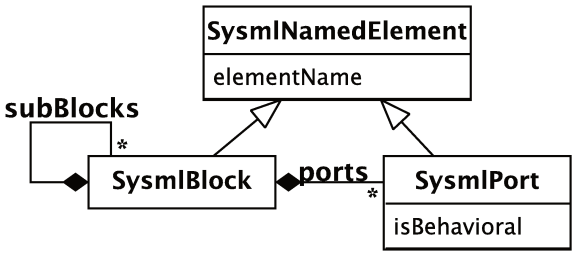


FIGURE 5.8. – Version adaptée du méta-modèle SysML concernant la définition d'un diagramme de blocs.

La description d'un déploiement des blocs précédemment modélisés a également été méta-modélisée. La Figure 5.9 présente la partie du méta-modèle défini pour la description du déploiement de blocs.

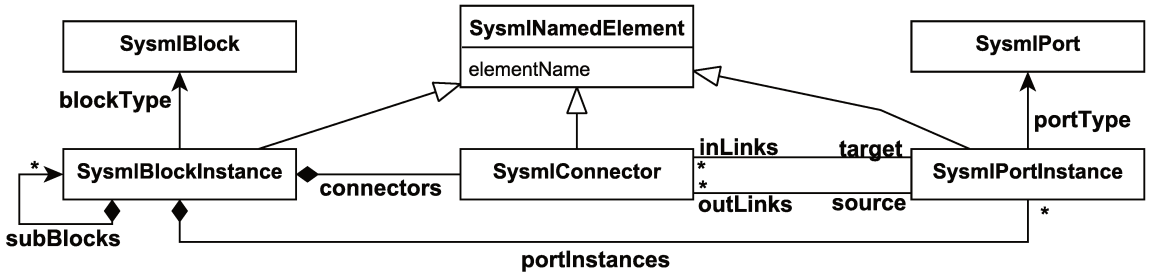


FIGURE 5.9. – Description de la version adaptée de la partie du méta-modèle du langage SysML décrivant les diagrammes de blocs internes.

Les méta-classes SysmlBlockInstance et SysmlPortInstance permettent de décrire un déploiement physique des blocs modélisés sous la forme de SysmlBlock et de SysmlPort. La méta-classe SysmlConnector permet de créer les liens entre les différentes instances de ports. Il est ainsi possible de définir les flux d'informations qui existent entre les différentes instances de blocs au travers des instances de ports. La méta-classe SysmlBlockInstance possède une référence vers la méta-classe SysmlBlock permettant d'identifier le type de l'instance. Il en est de même pour la méta-classe SysmlPortInstance qui possède une référence vers la méta-classe SysmlPort.

Ces différentes méta-classes sont implémentées sous la forme de classes Smalltalk. La classe SysmlNamedElement qui correspond à la méta-classe du même nom hérite de la classe Player. Cette relation d'héritage implique que toutes les instances des classes représentant les concepts du langage SysML peuvent jouer des rôles.

Méta-modèle pour le langage eFFBD

Le même travail de méta-modélisation a été effectué pour le langage eFFBD. Une version du méta-modèle eFFBD a été proposée par Nastov et al. [64]. Nous nous en sommes inspirés en le simplifiant pour obtenir le méta-modèle présenté Figure 5.10.

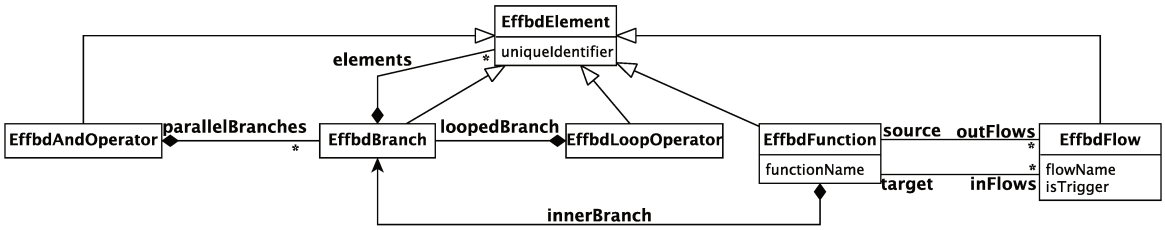


FIGURE 5.10. – Méta-modèle simplifié pour le langage eFFBD.

La méta-classe EffbdElement est la classe-mère pour tous les éléments constituants un modèle eFFBD. Elle hérite de la méta-classe Player.

La méta-classe EffbdBranch modélise l'enchaînement en séquence d'éléments de modélisation eFFBD.

La méta-classe EffbdAndOperator modélise l'opérateur de parallèlisation de branches. Il possède une collection de références vers toutes les branches qui sont parallélisées.

La méta-classe EffbdFunction modélise le concept de fonction au sein du langage eFFBD. Une fonction peut être décrite elle-même par un enchainement de fonctions. La méta-classe EffbdFunction possède une relation de composition avec la méta-classe EffbdBranch qui décrit le comportement de la fonction.

La méta-classe EffbdLoopOperator permet de décrire qu'une branche peut être exécutée de manière répétitive. La méta-classe EffbdLoopOperator possède une relation de composition avec la méta-classe EffbdBranch qui représente la branche qui sera bouclée.

La méta-classe EffbFlow permet de modéliser les flux d'informations échangés entre deux fonctions. Ces flux portent un nom (attribut flowName) et peuvent être déclencheurs d'une fonction (attribut isTrigger). Pour modéliser les fonctions émettrice et réceptrice du flux, la méta-classe EffbdFlow possède deux associations bidirectionnelles avec la méta-classe EffbdFunction.

Les méta-classes du langage eFFBD sont également implémentées sous la forme de classes Smalltalk. La classe EffbdElement hérite de la classe Player. Cet héritage permet aux instances des classes décrivant le langage eFFBD de jouer des rôles.

5.3.2. Analyse des formats de sérialisation des modèles

Afin d'illustrer la création d'importateurs modulaire et de démontrer l'utilisation des parsers combinateurs y compris sur des formats non standards, les modèles SysML et eFFBD ont été sérialisés dans un format textuel dédié pour le cas d'utilisation. Ce format de sérialisation est inspiré des langages IDL et JSON.

La première étape pour la création des importateurs de modèle est l'analyse du format de sérialisation. Le format utilisé répond au patron décrit Listing 5.1.

```
entite identifiant {
  proprietes entite
}
```

Listing 5.1 – Patron du format de sérialisation adopté pour les modèles SysML et eFFBD

Chaque élément du modèle est sérialisé sous la forme d'un bloc de code. Ce bloc est précédé par la dénomination du type de l'élément de modèle suivie par un identifiant. Le bloc contient un liste de toutes les propriétés de l'entité.

Le Listing 5.2 illustre l'utilisation du format de sérialisation sur une instance de bloc SysML. Le modèle SysML sérialisé complet utilisé est fourni en annexe B.

```
ComponentInstance componentInstance.3 {
  name: Detection1
  type: component.3
  ports: portInstance.3, portInstance.4
  subComponents:
  connectors:
}
```

Listing 5.2 – Exemple d'éléments de modèles SysML sérialisés.

La description de l'instance de bloc SysML comprend un identifiant unique *componentInstance.3* suivi de la définition des valeurs de l'ensemble des attributs de l'instance de bloc. L'instance a pour nom *Detection1*. Le *type* de l'instance de bloc est le bloc dont l'identifiant est *Component.3*. L'attribut *ports* est décrit par une liste d'identifiants correspondants aux instances de ports liées à l'instance de bloc. Les attributs *subComponents* et *connectors* sont vides mais peuvent contenir une liste d'identifiants. A la fin de la phase d'analyse, chaque identifiant devra être remplacé par l'élément de modèle associé à cet identifiant.

Le Listing 5.3 illustre l'utilisation du format de sérialisation sur une instance de flux eFFBD. Le modèle eFFBD sérialisé complet utilisé est fourni en annexe C.

```
Flow flow.1 {
  name: dataEntry1
  source: function.1
  target: function.4
  isTrigger: true
}
```

Listing 5.3 – Flux eFFBD sérialisé

La description du flux eFFBD commence par la définition de l'identifiant unique du flux. L'attribut *name* se voit affecter une valeur tandis que les attributs *source* et *target* sont associés à des identifiants d'autres éléments de modèles. L'attribut *isTrigger* prend la valeur *true* qui devra être remplacée par son équivalent sous la forme d'un booléen.

5.3.3. Définition des parsers pour importer les modèles sérialisés

Les parsers pour les formats sérialisés des modèles SysML et eFFBD sont définis à l'aide du framework PetitParser. L'identification du patron de sérialisation permet de définir un patron pour l'écriture du parser pour le format de sérialisation.

La description d'une entité dans le format textuel implique que six parsers soient définis :

- Le parser pour le type de l'entité.
- Le parser pour l'identifiant (commun).
- Le parser pour le symbole définissant l'ouverture du bloc de description de l'entité (commun).
- Le parser pour la description de l'entité.

— Le parser pour le symbole définissant la fermeture du bloc de description de l'entité (commun).
— Le parser qui combine les cinq parsers précédents.

Les parsers marqués communs peuvent être utilisés pour tous les parsers des entités décrites dans le format de sérialisation.

Parser le format de sérialisation des modèles SysML

Les parsers sont définis au sein d'une classe héritant de la classe PPCompositeParser fournie par le framework PetitParser.

Dans le cas de l'exemple de l'instance de bloc SysML, les parsers associés sont définis dans la classe Text2Sysml. Le Listing 5.4 illustre la partie du parser dédiée au traitement des instances de blocs SysML. Pour des questions de présentation et de lisibilité du code, seule la propriété *nom* est analysée. Les autres propriétés sont omises mais le même raisonnement peut être appliqué pour les analyser.

```
1 Text2Sysml >> blockInstanceDesc
2   ^('name:' asParser trim,
3     (#word asParser star) trim flatten,
4    ) ==> [ :nodes | |comp|
5     comp := SysmlComponentInstance new elementName: (nodes second trim).
6   ].
```

Listing 5.4 – Parser pour la désérialisation de la description d'une instance de bloc SysML

Le parser *blockInstanceDesc* doit créer une instance de bloc SysML à partir des informations contenues dans la description. Le parser *blockInstanceDesc* est un parser obtenu par combinaison de tous les parsers pour les caractéristiques d'un élément de type BlockInstance. Ici, seul le parser pour la caractéristique *name* est présenté. Ligne 2, la chaîne de caractère *name:* est utilisée pour construire un parser. Celui-ci est combiné avec un parser qui analyse le nom de l'instance de bloc.

Le parser pour le nom doit analyser une chaîne de caractères. Il est constitué du parser pour un caractère alphanumérique *#word* utilisé plusieurs fois (opérateur *star*). Les différents caractères analysés sont groupés dans une chaîne de caractères (opérateur *flatten*) dont les espaces de début et de fin sont supprimés (opérateur *trim*). Lorsque le parser *blockInstanceDesc* est vérifié, une instance de la classe SysmlBlockInstance est créée. Les attributs de cette classe sont initialisés avec les valeurs renvoyées par les différents parsers qui constituent *blockInstanceDesc*. Ligne 5, l'attribut *elementName* de l'instance de SysmlBlockInstance est initialisé avec la valeur renvoyée par l'analyse de la caractéristique *name:*. L'instance créée et initialisée est retournée par le parser.

Le parser *blockInstanceDesc* est utilisé au sein du parser *blockInstanceParser* qui doit analyser la forme sérialisée d'une instance de bloc. Le Listing 5.5 présente le code de ce parser.

```
Text2Sysml >> blockInstanceParser
^(blockInstanceName, identifier, beginDescSymbol, blockInstanceDesc, endDescSymbol)
  ==> [:nodes |
    |bi|
    bi := nodes fourth uniqueId: (nodes second).
    result at: (nodes second) put: bi.
  ].
```

Listing 5.5 – Parser pour une instance de bloc sérialisée.

Le parser *blockInstanceParser* est obtenu par la combinaison en séquence des parser *blockInstanceName*, *identifier*, *beginDescSymbol*, *blockInstanceDesc* et *endDescSymbol*. Le parser *blockInstanceName* attend la chaîne de caractères *ComponentInstance*.

Le parser *identifier* attend une chaîne de caractères qui commence par une lettre suivie de plusieurs caractères alpha-numériques suivis par un point puis une séquence de chiffre. Un exemple d'identifiant valide est *ComponentInstance.1*.

Le parser *beginDescSymbol* (respectivement *endDescSymbol*) attend le caractère { (resp. }).

Lorsque le parser ainsi combiné est vérifié, l'identifiant de l'instance renvoyée par le parser *blockInstanceDesc* prend comme valeur le résultat du parser *identifier*. L'instance est également ajoutée à un dictionnaire nommé *result* qui contient toutes les instances issues de l'analyse d'un modèle sérialisé. La clé est l'identifiant associé à l'élément de modèle.

En plus du parser *blockInstanceDesc*, des parsers pour les autres entités constituants un modèle SysML sont définis au sein de la classe Text2Sysml en respectant la même structure. Nous avons défini les parsers :

blockParser pour la description des blocs SysML,

portParser pour la description des ports SysML,

portInstanceParser pour la description des instances de ports SysML,

connectorParser pour la description des connecteurs SysML.

Tous les parsers sont assemblés dans le parser composite défini au sein de la méthode *start* de la classe Text2Sysml comme illustré Listing 5.6.

```
Text2Sysml >> start
  ^(blockParser,
    (blockParser / portParser / blockInstanceParser / portInstanceParser / connectorParser) star)
     ==> [ :instance |
       self resolveDependencies .
       result values.
     ]
```

Listing 5.6 – Parser principal de la classe Text2Sysml

Le parser Text2Sysml est défini comme la séquence :

— d'un blockParser : ce premier bloc obligatoire représente le système,

— d'un choix ordonné entre blockParser, portParser, blockInstanceParser, portInstanceParser et connectorParser, ce choix peut être répété plusieurs fois.

Le Listing 5.6 montre que lorsque le parser est vérifié, la méthode *resolveDependencies* est appelée avant qu'un résultat ne soit renvoyé.

La méthode *resolveDependencies* permet de remplacer toutes les mentions des identifiants par les instances des entités correspondantes. La méthode *resolveDependencies* parcourt le dictionnaire des résultats de l'analyse (result). Pour chaque valeur du dictionnaire, la méthode

resolveDependencies: est appelée sur la valeur en passant le dictionnaire result en paramètre de l'appel. Le comportement supplémentaire qui permet de créer les liens entre les différentes instances ne fait pas partie du méta-modèle. Il ne doit pas apparaître dans les classes qui définissent le méta-modèle SysML. La méthode *resolveDependencies* est donc définie en tant que méthode d'extension sur toutes les classes Smalltalk du méta-modèle SysML. L'utilisation d'une méthode d'extension permet de ne pas modifier les classes qui définissent le méta-modèle SysML en séparant clairement la définition des éléments des activités d'analyse.

Le Listing 5.7 présente l'implémentation de cette méthode pour la classe SysmlBlockInstance.

```
1 resolveDependencies: aDictionary
2   self resolveComponentType: aDictionary.
3   self resolvePortInstances: aDictionary.
4   self resolveSubComponents: aDictionary.
5   self resolveConnectors: aDictionary.
```

Listing 5.7 – Implémentation de la méthode *resolveDependencies:* pour la classe associée aux instance de blocs SysML

Pour chaque attribut dont la valeur est un identifiant, un objet est recherché dans le dictionnaire *aDictionary* passé en paramètre de la méthode *resolveDependencies*. Le travail est fait de manière répétitive sur les attributs dont la valeur est une liste d'identifiants. La ligne 2 permet de créer la référence avec l'instance de SysmlBlock qui décrit le type de l'instance de bloc. La ligne 3 permet de créer les références vers les instances de ports qui sont affectés à l'instance de block. La ligne 4 permet de créer les références vers les instances de SysmlBlockInstance qui sont déployées dans l'instance de SysmlBlockInstance courante. La ligne 5 permet de créer les références vers les liens qui sont définis entre les sous-blocs de l'instance courante.

Parser le format de sérialisation des modèles eFFBD

Dans le cas du langage eFFBD, nous avons défini un parser sous la forme de la classe EffbdTextParser héritant de PPCompositeParser. Les parsers définis au sein de cette classe sont :

andDefinition pour la description de l'opérateur parallèle,

branchDefinition pour la description d'une branche,

flowDefinition pour la description d'un flux,

functionDefinition pour la description d'une fonction,

loopDefinition pour la description de l'opérateur boucle.

Le parser principal de la classe EffbdTextParser est présenté Listing 5.8.

```
1 EffbdTextParser >> start
2   ^branchDefinition, (branchDefinition / andDefinition / loopDefinition / functionDefinition / flowDefinition) plus
      ==> [ :instance |
3     self resolveDependencies .
4     result values.
5   ]
```

Listing 5.8 – Parser principal de la classe EffbdTextParser.

Le parser principal de la classe EffbdTextParser est défini comme la séquence :

— du parser *branchDefinition* qui décrit le système,

— d'un choix ordonné entre les parsers *branchDefinition*, *andDefinition*, *loopDefinition*, *functionDefinition* et *flowDefinition*, ce choix peut être répété plusieurs fois.

Lorsque le parser est vérifié, la méthode *resolveDependencies* est appelée. Cette méthode permet de créer les liens entre les différents éléments qui composent le modèle eFFBD. L'appel au parser renvoie une collection de toutes les instances d'éléments de modèle eFFBD créées.

L'utilisation de parsers permet d'analyser les formats textuels qu'ils soient propriétaires ou standards (comme par exemple JSON [6]). Il est donc possible d'analyser tous modèles sérialisés. De plus, l'utilisation de parsers combinateurs permet d'écrire des parsers modulaires. En effet, comme dans le cas de nos parsers pour les modèles SysML et eFFBD sérialisés, les parsers réalisés sont constitués de parsers dédié à un schéma textuel particulier et assemblés grâce à des opérateurs dédiés. Ainsi, en cas de modification du schéma textuel, seul le parser concerné est impacté ce qui facilite la maintenance du parser du format de sérialisation. Les parsers combinateurs permettent également de réutiliser les parsers déjà écrit et de les intégrer dans un ensemble de parser plus générique.

5.3.4. Intégration des parsers à l'environnement Role4All

Les parsers permettent d'importer les modèles au sein de l'environnement Role4All en instanciant les classes Smalltalk qui correspondent aux méta-modèles des langages de modélisation. Les parsers sont à la base des importateurs de modèles qui sont les points d'entrées des modèles. Les parsers peuvent toutefois être définis de manière indépendante de l'environnement Role4All. Ils doivent donc y être intégrés. Pour cela, les classes SysmlImporter et EffbImporter sont définies. Elles héritent toutes deux de la classe ModelImporter définie dans Role4All. Cet héritage est illustré Figure 5.11.

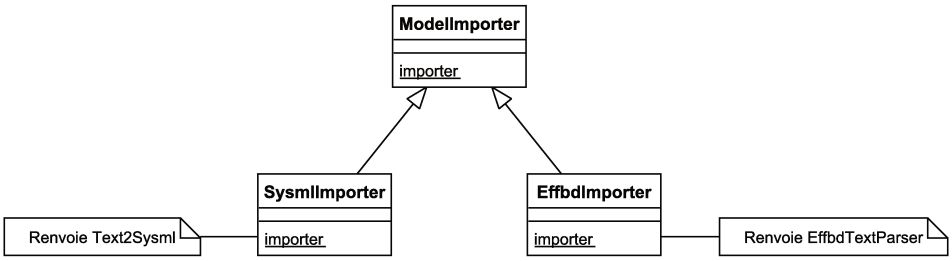


FIGURE 5.11. – Héritage entre les classes SysmlImporter et EffbdImporter et ModelImporter.

L'héritage implique que les classes SysmlImporter et EffbdImporter héritent de la méthode de classe *importer* définie dans ModelImporter. Les redéfinitions de cette méthode dans les deux sous-classes sont illustrées Listing 5.9.

```
1 SysmlImporter class >> importer
2   ^Text2Sysml asParser.
3
4 EffbdImporter class >> importer
5   ^EffbdTextParser asParser.
```

Listing 5.9 – Implémentation de la méthode de classe importer dans les classes SysmlImporter et EffbdImporter

Le corps des deux méthodes fait référence aux parsers Text2Sysml et EffbdTextParser précédemment définis.

L'utilisation des parsers combinateurs nous permet d'étendre la définition d'un importateur en ajoutant de nouveaux parsers à l'aide des opérateurs de combinaisons. Il est ainsi aisé de complexifier la définition d'un importateur.

De plus, seule la méthode de classe importer doit être redéfinie. En effet, l'utilisation de l'introspection par Role4All pour créer l'importateur global consiste à chercher les sous-classes de ModelImporter et à leur envoyer le message *importer* sur celles-ci. Par conséquent, un changement de la définition d'un importateur est fait de manière transparente pour Role4All.

Grâce à l'utilisation de parsers combinateurs nous sommes capables d'importer les modèles SysML et eFFBD de l'observatoire sous-marin au sein de l'environnement Role4All. Nous avons donc la capacité de manipuler des modèles dans des langages différents. La classe ModelImporter définie dans l'environnement Role4All a permis d'intégrer les parsers pour les modèles directement et facilement dans l'environnement. Ce mécanisme facilite l'intégration de nouveaux langages de modélisation. De plus, les parsers sont indépendants des outils de modélisation, dans notre cas IBM Rhapsody et MDWorkbench, qui ont servi à la création des modèles. Ainsi, même si ces outils ne sont plus utilisés, nous pouvons continuer à lire les modèles sérialisés réalisés avec ces outils. L'information contenue dans les modèles n'est pas perdue lors de l'arrêt de l'utilisation des outils de modélisation.

L'utilisation des parsers combinateurs et de l'introspection a permis de rendre l'architecture de Role4All indépendante des outils de modélisation utilisés pour créer les modèles système. Le cœur de Role4All n'est pas impacté par l'ajout de nouveaux langages de modélisation ou un changement de format de sérialisation. Par contre, Role4All peut être étendu facilement pour prendre en compte de nouveaux langages de modélisation ou un changement de format de sérialisation.

5.4. Manipulation de modèles hétérogènes

Des modèles hétérogènes (modèles SysML et eFFBD) sont utilisés pour concevoir l'architecture fonctionnelle de l'observatoire MeDON. Ces différents modèles doivent être utilisés pour créer des prototypes virtuels, c'est-à-dire des modèles de simulation, malgré les différences qui existent entre les langages de modélisation utilisés pour créer les modèles système. Grâce aux importateurs modulaires et aux parsers combinateurs, décrit dans la section précédente, nous sommes capables de lire les modèles dans les différents formalismes.

Cependant, nous devons encore être capables de créer une vision unifiée sur les différents modèles et donc faire de l'unification de modèles en fonction du point de vue de l'outil de simulation qui doit être utilisé pour le prototypage virtuel. Pour réaliser cette unification, nous définissons des modèles de rôles à l'aide de Role4All et de son DSL R2DL. Pour cela nous devons :

1. analyser l'outil de simulation utilisé pour le prototypage virtuel :
   a) lire l'API de l'outil,
   b) faire un modèle de l'API
2. créer un modèle de rôle qui correspond à l'outil de simulation et qui est associé aux éléments de modèles à unifier :

a) définir les types de rôles en fonction des concepts d'intérêts identifiés dans l'API de l'outil de simulation,

b) définir les associations entre les types de rôles et les éléments de modèles système,

c) définir les adaptations des éléments de modèles aux rôles qui leurs sont associés.

Dans cette section, nous illustrons cette démarche en utilisant un outil capable d'exécuter des modèles d'architecture fonctionnelle que nous avons développé. Cependant, les modèles utilisés par cet outil ne sont ni des modèles SysML ni des modèles eFFBD. Ce sont des modèles exprimés dans un langage dédié et propre à l'outil. Ainsi, nous devons :

1. analyser la structure des modèles d'architecture fonctionnelle utilisés par l'outil ainsi que le fonctionnement du moteur d'exécution (Sous-section 5.4.1),
2. définir les types de rôles à utiliser en fonction de l'analyse de la structure des modèles d'architecture fonctionnelle précédemment réalisée (Sous-section 5.4.2),
3. de définir les associations entre les rôles et les éléments de modèles (Sous-section 5.4.3),
4. de définir les adaptateurs entre les rôles et les éléments de modèles pour permettre l'interprétation des éléments de modèles par le moteur d'exécution de l'outil (Sous-section 5.4.4).
5. de définir un contexte pour faciliter la réutilisation du modèle de rôles (Sous-section 5.4.5).

5.4.1. Analyse de l'outillage d'exécution de modèles

Parmi les processus techniques de l'ingénierie système, l'architecture fonctionnelle s'intéresse à ce que le système doit faire pour répondre aux besoins des parties prenantes du système. L'architecture fonctionnelle est indépendante d'une solution technique. Par conséquent, l'architecture fonctionnelle permet de faire comprendre aux différents experts ce que le système doit faire sans avoir besoin d'être expert dans tous les domaines concernés par la conception du système.

Dans le cadre du projet MeDON, l'architecture fonctionnelle permet de comprendre les services qui sont rendus par les différents sous-systèmes. Pour renforcer cette compréhension, nous voulons réutiliser un outillage pour exécuter l'architecture fonctionnelle que nous avions déjà développé. Cet outillage permet d'observer le fonctionnement du système et de comprendre les enchaînements de fonctions. Cependant, l'outillage possède son propre langage de modélisation d'architecture fonctionnelle.

Afin de ne pas avoir à refaire les modèles réalisés dans des langages de modélisation standards, nous souhaitons connecter directement notre outillage d'exécution avec les modèles systèmes. Nous avons donc étudié la structure de notre outillage.

L'outillage peut être décomposé en deux parties que nous allons analyser successivement :

— un méta-modèle qui définit un langage dédié permettant de décrire des architectures fonctionnelles [3],

— un moteur d'exécution basé acteurs interprètant les modèles réalisés avec le méta-modèle précédent [4].

3. Ce méta-modèle a été décrit dans [82] publié dans les proceedings du workshop de la conférence Complex System Design & Management 2014 [10].

4. Ce moteur d'exécution a été décrit dans [81] publié dans les proceedings de l'International Workshop on Smalltalk Technologies.

Nous souhaitons pouvoir simuler les modèles SysML et eFFBD de l'observatoire MeDON en nous concentrant sur l'architecture fonctionnelle. L'analyse du méta-modèle de l'outil permet donc de définir les types de rôles qui nous intéressent. Comme le moteur d'exécution interprète les modèles réalisés avec le méta-modèle analysé, l'analyse du moteur d'exécution permettra de définir les adaptateurs nécessaires à l'adaptation des éléments de modèles SysML et eFFBD aux rôles.

Méta-modèle de description d'architecture fonctionnelle

Le méta-modèle présenté Figure 5.12 permet de décrire des architectures fonctionnelles.

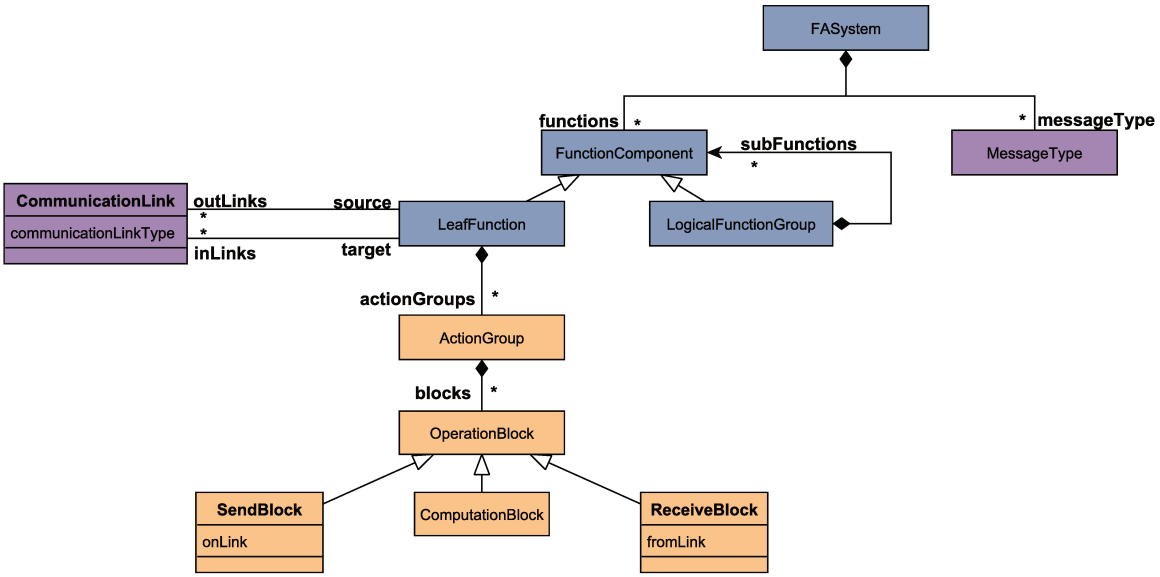


FIGURE 5.12. – Méta-modèle pour la description d'architectures fonctionnelles.

Les méta-classes représentées en bleu modélisent les caractéristiques structurelles de l'architecture fonctionnelle. Les méta-classes représentées en mauve modélisent l'aspect communication. Les méta-classes représentées en orange modélisent le comportement des fonctions.

D'un point de vue structurel, l'architecture fonctionnelle est vue comme une arborescence de fonctions. Les fonctions de plus bas niveau sont modélisées par la méta-classe LeafFunction. Les fonctions constituées d'autres fonctions sont modélisées par la méta-classe LogicalFunctionsGroup. Un patron de conception Composite [3] fait le lien entre les méta-classes LeafFunction et LogicalFunctionsGroup par l'intermédiaire de la méta-classe FunctionComponent.

D'un point de vue communication, la méta-classe MessageType permet de décrire les messages qui sont échangés par les fonctions. Dans le cadre du projet MeDON, nous travaillons avec des capteurs dont la conception nécessite de tenir compte de la synchronisation entre fonctions. Nous souhaitons pouvoir modéliser des synchronisations basées rendez-vous ou non. Dans un système complexe, il n'existe pas qu'un unique schéma de synchronisation entre les fonctions du système. L'exécution d'une fonction peut requérir un accusé de réception pour les données qui sont produites. Une autre fonction peut diffuser en continu ses données. Le méta-modèle pour définir une architecture fonctionnelle doit donc permettre de modéliser différents comportements de synchronisation. Nous avons donc réifié le concept de lien de communication sous la

forme de la méta-classe CommunicationLink. Cette méta-classe possède un attribut spécifiant le comportement basé rendez-vous ou asynchrone de la communication.

D'un point de vue comportemental, seules les fonctions de plus bas niveau dans l'arborescence possèdent un comportement. Le comportement des fonctions est décrit par des séquences d'opérations de lecture sur une entrée (méta-classe ReceiveBlock), de calculs sur ces entrées (méta-classe ComputationBlock) et d'écriture sur les sorties (SendBlock). La méta-classe ReceiveBlock (respectivement SendBlock) identifie le lien sur lequel lire (resp. écrire) des informations par l'intermédiaire de l'attribut *fromLink* (resp. *onLink*).

A un instant donné, toutes les entrées n'ont pas besoin d'être lues tout comme une écriture n'est pas forcément nécessaire sur toutes les sorties. Ces actions sont regroupées au sein d'une structure de contrôle définies par la méta-classe ActionGroup qui permet de modéliser des modes de fonctionnement des fonctions. De plus, la notion d'ActionGroup permet de séquencer les opérations de lectures/calculs/écriture.

Modèle du moteur d'exécution

Le moteur d'exécution interprète les modèles réalisés avec le méta-modèle précédent. Le moteur d'exécution est séparé du méta-modèle servant à définir les architectures fonctionnelles pour ne pas imposer aux modèles réalisés avec le méta-modèle de posséder des informations concernant les aspects exécution. Le moteur d'exécution est basé sur le modèle acteur défini par Agha [2]. Le modèle acteur définit des entités (les acteurs) qui s'exécutent en parallèle et qui ne possèdent pas d'état commun. Dans le cadre de l'architecture fonctionnelle, les fonctions sont des entités indépendantes qui peuvent s'exécuter en parallèle. Ainsi, le modèle acteur semble pertinent pour exécuter une architecture fonctionnelle. La Figure 5.13 représente le modèle issu de l'analyse du moteur d'exécution.

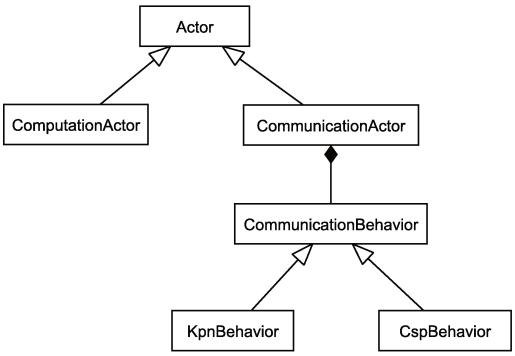


FIGURE 5.13. – Architecture du moteur d'exécution basé acteurs.

La classe Actor fournit une implémentation en Smalltalk des acteurs selon le modèle de Agha [2]. Dans ce modèle, les acteurs sont des entités qui s'exécutent en parallèle sans partager d'état. Les acteurs interagissent entre eux par l'intermédiaire de messages. Ces messages sont stockés au sein d'une file de messages. Le traitement des messages se fait dans l'ordre de leur réception. Ce modèle ne définit pas de règles d'ordonnancement entre les acteurs. Il n'est donc pas possible d'émettre une hypothèse sur l'ordre d'exécution des acteurs et par voie de conséquence sur l'ordre de réception des messages par les différents acteurs.

Les liens de communication dans l'architecture fonctionnelle possèdent un comportement

propre et déterminé. L'absence de contrôle sur l'ordre d'exécution des acteurs et le besoin d'être capable de modifier la nature des communications, nous ont amenés à créer un acteur qui implémente le comportement des échanges d'informations entre les acteurs métiers.

La classe Actor est étendue par les classes ComputationActor et CommunicationActor. La classe ComputationActor fournit un cadre pour la description des acteurs métiers. La création d'un nouvel acteur métier se fait en sous-classant la classe ComputationActor. La classe CommunicationActor décrit les acteurs implémentant le comportement des communications. Un CommunicationActor possède une référence vers la classe CommunicationBehavior. Cette dernière est la classe-mère de tous les comportements de communication implémenté. Pour créer de nouveaux comportements de communication, il faut étendre la classe CommunicationBehavior. Les comportements sont programmés sous la forme de machines à états. Nous fournissons une réalisation informatique d'une communication basée rendez-vous d'après le modèle de calcul CSP [42] et une réalisation informatique d'une communication asynchrone d'après le modèle de calcul KPN [50, 51].

Une synchronisation asynchrone ne bloque pas la fonction émettrice des données mais ne lui permet pas de s'assurer que les données ont été bien reçues par la fonction réceptrice. Un synchronisation par rendez-vous donne cette assurance mais bloque la fonction émettrice tant que la réceptrice n'est pas prête à recevoir des données. Il faut donc faire un compromis entre temps de blocage d'une fonction (voire d'une chaîne de fonctions) et fiabilité d'une transmission de données. Avoir la capacité d'exécuter différents schémas de synchronisation permet d'analyser l'impact en terme de délai temporel induit par un choix de schéma de synchronisation. Il est donc possible de comparer plusieurs choix de schémas de synchronisation et choisir le meilleur compromis.

5.4.2. Définition des types de rôles

Nous souhaitons pouvoir connecter directement le moteur d'exécution de l'outillage précédent avec les modèles SysML et eFFBD utilisés pour modéliser les sous-systèmes de l'observatoire MeDON. Pour cela, nous devons définir un modèle de rôles pour créer l'interface entre le moteur d'exécution et les modèles système. Nous définissons tout d'abord les types des rôles qui composeront le modèle de rôles.

Pour définir les types de rôles dont nous avons besoin, nous avons sélectionné les concepts définis au sein du méta-modèle d'architecture fonctionnelle de l'outil d'exécution que nous avons analysé. Cette sélection s'est basée sur le point de vue que nous souhaitons adopter sur les modèles de l'observatoire MeDON, c'est-à-dire l'aspect structurel de l'architecture fonctionnelle. En effet, nous souhaitons pouvoir travailler sur le découpage en fonctions de l'observatoire MeDON indépendamment du comportement précis des fonctions, dans un premier temps. Un comportement générique nous convient dans un premier temps.

Nous nous sommes donc concentrés sur la définition structurelle de l'architecture fonctionnelle présente dans le méta-modèle de l'outil d'exécution pour définir les rôles. Nous avons défini les rôles de groupes logiques fonctionnels, de fonctions feuilles et de lien de communication. Nous devons également ajouter une direction aux liens de communication entre les composants. En effet, le moteur d'exécution n'utilise que des communications unidirectionnelles. Nous avons ajouté les rôles de source et de cible d'une communication pour pouvoir modéliser la direction des communications dans notre modèle de rôle. Le Listing 5.10 illustre la création des rôles permettant de décrire une architecture fonctionnelle à l'aide du DSL interne que nous avons implémenté.

```
1 Role named: 'LeafFunctionRole' classifiedIn: 'JPS−Role−FAActors'.
2 Role named: 'LogicalFunctionsGroupRole' classifiedIn: 'JPS−Role−FAActors'.
3 Role named: 'CommunicationLinkRole' classifiedIn: 'JPS−Role−FAActors'.
4 Role named: 'SourceRole' classifiedIn: 'JPS−Role−FAActors'.
5 Role named: 'TargetRole' classifiedIn: 'JPS−Role−FAActors'.
```

Listing 5.10 – Définition des rôles pour décrire la structure d'une architecture fonctionnelle

La Figure 5.14 présente les rôles définis pour décrire l'architecture fonctionnelle une fois intégrés à l'outillage.

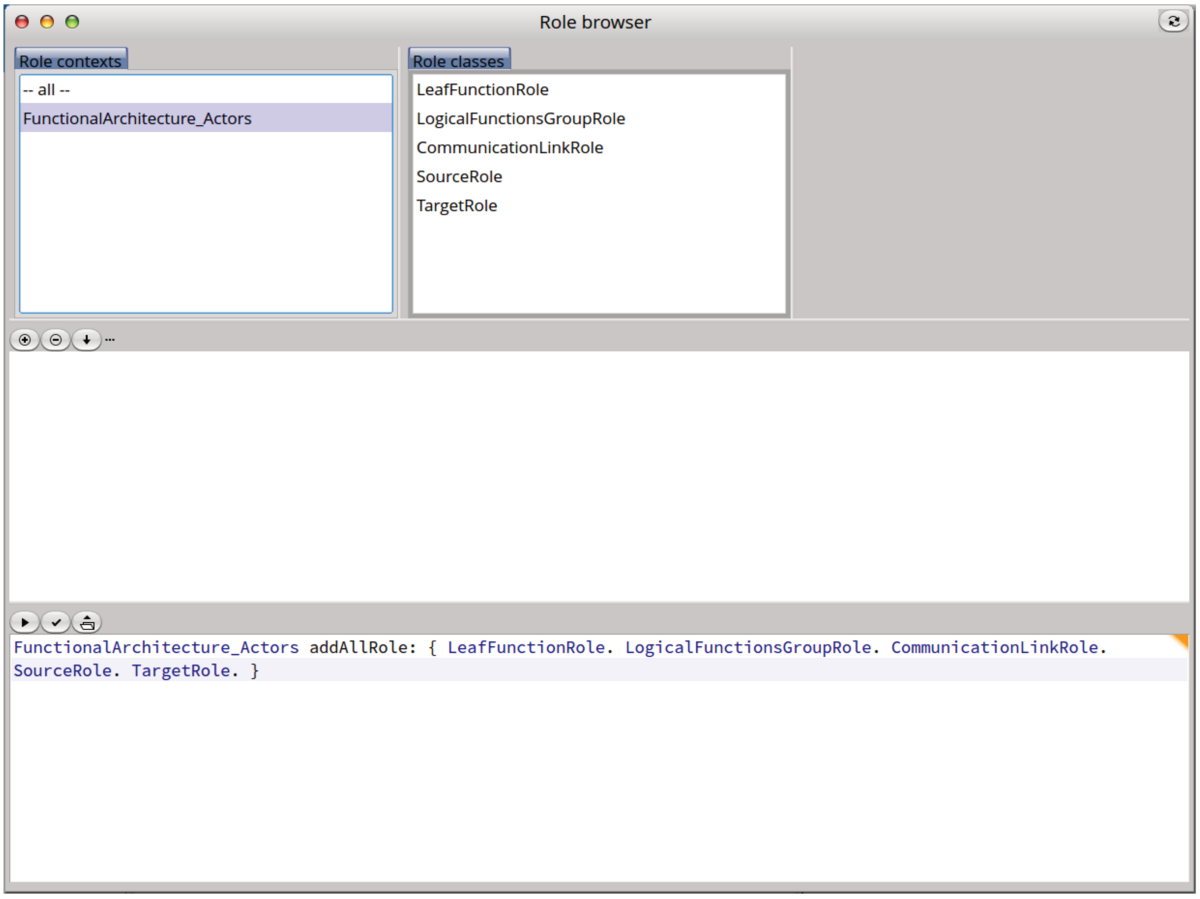


FIGURE 5.14. – Définition des rôles orientés architecture fonctionnelle et association avec un contexte au sein de notre outillage.

Les rôles possèdent également des propriétés. L'utilisation de l'outil d'exécution impose que chaque rôle soit capable de répondre aux messages *isLeafFunction*, *isLogicalFunctionsGroup* et *isCommunicationLink*. Il s'agit d'une contrainte imposée par le moteur d'exécution, les rôles doivent donc s'y plier. Pour ajouter ces propriétés, chaque rôle se voit associer les attributs correspondant tel qu'illustré dans le Listing 5.11.

```
1 LeafFunctionRole addAttributeNamed: 'isLeafFunction'.
2 LeafFunctionRole addAttributeNamed: 'isLogicalFunctionsGroup'.
3 LeafFunctionRole addAttributeNamed: 'isCommunicationLink'.
```

Listing 5.11 – Ajout des propriétés au rôle LeafFunctionRole

Le moteur d'exécution doit avoir accès au nom de la fonction ainsi qu'aux actions implémentées par la fonction. Ces propriétés sont déléguées par le rôle LeafFunctionRole. Le nom sera obtenu via la propriété *functionName* tandis que les actions seront obtenues via la propriété *actionGroups*. Ces propriétés étant déléguées, la méthode addDelegateAttribute est appelée sur le rôle LeafFunctionRole pour créer, dans la classe associée, les méthodes de délégation nécessaires.

```
1 LeafFunctionRole addDelegateMethod: 'functionName'.
2 LeafFunctionRole addDelegateMethod: 'actionGroups'.
```

Listing 5.12 – Ajout des propriétés de délégation au sein du rôle LeafFunctionRole

La définition de types de rôles repose sur l'analyse de l'outil de simulation qui doit être utilisé pour exécuter les modèles. Cet analyse repose sur des techniques de modélisation et de rétro-ingénierie existantes. Par conséquent, la définition des types de rôles ne nécessite pas de compétences nouvelles. De plus, l'utilisation du DSL R2DL permet de masquer la complexité des mécanismes du langage Smalltalk. Le modeleur de rôles peut se concentrer sur la description des types de rôles plutôt que sur leur réalisation dans l'environnement Smalltalk.

5.4.3. Définition des associations des rôles avec les éléments de modèles

L'étape suivante du processus de définition d'un modèle de rôles est de définir l'association entre les rôles (instances d'une classe qui définit un type de rôle) et les éléments de modèles.

Les fonctions de l'architecture fonctionnelle modélisée ayant un comportement sont les fonctions feuilles. Nous avons donc fait le choix de ne pas travailler dans un premier temps avec les groupes logiques de fonctions car ce sont les fonctions feuilles qui doivent être exécutées. Pour traiter le langage SysML, nous associons le rôle de LeafFunctionRole à toutes les instances de blocs qui ne possèdent pas de sous-bloc. En effet, des instances de blocs sans sous-bloc sont structurellement identiques à des fonctions feuilles et peuvent donc être interprétées de la même façon.

Nous associons le rôle de SourceRole à tous les ports SysML qui possèdent des liens sortant et le rôle de TargetRole à ceux qui possèdent des liens entrants à la condition que le port soit de type *behavioral*. En effet, seul les ports qui sont de type *behavioral* envoient ou reçoivent effectivement de l'information, les autres ne sont que des relais.

Le rôle CommunicationLinkRole n'est pas lié à un élément de modèle particulier, il agrège d'une instance du rôle SourceRole et d'une instance du rôle TargetRole.

Pour traiter le langage eFFBD, nous associons le rôle de LeafFunctionRole aux fonctions eFFBD. En effet, la structure du modèle est tel que les fonctions ne sont pas raffinées. Les fonctions ne contiennent pas de sous-fonctions. Dans le cas contraire, le test aurait dû être complété par un test sur la nullité de la référence de la fonction vers la classe EffbdBranch.

Nous associons le rôle SourceRole aux fonctions eFFBD qui possèdent des flux sortants et le rôle TargetRole à celles qui possèdent des flux entrants. Les flux eFFBD sont associés au rôle CommunicationLinkRole. De plus, les instances de ce rôle agrègent une instance de SourceRole et une instance de TargetRole.

Le tableau 5.1 résume les associations de rôles et d'éléments de modèles.

TABLE 5.1. – Tableau des associations entre rôle et élément de modèle.

Rôle	Elément de modèle	Condition
LeafFunctionRole	SysmlBlockInstance	Ne possède pas de sous-part.
SourceRole	SysmlPortInstance	Le port est de type behavioral et possède des liens en sortie.
TargetRole	SysmlPortInstance	Le port est de type behavioral et possède des liens en entrée.
LeafFunctionRole	EffbdFunction	
CommunicationLinkRole	EffbdFlow	
SourceRole	EffbdFunction	Possède des flux sortants.
TargetRole	EffbdFunction	Possède des flux entrants.

Les règles d'association définies dans le tableau 5.1 sont implémentées dans notre DSL. Le Listing 5.13 illustre l'association entre le rôle LeafFunctionRole et les instances de SysmlBlockInstance.

```
1 SysmlBlockInstance plays: LeafFunctionRole underCondition: [:p | p hasSubBlocks not].
```

Listing 5.13 – Implémentation de la règle d'association entre le rôle LeafFunctionRole et les éléments de modèles de type SysmlPart

Nous sommes dans une étape de mise au point du modèle de rôles. Nous avons donc connaissance de la structure des méta-modèles des langages manipulés. Ici, nous connaissons l'existence de la méthode *hasSubBlocks* de la classe SysmlBlockInstance. Nous pouvons utiliser cette méthode pour définir la condition d'association des rôles de type LeafFunctionRole avec les instances de SysmlBlockInstance. Cette méthode permet de tester l'existence de sous-blocs. La méthode *not*, définie pour les objets booléen, permet d'inverser le résultat renvoyé par la méthode *hasSubBlocks*. Par conséquent, des rôles du type LeafFunctionRole seront uniquement associés à des instances de SysmlBlockInstance qui ne possèdent pas de sous-bloc.

A l'issue de cette étape, nous avons obtenu un modèle de rôles constitué d'instances des types de rôles précédemment définis. Ces instances sont associées avec des éléments de modèles exprimés dans des langages de modélisation différents. Nous avions mis l'accent sur l'indépendance des rôles par rapport aux types des objets définis dans les langages de modélisation. Cette indépendance permet aux modeleurs de rôles d'utiliser des rôles du même type sur des objets de langages de modélisation différents sans avoir à adapter le type de rôle aux langages de modélisation. Le modeleur de rôles peut donc se concentrer sur le point de vue (modélisé par les rôles) qu'il souhaite associer aux éléments de modèles indépendamment du langage de modélisation et ainsi permettre l'unification des modèles.

5.4.4. Définition des adaptateurs

Une fois les associations entre rôles et éléments de modèles créées, il est possible de définir les adaptateurs dynamiques qui doivent être associés à ces relations. Le tableau 5.2 décrit l'affectation des adaptateurs dynamiques sur les relations entre rôles et éléments de modèles. Ce tableau permet de spécifier les adaptateurs qui doivent être définis ou réutilisés pour adapter les éléments de modèles aux rôles.

TABLE 5.2. – Règles d'association d'adaptateurs dynamiques sur les relations entre rôles et éléments de modèles

Rôle	Elément de modèle	Type de l'adaptateur
LeafFunctionRole	SysmlBlockInstance	InterpretCI
SourceRole	SysmlPortInstance	SourcePort
TargetRole	SysmlPortInstance	TargetPort
LeafFunctionRole	EffbdFunction	InterpretEffbdFunction
SourceRole	EffbdFunction	InterpretFuncAsSource
TargetRole	EffbdFunction	InterpretFuncAsTarget
CommunicationLinkRole	EffbdFlow	InterpretFlowAsCL

L'association d'un adaptateur dynamique sur une relation entre un rôle et un élément de modèle se fait selon les règles définies dans le tableau 5.2 à l'aide de notre DSL. Le Listing 5.14 illustre l'association d'instances de la classe InterpretCI sur les relations entre les rôles LeafFunctionRole et les instances de la classe SysmlBlockInstance.

```
1 LeafFunctionRole linkedToKind: SysmlPart withAdapter: 'InterpretCI'.
```

Listing 5.14 – Implémentation de l'association d'une instance de l'adaptateur InterpretCI sur les relations entre instance de LeafFunctionRole et de SysmlPart.

La méthode *linkedToKind:withAdapter* permet soit de réutiliser une classe d'adaptateur soit d'en définir une nouvelle. Dans le second cas, une phase de mise au point de l'adaptateur est nécessaire.

Pour faciliter la mise au point des adaptateurs, nous utilisons les facilités fournies par l'environnement Smalltalk. Les méthodes déléguées à l'adaptateur sont créées à la volée lors de leur appel sur le rôle.

Le code généré provoque un appel au debugger. Ce dernier nous donne accès aux valeurs des variables de la méthode déléguée et donc à l'élément de modèle. Il est ainsi possible d'accèder à cet élément de modèle et de tester l'écriture de la méthode d'adaptation avant de l'écrire au sein de l'adaptateur dynamique. Le Listing 5.15 illustre la méthode d'adaptation *actionGroups:* créée à la volée au sein de la classe InterpretCI.

```
1 actionGroups: obj1
2   |ag|
3   ag := self createActionGroup: obj1.
4   self createOperationBlocksFor: obj1 inActionGroup: ag.
5   ^(OrderedCollection new addLast: ag).
```

Listing 5.15 – Implémentation de la méthode de création des instances d'ActionGroup dans l'adaptateur InterpretCI.

La variable *obj1* est créée à la volée par l'environnement Role4All. Role4All utilise un nom générique comme *obj1* pour les paramètres des méthodes crées dans les adaptateurs afin d'éviter que deux paramètres aient le même nom. Pour écrire le corps de la méthode créée, il est nécessaire d'utiliser l'inspecteur de Smalltalk. En utilisant l'inspecteur, il est possible de déterminer que le paramètre *obj1* de la méthode *actionGroups:* est une instance de SysmlBlockInstance. L'objectif de l'adaptateur est de créer une instance d'ActionGroup contenant une instance de

ReceiveBlock par port d'entrée, une instance de computationBlock et une instance de SendBlock par port de sortie.

Cette instance d'ActionGroup est réutilisée par le moteur d'exécution basé acteur. Ce dernier interprète une collection d'ActionGroup qui doit être renvoyée par la méthode *actionGroups:* de l'adaptateur. Une instance d'ActionGroup est créée ligne 3 et stockée dans une variable nommée *ag*. L'instance d'ActionGroup reçoit pour nom celui de l'instance de bloc suivi par *_ag1*. Cette action est réalisée par la méthode *createActionGroup:*. Cette méthode rend également l'instance d'ActionGroup active en donnant la valeur *true* à l'attribut *isActive* de l'instance d'ActionGroup. L'appel à la méthode *createOperationBlocksFor:inActionGroup:* (ligne 4) permet d'ajouter la collection de ReceiveBlocks, l'instance de ComputationBlock et la collection de SendBlocks à l'instance d'ActionGroup créée. La collection à renvoyer est créée ligne 5 et l'instance d'ActionGroup y est ajoutée.

Les adaptateurs assurent l'indépendance des rôles par rapport aux éléments de modèles en concrétisant l'unification des modèles par rapport aux rôles. La capacité offerte par Role4All de définir à la volée les méthodes d'adaptation permet d'obtenir des adaptateurs qui sont spécifiques aux besoins d'unification. L'utilisation du débogueur et celle de l'inspecteur Smalltalk permettent également de définir le corps des méthodes d'adaptation à la volée sans nécessiter de recommencer le processus d'unification des modèles. Un gain de temps est donc obtenu.

5.4.5. Définition d'un contexte

Nous avons mis au point les rôles, leurs associations avec des éléments de modèles et l'adaptation de ces éléments aux rôles. Pour faciliter la réutilisation de tout ce travail, nous définissons un contexte dédié.

Le contexte que nous souhaitons définir sert à exécuter une architecture fonctionnelle à l'aide d'un moteur d'exécution basé acteurs. Le Listing 5.16 illusttre la définition du contexte.

```
1 context := (RoleContext named: 'FunctionalArchitecture_Actors') registerContext.
```

Listing 5.16 – Définition du contexte d'exécution d'architectures fonctionnelles à l'aide d'acteurs

Ligne 1, nous définissons le nom du contexte et nous indiquons dans quel paquetage la classe associée au contexte doit être créée. Nous enregistrons directement le contexte pour son utilisation future.

Une fois le contexte défini, nous pouvons lui ajouter des règles d'association de rôles avec des éléments de modèles et d'adaptateurs sur les relations entre rôles et éléments de modèles.

```
1 context addRole: LeafFunctionRole to: SysmlBlockInstance underCondition: [ :bi | bi hasSubBlocks not].
2 context addAdapter: 'InterpretCI' to: LeafFunctionRole whenLinkedTo: SysmlBlockInstance.
3 context export.
```

Listing 5.17 – Ajout d'une règle d'association de rôle et d'une règle d'association d'adaptateur dynamique à un contexte

Ligne 1, nous ajoutons au contexte la règle qui associe le rôle LeafFunctionRole aux instances de SysmlBlockInstance qui ne possèdent pas de sous-blocs. Ligne 2, nous ajoutons la règle qui associe l'adaptateur dynamique nommé InterpretCI aux relations entre le rôle LeafFunctionRole et les instances de SysmlBlockInstance. Afin de réutiliser le contexte, il est possible d'exporter le contexte sous la forme d'instructions R2DL et Smalltalk. Cela est réalisé ligne 3 par l'appel à la

méthode *export*. Le code ainsi obtenu pourra être rechargé dans une nouvelle image Smalltalk.

5.4.6. Exécution du modèle

Les rôles associés aux éléments de modèles et les adaptateurs dynamiques liés aux associations, permettent au moteur d'exécution basé acteurs d'interpréter directement les éléments des modèles SysML et eFFBD. La collection des rôles est passée en paramètre de la méthode d'initialisation de l'environnement de simulation. Le Listing 5.18 décrit l'implémentation de cette méthode.

```
1  populateFrom: aCollection
2    "Given a Collection creates the associated visualisations and actors."
3    |lfgs lfs comms|
4    lfgs := Dictionary new.
5    lfs := Dictionary new.
6    comms := Dictionary new.
7    (aCollection select: #isLogicalFunctionsGroup) do: [ :each | lfgs add: (each functionName) -> each ].
8    (aCollection select: #isLeafFunction) do: [ :each | lfs add: (each functionName ) -> each ].
9    (aCollection select: #isCommunicationLink) do: [ :each | comms add: (each linkName) -> each ].
10   self populateFromLfgs: lfgs leafFuncts: lfs comms: comms.
```

Listing 5.18 – Méthode d'initialisation de la simulation.

L'outil d'exécution prend en entrée trois tables de hachage, une pour les groupes logiques de fonctions, une pour les fonctions feuilles et une pour les liens de communications (ligne 10). Ces trois dictionnaires sont construits à partir des éléments de la collection aCollection qui répondent positivement aux tests *isLogicalFunctionsGroup*, *isLeafFunction* et *isCommunicationLink* (lignes 4 à 9). L'appel à la méthode *populateFromLfgs:leafFuncts:comms:* initialise le moteur d'exécution. Les rôles étant capables de répondre aux différentes méthodes utilisées pour initialiser la simulation, la collection passée en paramètre de l'initialisation peut être une collection de rôles sans nécessiter de changements dans le corps de la méthode. La Figure 5.15 illustre le résultat obtenu avec les modèles SysML et eFFBD associés aux rôles que nous avons définis.

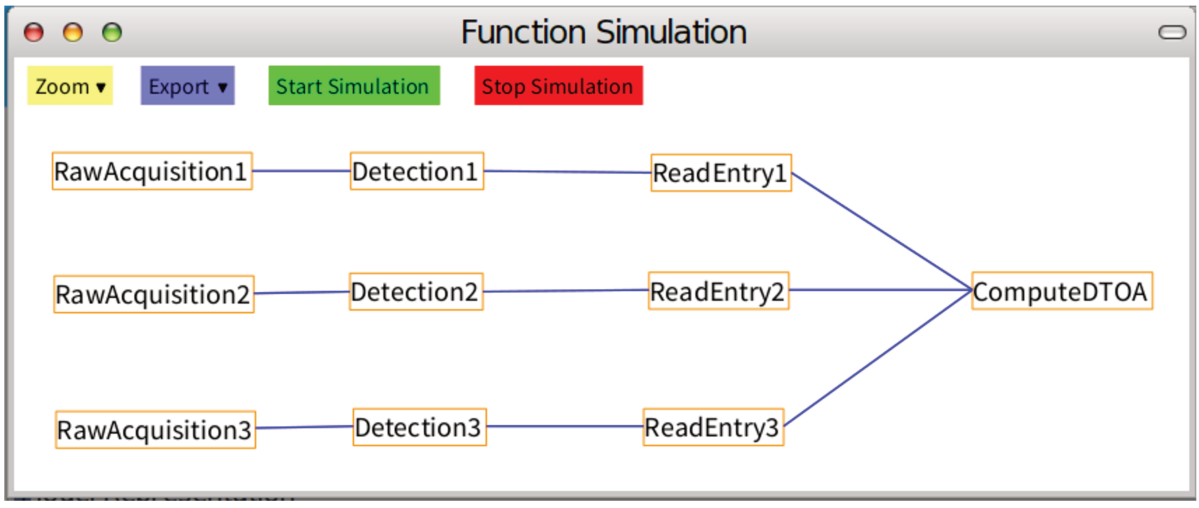


FIGURE 5.15. – Affichage obtenu après l'initialisation du moteur d'exécution avec les rôles.

Pour l'outil de simulation, les éléments représentés Figure 5.15 sont des instances du méta-modèle d'architecture fonctionnelle de l'outil. Ce modèle peut être exécuté par l'outil normalement. Mais en réalité, les éléments affichés sont les rôles que nous avons crées et associés aux éléments de modèles SysML et eFFBD de l'observatoire MeDON. L'exécution du modèle de rôles interprétés comme un modèle d'architecture fonctionnelle de l'outil utilise les données contenues dans les modèles SysML et eFFBD indistinctement pour l'outil. Cette exécution nous permet d'obtenir un diagramme de séquence complet de l'exécution de la simulation. Un extrait du diagramme de séquence obtenu est fourni Figure 5.16.

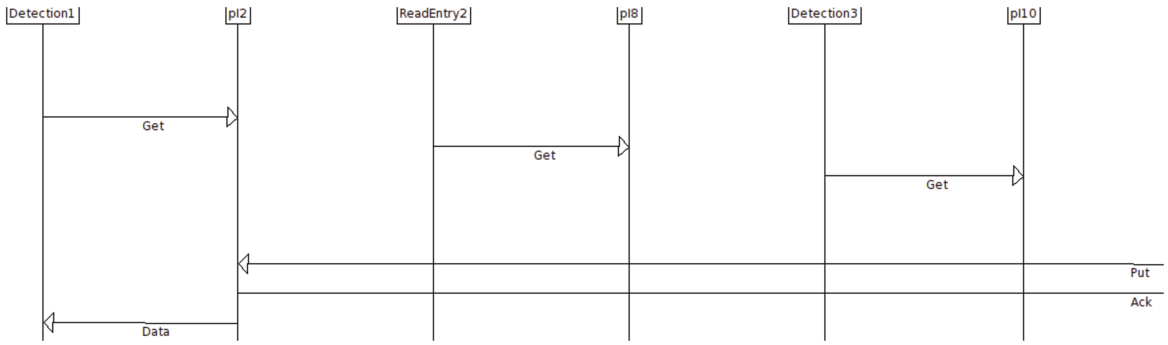


FIGURE 5.16. – Extrait du diagramme de séquence obtenu lors de l'exécution du modèle d'architecture fonctionnelle.

Ce diagramme de séquence permet de vérifier :

— que les fonctions écrivent bien sur les liens de communications sortants,
— que les fonctions lisent effectivement sur leurs liens de communications entrants,
— que les protocoles de types rendez-vous ou asynchrone sont bien respectés.

Role4All permet de définir des types de rôles qui sont indépendants des éléments de modèles et des différents types de modèles utilisés. Les rôles (instances des types de rôles) définis à l'aide du DSL R2DL de Role4All peuvent être associés à des éléments de types différents et provenant de langage de modélisation différents sans que le modeleur de rôles ait à se soucier du type des éléments auxquels il associe des rôles. Les rôles permettent de créer une interface entre l'outil de simulation et les différents modèles système. L'outil de simulation peut utiliser les éléments de modèles définis dans des langages de modélisation différents indistinctement grâce aux rôles qui modélisent le point de vue de l'outil de simulation. De plus, les adaptateurs définis dans le méta-modèle de rôle sur lequel Role4All s'appuie permettent de concrétiser l'unification de modèles fournis par les rôles.Ces adaptateurs permettent de travailler de manière équivalente avec des concepts équivalents définis dans des langages de modélisation différents. La capacité à modifier à la volée la définition d'un type de rôle et d'un adaptateur donne également la capacité d'ajouter de l'information dans les rôles et les adaptateurs pour permettre l'utilisation des modèles avec un outillage de simulation sans modifier les modèles.

En plus de l'hétérogénéité au niveau des modèles conceptuels en entrée, il est également nécessaire d'être capable de gérer l'hétérogénéité des outillages de simulation.

5.5. Adaptation du modèle de rôles pour permettre l'utilisation d'un nouvel environnement de simulation

Dans la section précédente, nous avons illustré la définition d'un modèle de rôle pour interfacer un outil qui permet d'exécuter des modèles d'architectures fonctionnelles en les interprétant. Cependant, dans un projet de système comme un observatoire sous-marin plusieurs simulateurs sont utilisés. En effet, un simulateur est plus adapté à certaines tâches que d'autres. Ainsi, utiliser plusieurs simulateurs permet d'utiliser le simulateur le plus adapté en fonction de l'objectif recherché. Dans le cadre de l'observatoire MeDON, l'aspect réseau est important. Nous avons donc choisi de simuler l'architecture fonctionnelle de l'observatoire avec un simulateur de réseau nommé OMNeT++.

Pour cela, nous avons repris la méthodologie présenté précédemment :

— analyse de l'API d'OMNeT++,
— définition des types de rôles en fonction des éléments d'intérêts de l'API,
— définition des rôles et de leurs associations avec des éléments de modèles,
— définition des adaptateurs.

Toutefois, OMNeT++ est un simulateur générique basé événements discrets. Le point de vue définit par l'API d'OMNeT++ est un point de vue simulation par événements discrets et ne fait pas référence aux concepts de l'architecture fonctionnelle. Or dans la section précédente, nous avons défini des types de rôles et un modèle de rôles pour permettre d'associer un point de vue architecture fonctionnelle aux éléments contenus dans les modèles SysML et eFFBD de l'observatoire MeDON. Nous allons illustré dans cette section comment la capacité des rôles à jouer des rôles permet de réutiliser des modèles de rôles déjà définis. Cette réutilisation crée un chaînage de points de vues qui permet d'adapter l'utilisation des modèles système à l'utilisation d'un nouvel outil de simulation. La possibilité de faire jouer des rôles par des rôles permet de modulariser la conception d'un modèle de rôles. L'unification des modèles pour permettre la simulation des modèles est elle-même rendue modulaire.

La Sous-section 5.5.1 détaille l'analyse de l'outil OMNeT++ qui a guidé l'élaboration des types de rôles présentés dans la sous-section 5.5.2. Comme nous voulons simuler une architecture fonctionnelle, des instances de ces types rôles sont associés aux rôles qui décrivent une architecture fonctionnelle. Ces associations sont présentées dans la Sous-section 5.5.3. Les rôles associés servent d'entrée à un visiteur décrit dans la Sous-section 5.5.4. Le visiteur sert à générer une simulation OMNeT++ dont le résultat est présenté dans la Sous-section 5.5.5.

5.5.1. Analyse de l'outil de simulation de réseau OMNeT++

OMNeT++ est un outil de simulation réseau par événements discrets. Différents éléments s'échangent des messages. Ces échanges prennent la forme d'événements de réception de messages. Ces événements sont datés ce qui permet de faire avancer le temps dans la simulation. Les concepts utilisés au sein du simulateur OMNeT++ ont été méta-modélisés. La Figure 5.17 présente la partie du méta-modèle obtenu pour la description d'une simulation.

Une simulation contient le réseau à simuler (méta-classe OppNetwork) ainsi que la description des modules (méta-classe OppModule) dont les instances composent le réseau.

Un module possède des ports (méta-classe OppGate). Chaque port est identifié par un identifiant et un nom. Les ports peuvent être des ports d'entrée (méta-classe OppInGate), de sortie (méta-classe OppOutGate) ou d'entrée/sortie (méta-classe OppInOutGate).

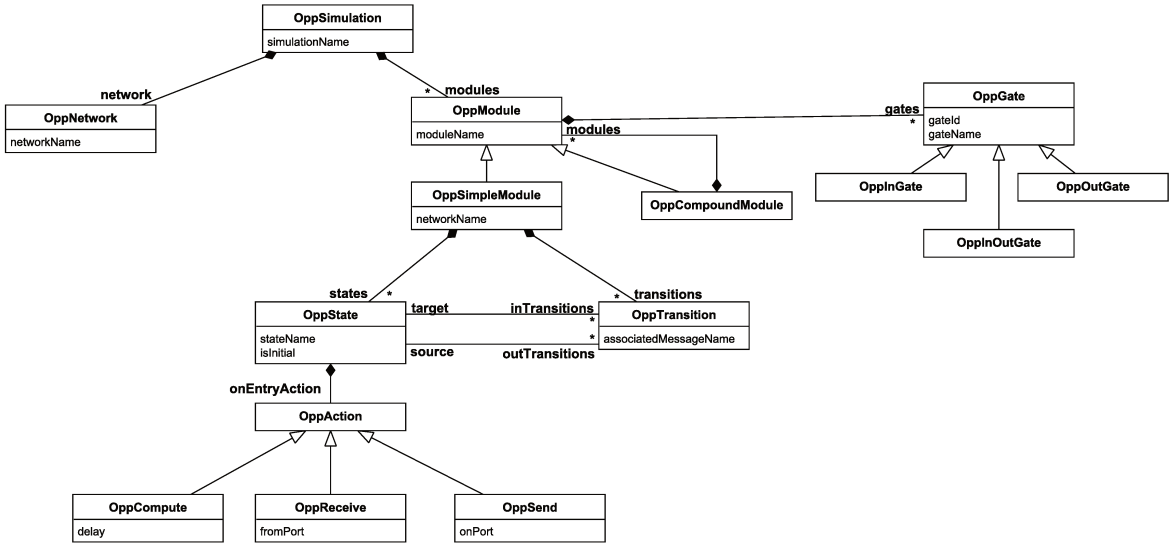


FIGURE 5.17. – Partie du méta-modèle d'OMNeT++ décrivant la structure d'une simulation.

Les modules sont de deux types : composite (méta-classe OppCompoundModule) ou simple (méta-classe OppSimpleModule). Les modules simples sont les seuls à posséder un comportement.

Ce comportement est modélisé par l'intermédiaire d'une machine à états (méta-classe OppState et méta-classe OppTransition). Chaque état possède une action à effectuer à l'entrée dans l'état. Cette action peut être de trois types : effectuer une opération (méta-classe OppCompute), recevoir une donnée (méta-classe OppReceive) ou envoyer une donnée (méta-classe OppSend).

Le réseau à simuler est identifié par un nom. La structure du réseau est décrite Figure 5.18.

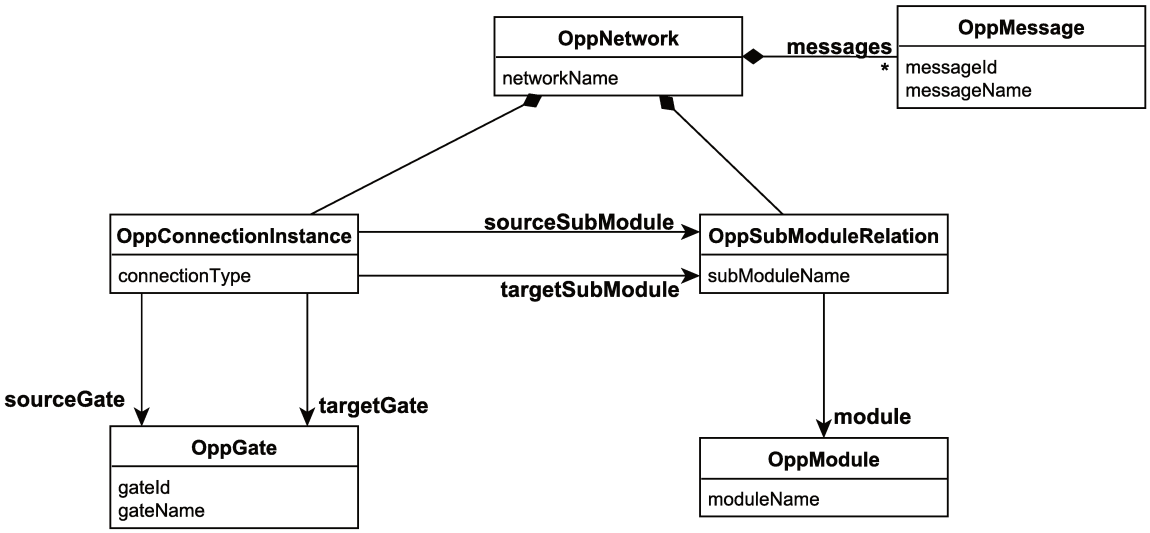


FIGURE 5.18. – Partie du méta-modèle d'OMNeT++ décrivant la structure d'un réseau.

Un réseau est constitué d'instances de modules (méta-classe OppSubModuleRelation). Chaque

instance de modules possède un nom et une référence vers son type.

Le réseau décrit également les relations qui existent entre les différentes instances de modules (méta-classe OppConnectionInstance). Cette instance de connexion décrit le type de lien qui peut être idéal, induire un délai ou avoir un débit limité. Chaque instance de connexion fait le lien entre un port source et un port cible.

La description d'un réseau contient également la définition des messages qui peuvent être échangés entre les différents modules (méta-classe OppMessage). Chaque message est identifié par un identifiant et un nom.

Les modèles conformes à ce méta-modèle servent de base à une génération de code. OMNeT++ définit un langage spécifique pour la description d'un réseau (format NED). La partie du méta-modèle décrivant la structure du réseau sert à la génération de fichiers au format NED. De plus, chaque module est décrit au format NED pour définir sa structure et une classe écrite en langage C++ permet de définir le comportement d'un module simple. La méta-classe OppModule sert de base à l'écriture du fichier NED correspondant et à la classe C++ si le module est un module simple.

5.5.2. Modèle de rôles associé à des rôles

OMNeT++ est un outil de simulation basés événements discrets. Les concepts d'une simulation basée événements discrets sont utilisés pour définir un point de vue sur les modèles à exécuter. Par conséquent, ces concepts sont à utiliser pour définir des rôles qui définissent un point de vue simulation basée événements discrets. Zeigler et al. ont donné la description d'un système simulé basé sur la notion d'événements discrets [103] : le formalisme DEVS (Discrete Event Specification). Dans DEVS, un système est composé de composants composites (qui peuvent contenir d'autres composants) ou atomiques. Les composants atomiques possèdent un comportement qui peut être décrit par une machine à états. Les composants peuvent être reliés les uns avec les autres pour créer des connexions et permettre des échanges d'événements. Les événements ont lieu à des moments précis dans le temps. Le temps avance dans la simulation grâce aux dates d'occurrence des événements.

Nous avons définis six rôles à partir de la définition d'un système DEVS :

AtomicDevsRole décrit un composant atomique d'un système DEVS.

CompositeDevsRole décrit un composant composite d'un système DEVS.

ConnectionRole décrit une connexion entre deux composants.

SourceRole décrit un composant qui émet un événement.

ReceiverRole décrit un composant qui reçoit un événement.

DevsSystemRole décrit un système DEVS.

La définition du modèle de rôles pour un point de vue simulation par événement discret permet à tous les outils de simulation par événements discrets d'avoir un point de vue adapté à leurs besoin. Il n'est donc pas nécessaire de se lier à des rôles éloignés du domaine et de faire des contorsions dans l'adaptateur pour arriver sur l'outillage. En effet, le modèle de rôle respecte les attentes des outils de simulation par événements discrets. Ce modèle de rôle peut ensuite être associé à d'autres modèles de rôles pour permettre de simuler ces modèles en utilisant une simulation par événements discrets.

5.5.3. Association de rôles avec des rôles

Nous souhaitons pouvoir simuler nos modèles SysML et eFFBD en adoptant un point de vue architecture fonctionnelle. L'outil de simulation utilisé sera OMNeT++ qui est un outil de simulation par événements discrets. L'architecture fonctionnelle sera simulée en utilisant une simulation par événements discrets.

Les rôles d'architecture fonctionnelle que nous avons affectés aux éléments de modèles SysML et eFFBD permettent d'extraire des modèles système toutes les informations relatives à une architecture fonctionnelle. Comme nous souhaitons simuler cette architecture en utilisant une simulation basée événements discrets, nous associons les rôles qui décrivent une simulation par événements discrets aux rôles qui décrivent une architecture fonctionnelle. Le tableau 5.3 décrit les associations des rôles ainsi que les conditions sous lesquelles ces associations peuvent être réalisées.

Table 5.3. – Association des rôles décrivant un système DEVS à ceux décrivant une architecture fonctionnelle.

Rôle	Player	Condition
AtomicDevsRole	LeafFunctionRole	
AtomicDevsRole	CommunicationLinkRole	
SenderRole	AtomicDevsRole	Associé à un LeafFunctionRole qui possède des liens sortants.
SenderRole	AtomicDevsRole	Associé à un CommunicationLinkRole
ReceiverRole	AtomicDevsRole	Associé à un LeafFunctionRole qui possède des liens entrants.
ReceiverRole	AtomicDevsRole	Associé à un CommunicationLinkRole.

De façon naturelle, les fonctions feuilles deviennent des composants atomiques au sein d'un système DEVS. Du fait de notre volonté de donner un comportement aux liens de communication pour simuler des protocoles de types rendez-vous ou asynchrone, les liens de communication deviennent eux-aussi des composants atomiques. De plus, par défaut, ces composants doivent jouer le rôle d'émetteur et de récepteur.

Le rôle ConnectionRole est un rôle composite qui est constitué d'une instance du rôle SenderRole et d'une instance du rôle ReceiverRole. Le rôle DevsSystemRole est un rôle composite qui contient toutes les instances des rôles AtomicDevsRole et ConnectionRole.

Des exemples de code dans l'environnement Role4All pour implémenter des associations de rôles ont déjà été présentées. Par conséquent, pour des raisons de présentation, nous omettons de donner le code correspondant aux règles décrites dans le tableau 5.3.

Les deux niveaux de modèles de rôles rendent la compréhension du modèle de rôles obtenu complexe. Nous avons utilisé notre fonctionnalité d'export GraphViz pour obtenir une image plus lisible du modèle. Un extrait du résultat est donné Figure 5.19.

L'instance de rôle LeafFunctionRole32 est associée à l'instance functionReadEntry3 de la classe EffbdFunction. L'instance de rôle permet de comprendre l'instance d'EffbdFunction comme étant une partie d'une architecture fonctionnelle. Une instance du rôle AtomicDevs (AtomicDevs44) est associée à l'instance de LeafFunctionRole. Ceci permet de considérer l'instance de LeafFunctionRole comme un composant atomique dans une simulation DEVS. Par relation de hiérarchie de rôles, la fonction functionReadEntry3 sera simulée comme un compo-

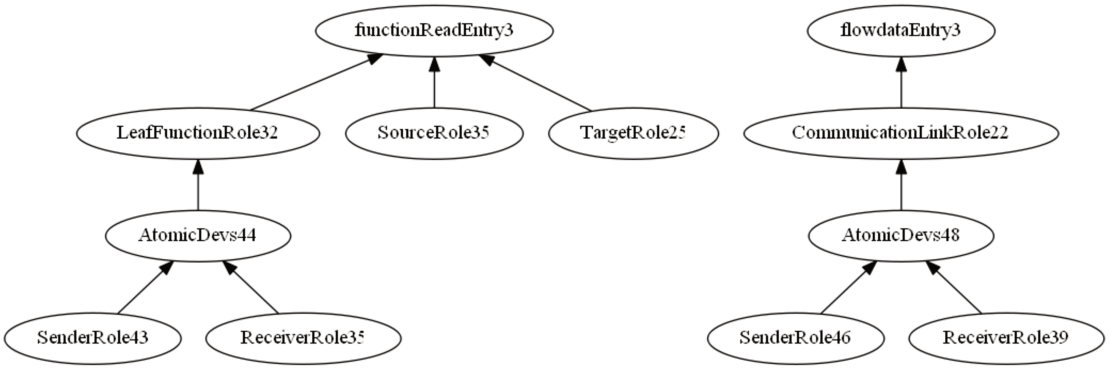


FIGURE 5.19. – Extrait de l'export GraphViz de l'allocation des rôles décrivant un système DEVS.

sant atomique DEVS.

Le modeleur de la simulation n'a cependant pas besoin de savoir qu'il manipule une instance d'EffbdFunction. Il n'a besoin de voir que le point de vue de l'architecture fonctionnelle. L'outil de simulation, quant à lui, peut manipuler uniquement des concepts proches de son domaine sans se soucier de savoir qu'il manipule une architecture fonctionnelle extraite de modèles SysML et eFFBD.

5.5.4. Ecriture du visiteur

Pour générer le code de la simulation, nous avons choisi d'utiliser le pattern Visiteur [3]. Ce pattern permet de parcourir facilement une collection d'objets pour réaliser des traitements en fonction des objets parcourus.

Dans la simulation que nous souhaitons générer, nous privilégions les communications point à point. Ainsi chaque lien entre deux fonctions est modélisé par un lien de communication dédié sur lequel une seule fonction peut écrire et une seule fonction peut lire. La Figure 5.20 présente la structure attendue dans le modèle OMNeT++ généré.

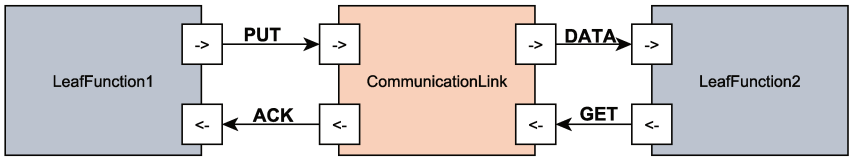


FIGURE 5.20. – Illustration du résultat attendu de la création du modèle OMNeT++

Chaque fonction possède pour chaque lien de communication sortant un couple port entrant / port sortant. Un message de type Put est envoyé sur un port sortant et un message de type Ack est attendu sur le port entrant associé.

Chaque fonction possède pour chaque lien de communication entrant un couple port sortant / port entrant. Un message de type Get est envoyé sur un port sortant et un message de type Data est attendu sur le port entrant associé.

Chaque module représentant un lien de communication possède un couple de port entrant / sortant pour traiter les messages provenant du module émetteur. Un message de type Put est

attendu sur le port d'entrée et un message de type Ack est envoyé sur le port sortant. Pour traiter les messages provenant du module récepteur, un couple de port entrant / sortant est ajouté au module du lien de communication. Un message de type Get est attendu sur le port en entrée et un message de type Data est envoyé sur le port sortant.

Un visiteur est implémenté pour réaliser la création des instances des méta-classes OMNeT++ à partir des instances des rôles décrivant un système DEVS. Le Listing 5.19 donne le code de la méthode du visiteur utilisé pour la création d'un module simple OMNeT++.

```
1 createAtomicDevsInstanceFrom: anAtomicDevs
2   |module|
3   module := result at: (anAtomicDevs moduleTypeName )
4     ifAbsent: (self createOppSimpleModule: anAtomicDevs).
```

Listing 5.19 – Code du visiteur pour la création d'un module simple OMNeT++ à partir d'une instance du rôle AtomicDevsRole.

La méthode *createAtomicDevsInstanceFrom:* prend en argument une instance du rôle AtomicDevsRole (ligne 1). Pour éviter de générer plusieurs fois le même module, tous les modules générés sont stockés dans un dictionnaire nommé result. Si le module qui porte le même nom que le type référencé par l'instance d'AtomicDevsRole n'existe pas encore (lignes 3 et 4) alors le processus de transformation est enclenché (ligne 4). L'appel à la méthode *createOppSimpleModule:* permet de créer une instance d'OppSimpleModule à partir des informations contenues dans l'instance de rôle anAtomicDevs. La méthode de création permet notamment de créer la machine a états du module à partir des actions récupérées depuis l'instance d'AtomicDevsRole.

Les adaptateurs dynamiques sont écrits en fonction des méthodes utilisées par le visiteur.

5.5.5. Simulation obtenue

Après avoir généré le code associé aux instances des méta-classes OMNeT++, nous avons obtenu la simulation illustrée Figure 5.21.

Les instances de LeafFunctionRole tels que ReadEntry3 ou Detection3 et de CommunicationLinkRole tels que dataEntry3 ou pI12 ont été transformées en modules simples. Les liens entre les différents modules sont aussi générés. L'exécution de la simulation permet de voir les échanges de messages entre les modules et un log permet de s'assurer du bon fonctionnement de l'ensemble.

De plus, lors de la génération de code, nous avons ajouté le log de l'état d'activité du module sous la forme d'un entier. La valeur 0 (respectivement 1) est associée à un état inactif (resp. actif). OMNeT++ fourni les moyens de dessiner un graphe de cette valeur et ainsi d'observer l'état d'activité d'un module au cours du temps. La Figure 5.22 montre l'état d'activité du module lié à la fonction ReadEntry1.

Entre les instants 0 et 20, le module est inactif : aucun message n'a encore été reçu. Le module est réveillé à l'instant 20 par la réception d'un message et reste actif jusqu'à l'instant 24.

La propriété des rôles de jouer des rôles fait partie des propriétés des rôles intégrées à la définition des rôles dans Role4All. Cette propriété nous a permis de réutiliser le modèle de rôles d'architecture fonctionnelle précédemment définis pour le simuler à l'aide d'un simulateur par événements discrets. Cette réutilisation a permis de modéliser la simulation par événements discrets de l'architecture fonctionnelle définie dans les modèles SysML et eFFBD. En effet, le

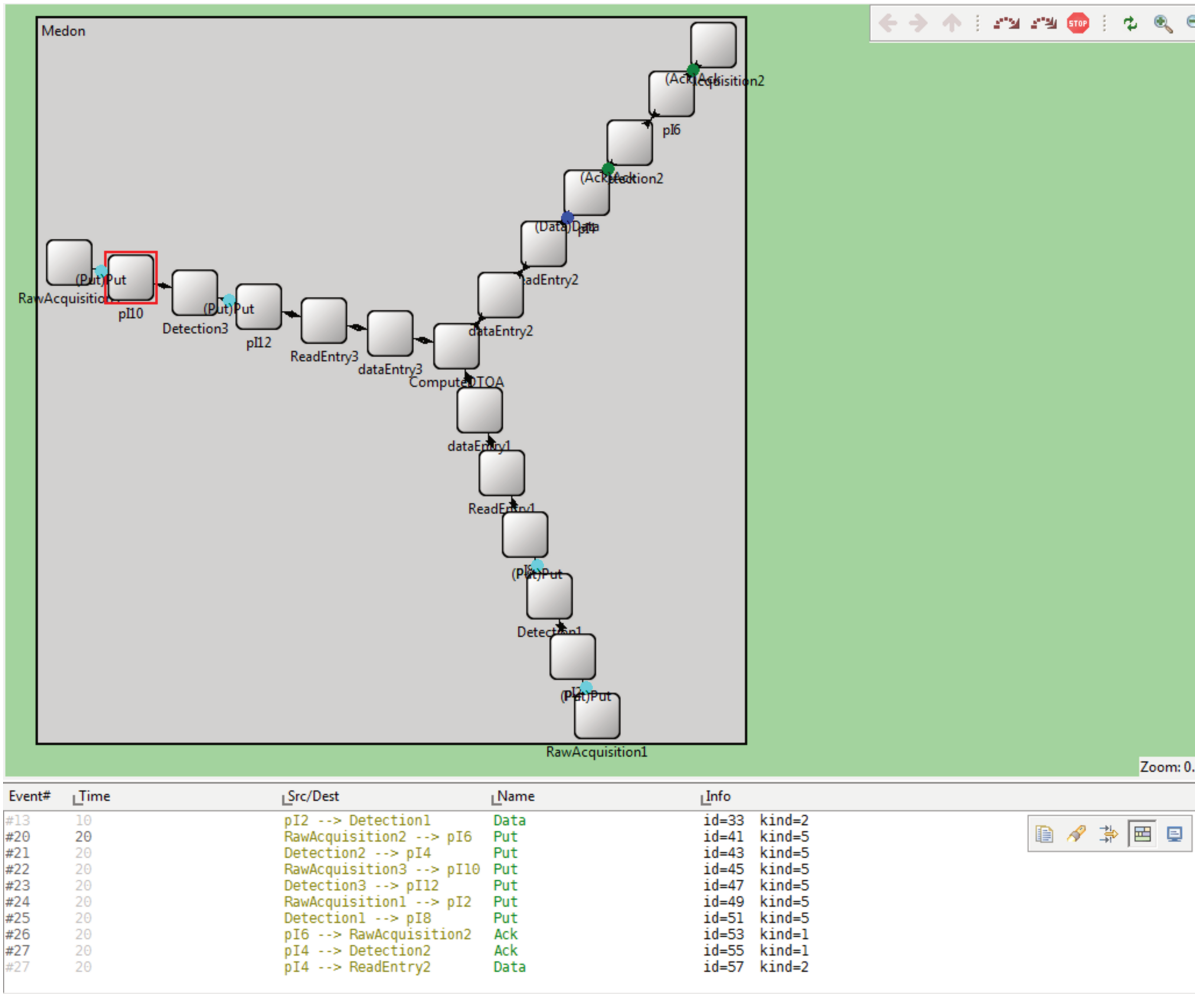


FIGURE 5.21. – Résultat de la génération de la simulation à partir des instances des éléments OMNeT++.

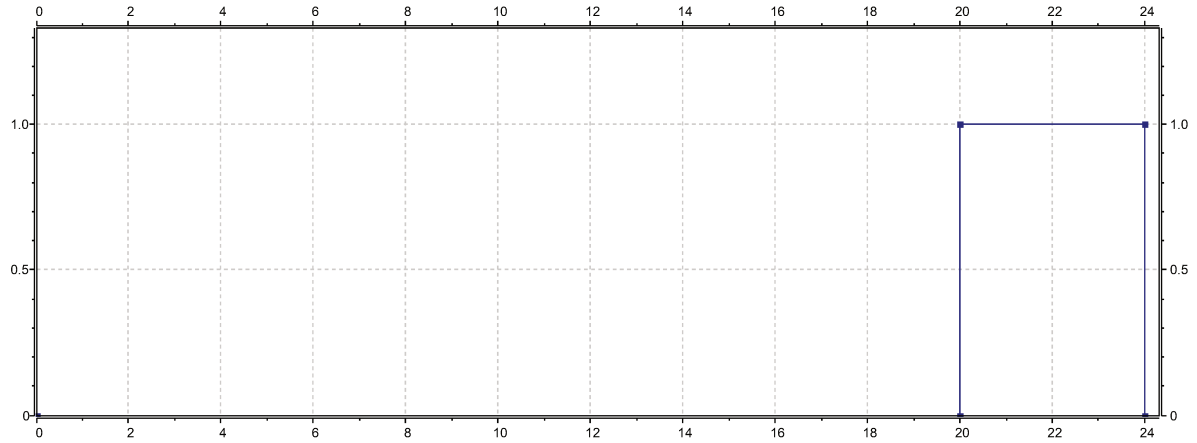


FIGURE 5.22. – Graphique de l'activité du module ReadEntry1.

modèle de rôles d'architecture fonctionnelle a permis d'unifier les modèles SysML et eFFBD en fonction du point de vue architecture fonctionnelle. Associer un modèle de rôles simulation par événements discrets aux rôles d'architecture fonctionnelle a permis de se concentrer sur la simulation par événements discrets de l'architecture fonctionnelle unifiée et n'a pas nécessité d'analyser les modèles SysML et eFFBD pour en extraire les informations relatives à l'architecture fonctionnelle. Le passage en deux étapes à la simulation par événements discrets a facilité la création de la simulation.

5.6. Adaptation à un remplacement d'outil de simulation

Dans la section précédente, nous avons illustré l'utilisation de modèles de rôles pour permettre de simuler les mêmes modèles avec des outils de simulation de types différents. Un projet de système comme l'observatoire sous-marin MeDON a une longue durée de vie. Ainsi, un outil de simulation utilisé pour créer des prototypes virtuels peuvent être remplacé par un outil équivalent . Pour illustrer l'utilisation de Role4All pour tenir compte du remplacement d'un outil de simulation, nous avons choisi de remplacer le simulateur OMNeT++ par un autre simulateur basé événements discrets nommé ADEVS[5].

ADEVS et OMNeT++ sont deux outils de simulation par événements discrets. Ils partagent donc le même point de vue. Dans Role4All, la définition des types de rôles et des association de rôles avec des éléments de modèles sont indépendantes des langages de simulation mais aussi des outils de simulation. Par conséquent, il est possible d'utiliser les mêmes types de rôles et la même association de rôles avec des éléments de modèles pour ADEVS que pour OMNeT++. Dans cette section, nous illustrons l'utilisation du concept d'adaptateur défini dans Role4All pour s'adapter au remplacement d'un outil de simulation comme OMNeT++ par un outil de simulation équivalent comme ADEVS.

La sous-section 5.6.1 présente la structure d'une simulation ADEVS dont l'analyse permet l'écriture des adaptateurs qui sont utilisés par le visiteur pour la génération de code présenté dans la sous-section 5.6.2. La sous-section 5.6.3 décrit les adaptateurs définis pour répondre aux exigences de la génération du code pour ADEVS.

5.6.1. Présentation du simulateur ADEVS

ADEVS est un simulateur basé événements discrets qui permet de construire des simulations parallèles ainsi que des simulations dont la structure est capable d'évoluer [1]. ADEVS est implémenté en C++ mais propose un *binding* Java sur l'implémentation C++. Nous avons utilisé le *binding* Java dans le cadre du projet MeDON. La Figure 5.23 présente le modèle du *binding* Java de la librairie du simulateur ADEVS.

Une simulation représentée par la classe Simulator possède des références vers un ensemble d'instances de classes implémentant l'interface EventListener et une référence vers une instance de la classe DEVS.

L'interface EventListener permet de décrire des opérations qui doivent être réalisés lors de la production d'une sortie par un élément du modèle DEVS ou lors d'un changement d'état au sein du modèle.

L'instance de la classe DEVS associée à l'instance de Simulator représente le modèle à simuler. Il s'agit soit d'un bloc atomique soit d'un réseau.

5. http://web.ornl.gov/~1qn/adevs/

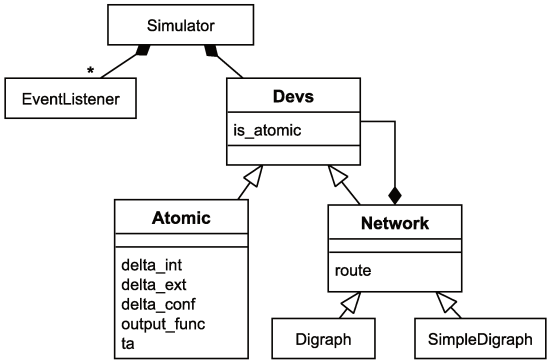


FIGURE 5.23. – Modèle du *binding* Java pour le simulateur ADEVS (réalisé d'après la documentation en ligne).

ADEVS implémente la description d'un système à événements discrets tel que spécifié dans le formalisme DEVS décrit par Zeigler et al. [103].

La classe Atomic représente des composants du système DEVS qui ne seront pas redécomposer mais qui possèdent un comportement. Conceptuellement, ce comportement peut être vu comme une machine à états. La méthode *ta* décrit pour chaque état le temps pendant lequel le composant peut rester dans l'état. A l'expiration du temps définit par *ta*, la fonction de transition interne (méthode *delta_ int*) est appelée. Un composant atomique peut également recevoir des événements en provenance d'autres composants. Sur réception d'un événement, la fonction de transition externe (méthode *delta_ ext*) est appelée. Il peut arriver qu'il y ait un conflit entre les appels aux deux fonctions de transitions. En effet, un événement peut être reçu en même temps que le *timer* d'un état arrive à expiration. Dans ce cas, la résolution de conflit doit être explicité. C'est le rôle de la méthode *delta_ conf*. Lors d'une transition entre deux états, le composant peut générer une sortie. Celle-ci est générée par la méthode *output_ func*.

La classe Network représente un groupement d'éléments DEVS qui peuvent donc être soit des éléments atomiques soit d'autres groupements d'éléments. La méthode *route* permet de définir les liaisons entre les différents constituants du réseau. La classe Network possède deux sous-classes : la classe SimpleDigraph et la classe Digraph. La classe SimpleDigraph permet de représenter des composants à simuler ne possédant qu'une entrée et qu'une sortie. Au contraire, la classe Digraph permet de représenter des composants à plusieurs entrées et plusieurs sorties.

La structure de ce modèle permet de guider la génération de code qui est nécessaire pour pouvoir simuler nos modèles SysML et eFFBD à l'aide d'ADEVS.

5.6.2. Ecriture d'un visiteur pour la création des éléments de modèle ADEVS

Pour générer le code de la simulation ADEVS, nous reprenons le principe de l'utilisation du pattern Visiteur.

Dans le cadre de la génération du code pour le simulateur OMNeT++, nous avions déjà défini un visiteur pour parcourir un modèle de rôle pour simulation à événements discrets. Le visiteur requis pour générer la simulation ADEVS peut suivre la même structure que le visiteur utilisé pour la simulation OMNeT++ car les deux simulations sont basées événements discrets.

Fort de ce constat, nous pouvons définir un visiteur abstrait pour la génération de simulation par événements discrets utilisant notre modèle de rôles. Les visiteurs pour la simulation

OMNeT++ et pour la simulation ADEVS en héritent comme illustré Figure 5.24.

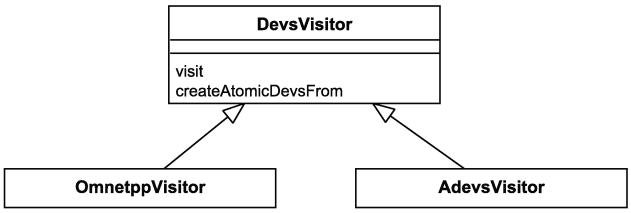


FIGURE 5.24. – Diagramme de classes pour les visiteurs générant des simulations basées événements discrets.

La classe DevsVisitor définit la méthode *visit* qui effectue le parcours de tous les éléments d'une collection. Elle définit également la méthode *createAtomicDevsFrom* qui est utilisée pour créer des instances de composants atomiques dans les différents outils de modélisation. Le contenu de cette méthode est propre à chaque outil de simulation. Elle doit ainsi être redéfinie dans les visiteurs associés aux différents outils de simulation.

Dans le cas de la génération de code pour le simulateur ADEVS, tous les rôles AtomicDevsRole doivent permettre la génération d'une sous-classe de la classe Atomic définie par ADEVS. Le code des différentes méthodes est obtenu par l'intermédiaire de la machine à états récupérée via le rôle AtomicDevsRole. Les transitions de la machine à états sont utilisées pour générer le code de la méthode *delta_ext* tandis que les actions associées à la transition sont utilisées pour le code de la méthode *output_fn*. Les délais éventuels servent pour le code de la méthode *ta*. Enfin, le principe de la méthode *delta_conf* est de privilégier les événements internes par rapport aux événements externes.

Nous constatons que la décomposition en terme de composants atomiques est identique entre les deux outils de simulation. Cependant, la structure de la machine à états utilisée pour OMNeT++ est différente de celle utilisée pour ADEVS. En effet, au sein de l'outil OMNeT++, les actions sont réalisées à l'entrée dans un état. Pour ADEVS, les actions sont réalisées au moment où la transition entre états à lieu. Cette différence nécessite de définir de nouveaux adaptateurs entre les rôles AtomicDevsRole et LeafFunctionRole pour générer une machine à états utilisable avec le simulateur ADEVS.

5.6.3. Ecriture d'adaptateurs dynamiques

Pour générer une machine à états utilisable avec le simulateur ADEVS, nous devons modifier les adaptateurs qui sont associés aux relations entre les rôles AtomicDevsRole et les rôles LeafFunctionRole. Comme nous souhaitons conserver la capacité de créer des simulations pour le simulateur OMNeT++ intacte, nous créons de nouveaux adaptateurs. Des instances de ces adaptateurs remplaceront celles précédemment associées aux relations. Le Listing 5.20 donne le code pour réaliser cette opération.

```
1 AtomicDevsRole linkedToKind: LeafFunctionRole withAdapter: 'AdaptLfForAdevs'.
```

Listing 5.20 – Définition d'un adaptateur pour adapter le rôle LeafFunctionRole au rôle AtomicDevsRole pour l'outil ADEVS

L'appel à la méthode *linkedToKind:withAdapter:* permet de créer la classe d'adaptateur nommée AdaptLfForAdevs. Des instances de cette classe sont associées aux relations entre les instances du rôle AtomicDevsRole et du rôle LeafFunctionRole. Les instances d'adaptateurs qui étaient précédemment associées aux mêmes relations sont remplacées mais la définition de l'adaptateur existe toujours. Nous pourrons à nouveau utiliser l'adaptateur remplacé ultérieurement.

L'adaptateur AdaptLfForAdevs doit fournir la méthode *stateMachine* qui permet de créer les machines à états utilisables avec l'outil ADEVS. Cette méthode doit :

1. créer pour chaque opération de lecture un état d'une durée 0 émettant lors de la transition un message de type GET et un état de durée infinie attendant un message de type DATA,
2. créer pour chaque bloc de calcul un état d'une durée égale au délai du bloc de calcul,
3. créer pour chaque opération de lecture un état d'une durée 0 émettant lors de la transition un message de type PUT et un état de durée infinie attendant un message de type ACK.

L'appel de cette méthode pour toutes les associations entre les rôles AtomicDevsRole et LeafFunctionRole permet de créer toutes les machines à états des composants atomiques qui constituent le modèle de la simulation.

Nous avons définis un visiteur abstrait pour le parcours d'un modèle de rôle de simulation basée événements discrets. Ce visiteur est spécialisé pour traiter le cas des différents outils de simulation désirés. Dans Role4All, la définition des types de rôles et de l'association de rôles avec des objets sont indépendantes des langages de modélisation et des outils qui utilisent les rôles. La prise en compte des différences se fait au niveau des adaptateurs qui se trouvent sur les liens entre les objets et les rôles. Il est possible de réutiliser un modèle de rôle pour plusieurs outils. Ainsi, contrairement aux transformations de modèles classiques, le temps de la définition de la correspondance entre les concepts des modèles et ceux du point de vue de l'outil est économisé.

5.7. Adaptation à un changement dans les langages de modélisation

Dans les sections précédentes, nous avons illustré l'utilisation des parsers combinateurs pour créer des importateurs de modèles définis dans des langages différents et facilitant l'ajout de la gestion de nouveaux langages de modélisation. Nous avons également illustré les capacités des rôles utilisés par Role4All pour permettre d'unifier des modèles système pour créer des simulations et gérer à la fois l'utilisation d'un nouvel outil de simulation ou un remplacement d'outil de simulation.

Nous avons montré la capacité de Role4All à gérer les évolutions du côté des outils de simulation. Cependant, la durée de vie des systèmes comme l'observatoire MeDON est longue. Les outils et les langages de modélisation vont donc évoluer également notamment au niveau de leur structure. Cette section illustre comment Role4All permet de gérer l'évolution de la définition d'un langage de modélisation. Tout d'abord, contrairement aux outils de transformation de modèles existants, la représentation interne des modèles est réalisée sous la forme de classes Smalltalk et est accessible dans Role4All. La représentation interne des modèles doit donc être modifiée manuellement mais elle n'oblige cependant pas à modifier toutes les

classes Smalltalk qui concernent le langage de modélisation modifié. Seules les classes correspondantes aux concepts modifiés du langage de modélisation doivent être modifiées (sous-section 5.7.1). Ensuite, il faut tenir compte de l'évolution du format de sérialisation pour les modèles du langage de modélisation modifiés. Les parsers combinateurs utilisés pour les importateurs modulaires de Role4All permettent de limiter l'impact des modifications dans le format de sérialisation sur l'importateur. Tout l'importateur n'a pas besoin d'être réécrit pour tenir compte des modifications du format de sérialisation (sous-section 5.7.2). Enfin, le modèle importé est associé à des rôles. Role4All permet de modifier un modèle de rôles existant pour l'adapter aux modifications de la sémantique des éléments de modèles du langage de modélisation modifié (sous-section 5.7.3).

5.7.1. Modification du méta-modèle du langage SysML et du format de sérialisation

Après plusieurs expérimentations sur l'architecture de l'observatoire MeDON, nous avons ajouté la possibilité de modéliser le comportement des blocs SysML sous la forme de diagramme d'activités. Cet ajout permet de modéliser plus finement le comportement des blocs. Cette modification implique d'adapter le méta-modèle du langage SysML qui est utilisé pour représenter les modèles SysML en interne de l'environnement Role4All. La Figure 5.25 présente la modification du méta-modèle du langage SysML que nous utilisons.

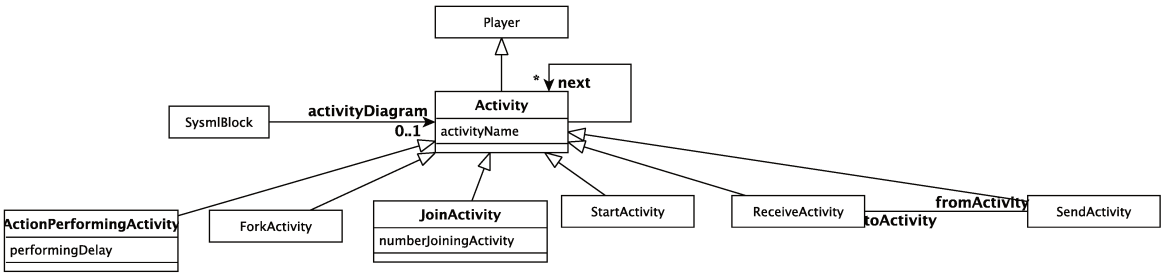


FIGURE 5.25. – Modification du méta-modèle du langage SysML pour inclure un diagramme d'activités.

Un bloc SysML possède désormais une référence vers une instance de la méta-classe Activity. Pour assurer la compatibilité avec les modèles précédemment réalisés, la référence entre un bloc SysML et une activité peut ne pas être présente tel qu'indiqué par la multiplicité *0..1* sur l'association entre la méta-classe SysmlBlock et la méta-classe Activity. Nous représentons un diagramme d'activités sous la forme d'une liste chaînée d'activité par l'intermédiaire de la référence *next* de la méta-classe Activity. Pour tenir compte de la possibilité de déclencher des activités en parallèle, la multiplicité de l'association *next* est de plusieurs (notée *).

Différents types d'activités sont disponibles. Pour représenter cela, des relations d'héritage sont définies entre les méta-classes qui définissent chaque type d'activité et la méta-classe Activity.

La méta-classe ActionPerformingActivity décrit un bloc de traitement (à prendre au sens large). Cette méta-classe possède un attribut *performingDelay* qui modélise la durée du traitement.

La méta-classe ForkActivity décrit une activité qui démarre d'autres activités en parallèle .

La méta-classe JoinActivity décrit une activité qui sert de point de synchronisation entre des chaînes d'activités démarrées en parallèle. Cette méta-classe possède un attribut *numberJoiningActivity* qui modélise le nombre de chaînes d'activités qui doivent être synchronisées par la JoinActivity.

La méta-classe StartActivity représente l'activité initiale.

La méta-classe ReceiveActivity modélise une activité de réception d'information. Le nom de cette activité correspond au port de réception. La méta-classe ReceiveActivity possède une référence vers une instance de méta-classe SendActivity qui modélise l'activité qui a envoyé l'information.

La méta-classe SendActivity modélise une activité d'envoi d'information. Le nom de cette activité correspond au port par lequel l'information est envoyée. La méta-classe SendActivity possède une référence sur une instance de ReceiveActivity.

L'ajout d'un diagramme d'activités aux blocs SysML doit aussi être pris en compte dans la sérialisation du modèle SysML. Ainsi, une nouvelle propriété peut être ajoutée au format de sérialisation comme illustré dans le Listing 5.21.

```
1 Component component.3 {
2   name: Detection
3   ports: port.3, port.4
4   subComponents:
5   activity: pathToSerializedActivityDiagram
6 }
```

Listing 5.21 – Modification de la sérialisation d'un block SysML.

Une propriété nommée activity est ajoutée au bloc de code qui sérialise un bloc SysML. La valeur de cet attribut est le chemin vers le fichier où le diagramme d'activités est sérialisé. Nous avons fait le choix de sérialiser à part les diagrammes d'activités des modèles de blocs SysML car nous souhaitons voir le langage de modélisation de diagramme d'activités comme un langage de modélisation à part. Cela permet de créer des diagrammes d'activités indépendamment de la définition d'un modèle SysML. Cependant, pour garder une compatibilité avec les formes sérialisées d'anciens modèles, cette propriété ne doit pas forcément être indiquée dans la forme de sérialisation d'un modèle SysML.

5.7.2. Impact sur le parser pour le langage de modélisation

Les diagrammes d'activités ajoutés aux blocs SysML sont eux aussi sérialisés dans le même format que le modèle SysML. Nous avons donc défini de nouveaux parsers pour analyser les diagrammes d'activités. Dans cet exemple, nous avons omis plusieurs parsers pour des raisons de lisibilité. Les équations suivantes définissent une partie des parsers définis pour analyser chaque type d'activités :

$$\text{APParser} : \text{SerializedAPActivity} \rightarrow \text{ActionPerformingActivity} \tag{5.1}$$

$$\text{forkActivityParser} : \text{SerializedFork} \rightarrow \text{SysmlFlowPort} \tag{5.2}$$

$$\text{startActivityParser} : \text{SerializedStart} \rightarrow \text{StartActivity} \tag{5.3}$$

Nous avons définis un parser par type d'activité. Ces parsers sont ensuite combinés à l'aide d'opérateurs de combinaison pour obtenir le parser d'un diagramme d'activités. L'équation 5.4 donne la définition du parser de diagramme d'activités.

$$\text{activityDiagramParser} = startParser, (APParser/forkActivityParser)* \quad (5.4)$$

Le parser de diagrammes d'activités s'attend obligatoirement à analyser la définition d'une activité de départ. Les activités suivantes peuvent être des activités de calculs ou de fork dans notre exemple qui sont analysées par leur parser respectif combiné par un opérateur de choix ordonné. Le nombre d'activité à analyser est indéfini. Dans notre exemple, nous avons omis plusieurs parsers pour des raisons de lisibilité. Dans l'implémentation concrète, le choix ordonné concerne tous les parsers pour les différents types d'activités hormis l'activité de départ.

L'ajout du parser de diagramme d'activités pour analyser une propriété des blocs SysML a un impact sur la hiérarchie de parsers qui est utilisée pour analyser les différents modèles système. La Figure 5.26 indique les parsers qui sont réellement impactés par ces modifications.

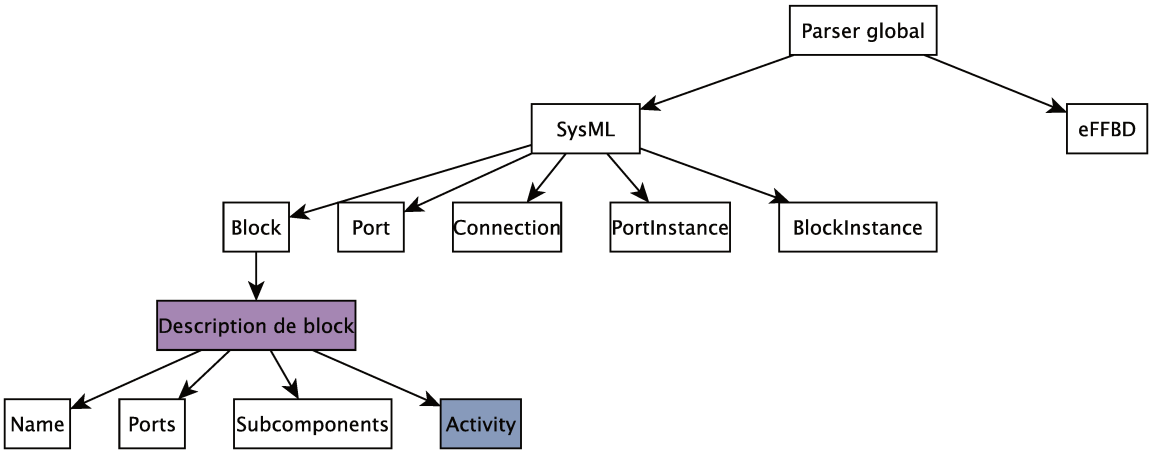


FIGURE 5.26. – Impact sur les parsers de l'ajout d'un lien vers un diagramme d'activités dans la description d'un bloc SysML.

L'ajout de l'analyse des diagrammes d'activités implique la définition d'un nouveau parser Activity indiqué en bleu sur la Figure 5.26. L'analyse des diagrammes d'activités est nécessaire pour analyser la description d'un bloc. Le parser pour analyser la description d'un bloc SysML indiqué en mauve sur la Figure 5.26 est obtenu par combinaison d'autres parsers qui analysent le nom du bloc, la liste des ports associés au bloc et la liste des sous-composants. L'ajout de l'analyse des diagrammes d'activité implique d'étendre la combinaison des parsers définies pour le parser de la description d'un bloc SysML. Tous les autres parsers ne sont pas impactés.

Grâce à la modularité apportée par les parsers combinateurs, une modification de la définition d'un langage de modélisation n'impacte pas l'ensemble des parsers définis pour analyser le langage. Seuls les parsers associés aux concepts modifiés sont impactés. Le temps de réécriture d'un parser en cas de modifications du langage de modélisation associé au parser est donc réduit.

5.7.3. Impact sur le modèle de rôles

La modification d'un langage de modélisation impacte également le modèle de rôle qui doit être associé aux différents éléments de modèles définis dans le langage modifié.

Un LeafFunctionRole permet de représenter des fonctions séquentielles. Or, la définition d'un diagramme d'activités permet d'avoir des activités qui s'exécutent en parallèle pour décrire un bloc SysML. Par conséquent, les instances des blocs SysML dont le bloc contient un diagramme d'activités ne peuvent pas jouer le rôle de LeafFunctionRole. Toutefois, en enregistrant notre contexte pour simuler des architectures fonctionnelles, nous avons automatisé l'association des rôles LeafFunctionRole avec des instances de la classe SysmlBlocInstance représentant les instances de blocs SysML. L'ajout de la définition des diagrammes d'activités implique que certains rôles ajoutés automatiquement ne l'ont pas été à bon escient. Pour remédier à ce problème, nous utilisons la capacité des rôles à pouvoir être dissociés des objets auxquels ils sont liés. Le Listing 5.22 illustre la dissociation des rôles LeafFunctionRole.

```
1 LeafFunctionRole detachFrom: SysmlBlockInstance underCondition:
2   [ :bi | (bi type class hasAttribute: #activity) and: (bi type activity isNotNil)].
```

Listing 5.22 – Dissociation des instances de LeafFunctionRole associées à des instances de SysMLBlockInstance dont le type possède un diagramme d'activités

La méthode *detachFrom:underCondition:* permet de dissocier des rôles associés à des instances d'une classe spécifique qui répondent à une condition donnée par un bloc. Cette méthode récupère tous les rôles associés à une instance de SysmlBlockInstance. La condition est testée sur l'instance de SysmlBlockInstance pour vérifier si la relation entre le rôle et l'objet doit être supprimée. Dans la condition décrite ligne 2, nous testons la présence de l'attribut *activity* dans la classe SysmlBlock associée à l'instance de SysmlBlockInstance. Ce premier test permet de gérer des instances de SysmlBlock qui ont été créées avec les classes non modifiées, c'est-à-dire avec des instances générées avant que nous ayons modifié le méta-modèle SysML dans Role4All. Un second test permet de vérifier si l'instance de SysmlBlock qui définit le type de l'instance possède un diagramme d'activités. Si les deux tests sont positifs, la relation entre le rôle et l'objet est supprimée.

Il est ensuite possible de définir de nouvelles associations de rôles pour prendre en compte les diagrammes d'activités. Nous faisons le choix d'associer un rôle LeafFunctionRole à toutes les instances d'activités peut importe leur type. Le Listing 5.23 donne le code pour réaliser cette association.

```
1 Activity plays: LeafFunctionRole.
```

Listing 5.23 – Association du rôle LeafFunctionRole aux différentes activités des diagrammes d'activités

Appeler la méthode *plays:* sur la classe Activity permet d'associer le rôle LeafFunctionRole aux instances de la classe Activity mais aussi de toutes ses sous-classes. Ainsi, les instances de tous les types d'activités sont associées avec une instance du rôle LeafFunctionRole.

Cette solution a toutefois un inconvénient. Les diagrammes d'activités sont associés aux instances de la classe SysmlBlock et non pas celles de la classe SysmlBlockInstance. Ces dernières permettent de représenter un déploiement des blocs pour créer un système. Il est donc possible d'avoir plusieurs instances de la classe SysmlBlockInstance liées à la même instance de SysmlBlock. Par conséquent, le nombre d'instances du rôle LeafFunctionRole associées à chaque instance d'Activity doit correspondre au nombre d'instances de SysmlBlockInstance. Notre code fonctionne parfaitement dans le cas où pour chaque instance de SysmlBlock il n'y a qu'une seule instance de SysmlBlockInstance associée.

Dans le cas contraire, il est nécessaire de décrire l'association de rôles pour chaque instance

d'activités. Pour cela, il faut :
- sélectionner toutes les instances de SysmlBlockInstance qui sont associées à une instance de SysmlBlock qui possède un attribut activity non nul,
- pour chaque instance sélectionnée, récupérer toutes les instances de la classe Activity liées à l'instance de SysmlBlock associée à l'instance de SysmlBlockInstance. Les instances d'Activity sont stockées dans un Set Smalltalk qui est une collection qui garantit l'unicité d'un objet rangé dans la collection,
- pour chaque instance d'Activity, associer une instance du rôle LeafFunctionRole.

Le modèle de rôle ainsi créé correspond alors aux exigences du déploiement du système.

Une modification dans la définition d'un langage de modélisation doit être prise en compte dans la définition des classes de l'environnement Role4All qui représentent le langage. La modularité apportée par les parsers combinateurs a permis de tenir compte de l'ajout des diagrammes d'activités dans la définition des blocs SysML sans nécessiter de réécrire la totalité du parser pour le langage SysML. La capacité d'un rôle à être attaché et détaché dynamiquement permet de modifier dynamiquement un modèle de rôle pour l'adapter aux nouveaux éléments qui apparaissent dans les modèles système.

5.8. Génération de documentation

Dans les sections précédentes, nous avons utilisé les rôles pour unifier des modèles systèmes définis dans des langages de modélisation différents pour créer des simulation avec divers outils de simulation. La simulation permet d'améliorer la communication entre les différents acteurs de la spécification et de la conception d'un système en permettant de voir un prototype virtuel du système fonctionner. La simulation ne remplace pas la documentation qui permet de garder une trace des choix de conception. La génération de la documentation a également besoin d'accéder aux informations contenues dans les modèles différents. Les modèles doivent donc être unifiés pour générer la documentation.

Dans cette section, nous montrons comment Role4All permet d'unifier les modèles systèmes SysML et eFFBD de l'observatoire MeDON pour permettre la génération d'une documentation. La génération de la documentation pourra se faire en parallèle de la génération des simulations.

5.8.1. Analyse du générateur de documentation

Le générateur de documentation permet de générer des pages HTML à partir des éléments de modèles. Le générateur de pages HTML doit générer trois familles de pages :
- La page d'index.
- La page décrivant l'artéfact d'ingénierie système (ex. : exigences, architecture fonctionnelle, architecture physique, etc).
- La page décrivant un élément d'un artéfact d'ingénierie système.

La page d'index comprend les liens vers les pages des différents artéfacts d'ingénierie système. La page pour chaque artéfact contient les liens vers tous les éléments de modèles définis pour cet artéfact. La page pour chaque élément de modèle contient la liste des propriétés de l'élément et leurs valeurs.

Chaque page à générer correspond à un template réalisé avec le framework Smalltalk Mustache.

Les pages HTML de la documentation sont générées à l'aide de patrons pour le moteur de templates Mustache. Ces patrons utilisent des dictionnaires de valeurs pour générer la page définitive. Les valeurs peuvent être des valeurs simples ou d'autres dictionnaires.

La génération des pages HTML se fait en deux temps. Dans un premier temps, le moteur de génération rassemble les informations contenues dans les différents éléments de modèles. Ces informations sont formatées sous la forme de dictionnaires comme spécifié dans les patrons utilisés avec le moteur de templates Mustache. La récupération des différentes informations nécessite la définition de types de rôles dédiés, d'un modèle de rôles et des adaptateurs adéquats. Dans un second temps, le moteur de templates crée les pages HTML en fonction des informations stockées dans les dictionnaires.

5.8.2. Définition du modèle de rôle et des adaptateurs dynamiques

La description du générateur de documentation guide la définition des rôles. Un seul rôle a besoin d'être défini : le rôle DocumentationElementRole. Le Listing 5.24 décrit la construction du rôle DocumentationElement.

```
1 Role named: 'DocumentationElementRole' classifiedIn: 'Document−Generation−Role'
2 DocumentationElementRole addDelegateMethod: 'owningArtefact'.
3 DocumentationElementRole addDelegateMethod: 'elementName'.
4 DocumentationElementRole addDelegateMethod: 'properties'.
```

Listing 5.24 – Définition du rôle DocumentationElementRole.

Le rôle DocumentationElementRole est créé ligne 1. Les lignes 2 à 4 permettent de lui ajouter des attributs qui serviront à la création des dictionnaires utilisés avec le template Mustache :

owningArtefact décrivant l'artéfact d'ingénierie système dans lequel l'élément est déclaré.

elementName décrivant le nom de l'élément.

properties décrivant les propriétés possédées par l'élément.

Toutes ces propriétés sont déléguées à l'adaptateur dynamique qui sera lié à la relation entre une instance du rôle DocumentationElementRole et un élément de modèles. Nous pouvons désormais instancier ce type de rôle pour créer un modèle de rôles et associer un adaptateur sur les relations entre les instances de DocumentationElementRole et des objets comme illustré Listing 5.25.

```
1 LeafFunctionRole plays: DocumentationElementRole.
2 DocumentationElementRole linkedToKind: LeafFunctionRole withAdapter: 'AdaptLfAsDocElt'.
```

Listing 5.25 – Association de rôles du type DocumentationElementRole et ajout de l'adaptateur adéquat.

Dans le cas de notre étude, toutes les instances du rôle LeafFunctionRole vont jouer le rôle de DocumentationElementRole. Sur toutes les relations entre des instances de rôles, une instance de l'adaptateur dynamique AdaptLfAsDocElt est ajoutée. Cet adaptateur doit permettre de remplir les différents dictionnaires nécessaires au moteur de template pour générer les pages HTML de la documentation.

Par exemple, pour créer la liste des propriétés d'un élément de modèle, la méthode *properties:* est implémentée au sein de l'adaptateur AdaptLfAsDocElt. Le Listing 5.26 présente une partie de l'implémentation de cette méthode.

```
AdaptLfAsDocElt>>properties: player
  |type|
  type := player functionType.
   ^({'properties'->{
    {'propertyName'->'type'.
     'hasValue'->(type isNotNil).
     'propertyValue'->(player functionType).} asDictionary.}} asDictionary.)
```

Listing 5.26 – Implémentation de la méthode déléguée properties de l'adaptateur AdaptLfAsDocElt

La méthode *properties:* permet de récupérer les propriétés de l'élément à documenter. Nous devons renvoyer un dictionnaire contenant la clé *properties*. La valeur associée à cette clé est une liste de dictionnaires. Ces derniers doivent contenir trois clés : *propertyName*, *hasValue* et *propertyValue*. Dans notre cas, nous souhaitons récupérer le type de la fonction. La valeur associée à la clé *propertyName* est *type*. Pour la valeur de la clé *hasValue*, nous testons si la valeur du type renvoyée par le rôle LeafFunctionRole n'est pas nulle. Cette valeur peut être utilisée comme valeur pour la clé *propertyValue*.

Un visiteur est défini pour créer les différents dictionnaires qui seront utilisés par le moteur de template. Ce visiteur récupère toutes les instances du rôle DocumentationElementRole par l'intermédiaire de la méthode *retrieveInstances* comme illustré dans le Listing 5.27.

```
DocumentationElementRole retrieveInstances.
```

Listing 5.27 – Utilisation de la méthode retrieveInstances pour les instances du rôle DocumentationElementRole

Il est ensuite possible d'appeler les méthodes *owningArtefact*, *elementName* et *properties* sur chaque instance du rôle DocumentationElementRole. Les résultats de ces appels sont stockés dans des dictionnaires qui serviront de point d'entrée pour la génération des fichiers HTML.

Au sein de Role4All, un même objet peut jouer plusieurs rôles de types différents. En plus des rôles associés à la simulation basée événements discrets, nous avons fait jouer aux rôles d'architecture fonctionnelle des rôles de documentation. Nous avons donc été capable d'extraire des modèles SysML et eFFBD des informations sur l'architecture fonctionnelle et de l'utiliser dans un contexte autre que la simulation pour créer une documentation au format HTML de l'architecture fonctionnelle. La création de la documentation n'a pas nécessité de modifier la création des simulations. Ainsi, les deux activités peuvent être menées en parallèle et sans interférer l'une avec l'autre. L'utilisation de l'environnement Role4All n'est ainsi pas limitée aux seuls besoins de la simulation de modèles.

5.9. Conclusion

Dans le cadre du projet d'observatoire sous-marin MeDON, des équipes de l'IFREMER, du Plymouth Marine Laboratory, de l'ENSTA Bretagne, de l'université de Plymouth, d'Océanopolis et de l'aquarium de Plymouth. Pour faciliter la communication entre les différentes équipes, nous avons d'utiliser des modèles et de simuler ces modèles. Cependant, les différentes équipes ont des cultures techniques différentes et travaillent sur des aspects du système nécessitant des une expertise dans des domaines séparés. Par conséquent, les différentes équipes ont travaillé

avec des langages de modélisation propres à chaque domaine d'expertise.

Afin de pouvoir créer les simulations à partir des modèles différents des équipes du projet MeDON, il est nécessaire de pouvoir unifier les différents modèles en un modèle de simulation. Cependant, l'observatoire sous-marin MeDON est un système de systèmes qui est amené à évoluer. Cette évolution concerne le système en lui-même mais aussi les langages et les outils de modélisation qui servent à sa conception ainsi que les outils de simulation qui servent à simuler les modèles. Nous devons donc anticiper ces évolutions.

Nous avons utilisé notre environnement Role4All pour permettre d'unifier des modèles SysML et eFFBD de l'architecture fonctionnelle de l'observatoire MeDON. Role4All est fondé sur l'utilisation du concept de rôle pour la modélisation d'un point de vue sur des modèles. Role4All permet de définir des types de rôles qui sont indépendants des langages de modélisation et des outils de simulation. Cette indépendance permet d'unifier des modèles différents mais également d'utiliser les mêmes rôles pour interfacer les modèles avec des outils de simulation différents. Il est donc possible de s'adapter à l'utilisation de nouveaux outils de simulation ou bien au replacement d'un outil de simulation par un outil équivalent sans modifier les modèles de rôles utilisés pour unifier les modèles système.

Les rôles (instances de types de rôles) définis au sein de Role4All peuvent être attachés à des éléments de modèles à la volée. Les rôles peuvent être détachés des objets auxquels ils sont attachés également à la volée. Ainsi, il est possible de faire évoluer un modèle de rôles existant pour l'adapter à un nouveau point de vue d'unification de modèles différents. Le modèle de rôle peut être adapté à la volée aux changements de sémantique des éléments de modèles causés par un changement de la définition du langage de modélisation.

De plus, Role4All permet aux éléments de modèles de jouer plusieurs rôles. Il est donc possible de définir plusieurs points de vues sur les mêmes modèles et d'utiliser ces points de vue en parallèle. Ainsi, plusieurs schéma d'unification des modèles peuvent être employés en même temps sur les mêmes modèles.

Enfin, Role4All est un outil complètement indépendant des outils de modélisation système. Les modèles sont importés à l'aide d'importateurs modulaires définis par l'intermédiaire de parsers combinateurs. La modularité apportée par les parsers combinateurs permet à Role4All de s'adapter à l'ajout de nouveaux langages de modélisation tout comme à la modification partielle de la définition d'un langage de modélisation. Grâce aux parsers combinateurs, il n'est pas nécessaire de modifier complètement l'importateur, seul le parser concerné par le changement dans la définition du langage est à modifier.

Grâce à l'indépendance des rôles par rapport aux outils de modélisation et de simulation, Role4All a permis d'unifier les modèles d'architecture fonctionnelle de MeDON exprimés en SysML et eFFBD pour les simuler avec un outil d'exécution de modèle interne ainsi qu'avec le simulateur de réseau par événements discrets OMNeT++. L'indépendance des rôles par rapport aux outils de simulation a permis de remplacer OMNeT++ par un autre simulateur par événements discrets sans avoir à modifier les modèles système et le modèle de rôles associé. Enfin, les importateurs modulaires et la possibilité de redéfinir à la volée un modèle de rôles a permis d'adapter l'unification des modèles SysML et eFFBD à l'ajout dans les modèles SysML de diagrammes d'activités qui n'étaient pas gérés dans un premier temps.

6 Conclusion

6.1. Bilan du travail

6.1.1. Analyse des problématiques

Un système de systèmes est un système dont les constituants sont eux-mêmes des systèmes et qui possède les caractéristiques suivantes :

— les sous-systèmes sont gérés indépendamment,
— les sous-systèmes fonctionnent indépendamment,
— les sous-systèmes évoluent,
— les sous-systèmes collaborent,
— les sous-systèmes sont distribués géographiquement.

Ces caractéristiques rendent la conception et le développement d'un système de systèmes complexes. En effet, différentes équipes aux compétences diverses sont impliquées dans la conception et le développement d'un système de systèmes. Au niveau de la conception d'un système ou d'un système de systèmes, l'utilisation de modèles permet de définir des vues abstraites sur le système. Ces abstractions facilitent la communication entre les experts de domaines différents en cachant les détails techniques non essentiels pour la compréhension du système pour un point de vue donné. Les modèles définissent virtuellement le système en cours de conception. Les modèles sont ainsi des prototypes virtuels du système.

Ces prototypes virtuels (et donc les modèles) peuvent être analysés sans avoir à réaliser concrètement des éléments du système. Le prototypage virtuel peut être utilisé à toutes les étapes de la conception d'un système y compris celles en amont comme la définition de l'architecture fonctionnelle. La simulation est une technique d'analyse des prototypes virtuels.

Hétérogénéité Cependant, les outils de simulation utilisent des modèles qui leurs sont propres. Il est nécessaire de garantir la cohérence entre les modèles conceptuels du système et les modèles de simulation. Il est également nécessaire de permettre aux experts des différents domaines de continuer à utiliser leurs outils de modélisation sans avoir à refaire le travail de modélisation, déjà réalisé au niveau conceptuel, pour définir les simulations. Les modèles de simulation doivent pouvoir être créés à partir des modèles systèmes conceptuels.

Si l'utilisation de modèles facilite la communication entre les différentes équipes qui travaillent sur la spécification et la conception du système de systèmes, la gestion indépendante des sous-systèmes conduit chaque équipe à travailler indépendamment des autres équipes. Chaque

équipe possède donc ses propres outils et méthodes. Ainsi, chaque équipe utilise le langage de modélisation le plus approprié pour son domaine d'expertise.

Evolutivité L'évolution des constituants est une seconde caractéristique des systèmes de systèmes qui pèse sur l'utilisation de la modélisation et de la simulation. L'un des facteurs d'évolution du système de systèmes est la modification du besoin au cours du temps. Ceci signifie que le système a une durée de vie qui ne doit pas être négligée. La durée de vie du système de systèmes a également un impact sur les langages de modélisation et les outils de simulation utilisés. En effet, au cours du temps, la définition des langages de modélisation va évoluer tout comme celle des modèles utilisés par les outils de simulation. La durée de vie des systèmes de systèmes requiert de conserver la capacité de manipuler des modèles systèmes réalisés dans des langages de modélisation anciens.

La modélisation et la simulation dans un contexte système de systèmes nécessitent donc :

A d'être capable de manipuler des modèles systèmes de plusieurs natures :

- **1** sans modifier les méta-modèles qui définissent les langages de modélisation,
- **2** sans modifier les modèles à manipuler,
- **3** en travaillant de manière identique avec des concepts à sémantique équivalente dans des langages de modélisation différents,
- **4** en étant capable d'ajouter de l'information pour faire les liens entre les différents modèles.

B d'être capable de manipuler des spécifications de simulation différentes :

- **5** en travaillant avec différents outils de simulation utilisant un même formalisme de simulation ou des formalismes différents,

C de tenir compte de l'évolution des outils :

- **6** en gérant les dialectes des langages de modélisation et de simulation.
- **7** en permettant l'ajout de nouveaux langages de modélisation et de simulation.
- **8** en continuant à supporter des modèles dont les outils utilisés pour les créer sont obsolètes et retirés du service.

Parmi les méthodes identifiées dans la littérature sur la création de modèles de simulation à partir de modèles systèmes, nous retenons l'unification et la fédération. L'unification et la fédération permettent de créer des simulations à partir de multiples modèles d'entrée, même si ces derniers sont exprimés dans des langages de modélisation différents. L'unification de modèles consiste à définir un langage dédié qui servira à créer des modèles pivots entre les modèles d'entrée et les modèles de simulation. La fédération crée des liens sémantiques entre des éléments de différents modèles qui représentent le même objet, qu'ils s'agissent de modèles systèmes ou de modèles de simulation. Cependant, ces approches, basées sur les types des éléments de modèles, peinent à tenir compte des évolutions des langages de modélisation ou des outils de simulation. En effet, un changement dans la définition d'un type au sein d'un langage de modélisation force à réécrire soit les transformations de modèles utilisées pour l'unification, soit les réalisations des liens sémantiques utilisés pour la fédération.

Les rôles sont un concept de modélisation issu du monde des bases de données. Les rôles permettent de modéliser des vues sur des données de types différents. Les rôles peuvent être associés aux données et en être dissociés dynamiquement. La possibilité de créer des vues indépendamment des types de données peut être utilisée pour manipuler des modèles hétérogènes.

La capacité d'associer ou de dissocier des rôles permet, quant à elle, de tenir compte des évolutions des langages de modélisation.

6.1.2. Role4All : les rôles pour créer un point de vue homogène sur des modèles hétérogènes

Nous avons utilisé les rôles pour permettre d'interfacer des modèles systèmes avec des outils de simulation comme illustré Figure 6.1. Nous sommes capables de définir des types de rôles. Ces types de rôles sont créés d'après le formalisme de simulation de l'outil de simulation utilisé pour analyser les modèles systèmes. Les types de rôles définissent les services fournis par les rôles, qui sont des instances des types de rôles, aux outils de simulation qui vont extraire des informations des modèles du système. Les rôles peuvent être associés à des éléments de modèles indépendamment de leur type. A l'instar des approches par unification, le modèle de rôles, les rôles et leur association avec des éléments de modèles, est un modèle pivot homogène entre des outils de simulation et des modèles systèmes exprimés dans des langages de modélisation hétérogènes. Dans nos travaux, nous avons nous avons utilisé des modèles de systèmes disjoints, c'est-à-dire qu'un même objet n'était pas modélisé dans deux langages de modélisation distincts. Le besoin de faire de la fédération de modèle n'était pas présent.

Pour supporter cette approche originale, nous avons créé Role4All, un environnement dédié à la création de modèles de rôles. Cet environnement s'articule avec les outils de modélisation système par couplage lâche. Ceci garantit son indépendance par rapport aux outils de modélisation système. Cette indépendance assure que l'environnement reste opérationnel lorsque les outils de modélisation deviennent obsolètes. Toutefois, il faut avoir la capacité d'intégrer les modèles réalisés au sein de Role4All. Cela est réalisé par l'intermédiaire d'importateurs modulaires construits à l'aide de parsers combinateurs. Les parsers combinateurs permettent de combiner des parsers et donc d'écrire des parsers réutilisables et plus facilement maintenables.

Un langage dédié a été défini pour créer des modèles de rôles. Ce langage, R2DL (Role4All Role Description Language), est indépendant des outils et des langages de modélisation. L'utilisation des rôles n'implique pas la modification des méta-modèles qui définissent les langages de modélisation système. Les rôles sont des entités à part entière qui possèdent leurs propres propriétés.

L'adaptation d'un élément de modèle à un rôle est réalisée par un adaptateur spécifique. Par conséquent, les rôles n'exigent pas de modifier les modèles systèmes. Des rôles du même type peuvent être associées à des éléments de modèle définis dans des langages de modélisation différents. Ainsi, la manipulation des concepts de sémantique équivalente définis dans des langages de modélisation système différents est transparente. Le modèle de rôles associé aux éléments de modèle permet de s'abstraire du type des éléments pour se concentrer sur leur signification dans le cadre de la génération de modèles de simulation à l'image d'un modèle pivot dans le cadre de l'unification de modèles.

L'adaptation d'un élément de modèle à un rôle pour une utilisation spécifique est réalisée par un adaptateur. Pour utiliser différents outils de simulation de la même famille, un seul modèle de rôle, qui définit le point de vue des outils de simulation, doit être défini. La gestion des différences entre les outils de simulation est réalisée en changeant les adaptateurs. Ainsi, contrairement aux approches par unification classiques, la définition correspondance entre un concept du modèle pivot (dans notre cas un type de rôle) et un concept d'un modèle d'entrée est distincte de la définition de l'interprétation de l'élément du modèle d'entrée. Une approche par unification classique repose sur l'usage de règles de transformation de modèles. Ces règles

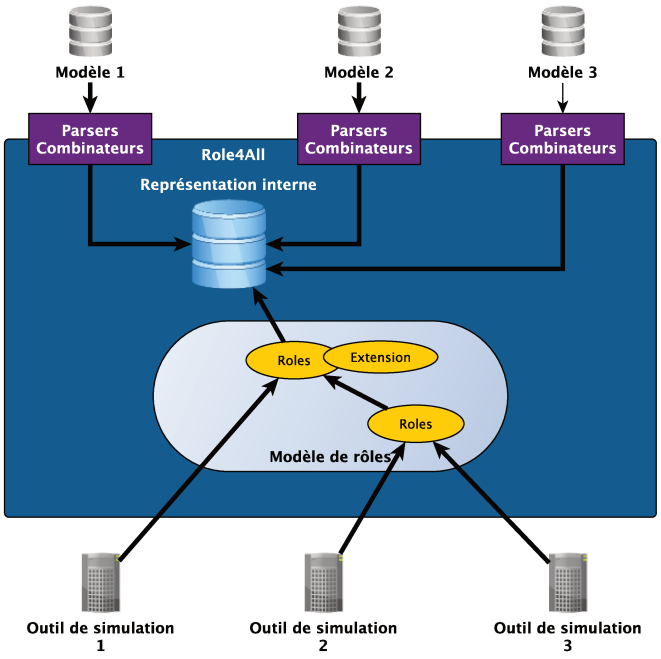


FIGURE 6.1. – Structure de l'environnement Role4All

servent à créer les éléments du modèle pivot. Elles servent également à donner des valeurs aux propriétés des éléments du modèle pivot en fonction de celles des éléments des modèles d'entrée. Ces règles deviennent complexes lorsqu'un même élément de modèle a plusieurs équivalents dans le modèle pivot. En séparant la définition de la correspondance entre modèle pivot et modèle système de la définition de l'interprétation des modèles système, l'utilisation de rôles et d'adaptateurs simplifie l'écriture des interprétations des éléments de modèles d'entrée et facilite la réutilisation d'un même modèle de rôles avec plusieurs outils de simulation.

La possibilité d'avoir des rôles différents associés à un même objet permet de gérer l'utilisation de types de simulateurs différents et de pouvoir travailler sur l'ensemble des phases d'ingénierie système. Il est donc possible de définir plusieurs schémas d'unification de modèles qui pourront accéder aux informations contenues dans les éléments de modèles de manière concurrente.

Dans les approches par unification classique, le schéma d'unification est contenu dans les règles de transformation de modèles entre les modèles d'entrée et le modèle pivot. En cas de changement dans un des modèle d'entrée, il est nécessaire de réécrire les règles de transformation de modèles associées puis de les rejouer pour recréer un modèle pivot complet. Les rôles quant à eux peuvent être associés ou dissociés des éléments de modèles à la volée. Dans Role4All, nous avons également permis de modifier la définition des types de rôles à la volée. Par conséquent, le modèle de rôles qui définit le schéma d'unification des modèles peut être modifié à la volée et uniquement aux endroits nécessaires pour tenir compte des modifications dans les langages de modélisation ou dans les outils de simulation.

Les parsers combinateurs utilisés pour créer les importateurs modulaires permettent de limiter l'impact des modifications de la définition d'un langage de modélisation sur l'écriture des importateurs modulaires. Ils facilitent la gestion des évolutions des importateurs. L'utilisation de l'introspection permet, quant à elle, de définir de nouveaux parsers et de les intégrer faci-

lement à l'environnement Role4All. Enfin, l'utilisation de parsers combinateurs permet d'être indépendant des outils de modélisation et donc de conserver la capacité de lire des modèles qui ont été réalisés avec des outils qui ne sont plus disponibles.

6.1.3. Limitations de Role4All

Malgré les capacités offertes par l'environnement Role4All, celui-ci possède certaines limitations. Tout d'abord, la littérature sur les rôles indique que l'association d'un rôle avec un objet dépend des relations que l'objet entretient avec d'autres objets. Les rôles sont associés à, ou dissociés des objets dynamiquement. Cependant, dans Role4All, les rôles servent à modéliser le point de vue qu'un modeleur souhaite avoir sur des éléments de modèles. L'association entre rôles et objets, dans Role4All, dépend donc de ce point de vue. Par conséquent, l'association et la dissociation entre rôles et objets est essentiellement manuelle. Nous avons défini des règles qui permettent d'automatiser l'association de rôles avec des objets. Cependant, en cas de modifications dans la structure des langages de modélisation système pris en entrée, certaines règles peuvent être invalidées. Actuellement, il est nécessaire de modifier ces règles manuellement en accédant directement aux règles à modifier. Il est donc nécessaire de fournir des méthodes d'accès de plus haut niveau pour modifier les règles définies.

Par rapport aux approches par unification qui utilisent un méta-modèle pour définir les modèles pivots, les modèles de rôles dans Role4All n'ont pas de structure autre que des relations hiérarchiques et la possession de propriétés. La compréhension du modèle de rôles est rendue difficile par cette absence de structure. En effet, il est difficile d'identifier l'objectif recherché par l'utilisation du modèle de rôles. Pour résoudre cette limitation, il est nécessaire d'améliorer la structure des contextes en permettant de lui ajouter l'information nécessaire pour caractériser son objectif. Il est également nécessaire d'identifier pour chaque rôle, et pas seulement pour chaque type de rôle, le contexte auquel il est attaché. Ce lien permettrait de mieux identifier l'objectif dans lequel s'inscrit chaque rôle. Cette modification a un impact non négligeable sur l'instanciation des rôles. Cet impact reste à chiffrer.

Enfin, l'utilisation de Role4All a également des limites. Actuellement, en cas de modification d'un modèle système précédemment importé dans Role4All, il est nécessaire de supprimer tous les éléments de modèle associés au modèle système modifié ainsi que tous les rôles associés pour pouvoir importer à nouveau le modèle dans Role4All. Un mécanisme de modifications par incrément des modèles déjà importés dans Role4All peut être mis au point. Ce mécanisme devra reposer sur :

— la capacité à faire la différence entre deux graphes d'objets représentant deux versions du même modèle,
— l'automatisation de la création de relations de composition entre rôles,
— l'automatisation de la dissociation et l'association de rôles.

L'utilisation de Role4All doit également être étendue. Nous n'avons utilisé Role4All uniquement dans le contexte de la simulation d'architectures fonctionnelles. Or, nous devons pouvoir utiliser les rôles à toutes les étapes de la conception d'un système ou d'un système de systèmes. Il est donc nécessaire d'investiguer l'utilisation de Role4all pour l'analyse d'exigences ainsi que pour la simulation d'architectures physiques.

6.2. Perspectives à court terme

Afin de traiter la limitation sur la gestion des règles d'association de rôles, nous devons fournir des méthodes d'accès aux règles qui sont enregistrées dans Role4All. Pour cela, nous devons être capable d'identifier de manière unique chaque règle. Chaque règle d'association de rôles contient le type de rôle à associer, le type de l'élément de modèles qui doit jouer le rôle et une condition d'association. Cependant, dans deux contextes différents, la même règle peut être utilisée. Il faut, au moins, ajouter aux règles une référence vers le contexte dans laquelle elle est définie. Une fois qu'une règle est identifiée, il est possible de la modifier. Cependant, il faut mener une réflexion sur l'automatisation de cette modification. La création de règles d'association de rôles peut être faite lors de l'appel aux instructions R2DL qui associe rôle et éléments de modèle. Il serait possible de reprendre le même principe avec les appels aux instructions R2DL pour la dissociation des rôles. L'intérêt de l'automatisation doit être vérifié par rapport à une approche qui consiste à supprimer tous les éléments de modèles et à les recréer pour leur associer de nouveaux rôles.

Pour améliorer l'identification de l'objectif du modèle de rôles, il est nécessaire d'étendre la définition du langage de rôles que nous avons proposé. L'objectif du modèle de rôles est défini par le contexte qui a servi à instancier les rôles. Les rôles devraient ainsi posséder une référence vers le contexte qui a servi à leur création. Cette référence doit être ajoutée au moment de la création des rôles. Lorsque la création de l'instance est réalisée manuellement, la référence peut également être ajoutée manuellement. Si l'instance est créée par l'intermédiaire d'une règle d'association de rôles, alors la référence doit être ajoutée par la règle. Ce dernier point augmente l'intérêt pour ajouter une référence au contexte dans les règles d'association de rôles.

La mise en place d'un système de gestion de versions des modèles permettrait d'obtenir un mécanisme de mises à jour incrémentales des modèles dans Role4All. Il s'agit d'avoir la capacité d'identifier, dans les modèles d'entrée, les modifications réalisées ainsi que leur nature (ajout / suppression d'élément, renommage). Une fois identifiées, les modifications peuvent être répercutées sur la représentation interne des modèles de Role4All.

Les problèmes précédents sont des problèmes qui peuvent être résolus en étendant la définition du langage de rôles. Cependant, les caractéristiques d'hétérogénéité des modèles systèmes et des outils de simulation ouvrent des perspectives sur le long terme.

6.3. Perspectives à moyen terme

Dans le cadre de la conception de systèmes et de systèmes de système, la collaboration entre les équipes en charge de la conception des constituants est un élément clé du succès du projet. L'intégration de Role4All dans des processus collaboratifs pose donc question. Une étude doit être réalisée pour permettre d'intégrer Role4All dans les processus actuels d'ingénierie système. Il s'agit notamment de pouvoir utiliser les capacités de création de prototypes virtuels fournis par Role4All durant toutes les phases des processus techniques du cycle d'ingénierie système. Il s'agit également d'évaluer les bénéfices de Role4All lors de réunions d'architecture pour la recherche de compromis.

Les problématiques de collaboration peuvent aussi être, en partie, traitées en utilisant une approche chaîne d'outils. Dans ce cadre, une réflexion doit être menée sur l'intégration de Role4All dans une chaîne d'outils. Un premier point à étudier est l'intégration automatisée de

nouveaux méta-modèles au sein de Role4All. Afin de garder une homogénéité dans Role4All, les parsers combinateurs pourraient être utilisés pour analyser les méta-modèles à importer et créer les représentations internes à Role4All correspondantes aux concepts définis dans les méta-modèles.

L'utilisation d'une chaîne d'outil concerne non seulement la collaboration mais aussi l'utilisation d'outils différents. Dans notre cas, il s'agit d'outils de modélisation. Comme les modèles que nous avons utilisés ne modélisaient pas les mêmes objets, nous avons adopté une approche par unification pour gérer les différents modèles créés à l'aide de ces outils. Or, les propriétés des rôles permettraient de faire de la fédération de modèles. L'étude de l'utilisation des rôles, dans le cadre de la fédération de modèles, est donc une perspective. Il s'agit notamment d'étudier les propriétés des rôles à utiliser par rapport à celles que nous avons choisi pour l'unification et déterminer s'il existe des conflits entre les deux ensembles de propriétés. Il est également nécessaire de déterminer les modifications à apporter au méta-modèle des rôles que nous avons défini pour pouvoir utiliser les rôles pour faire de la fédération de modèles. Une fois qu'une réponse à ces questions aura été apportée, il s'agira de définir les cadres d'utilisation des rôles pour l'unification et pour la fédération. Par exemple, quelle approche faut-il adopter pour obtenir du *Model at runtime* ?

6.4. Perspectives à long terme

Sur le long terme, il s'agira de pérenniser l'utilisation de Role4All en étendant ses domaines d'utilisation. Dans un premier temps, il s'agit d'étudier l'utilisation de Role4All pour l'analyse des besoins utilisateurs. Dans le processus d'ingénierie système, les besoins du client sont capturés et retraduits en exigences pour le système. Role4All pourrait être utilisé pour analyser la consistance des exigences par rapport aux besoins exprimés. Dans ce cas, il est nécessaire de définir un modèle pour capturer les besoins et un modèle pour formaliser les exigences. Il serait également possible de définir des simulations à partir des modèles des exigences pour permettre de vérifier la compréhension du besoin du client.

Role4All devrait également être utilisé pour permettre de vérifier les architectures physiques. Or, dans le cas des architectures physiques, il faut être capable de simuler des systèmes électroniques, des systèmes mécaniques et, potentiellement intégrer des humains ou du matériel dans la boucle. Il s'agit donc d'être capable de créer des simulations distribuées avec la possibilité d'intégrer des simulations *Man In the Loop* ou *Hardware In the Loop*. Les différentes simulations doivent pouvoir être interopérables pour garantir la cohérence de la simulation globale par rapport à l'architecture du système.

6.5. Conclusion

La complexité des systèmes et des systèmes de systèmes oblige à utiliser des abstractions pour les concevoir, en utilisant par exemple des modèles (approche Model-Based Systems Engineering). Pour garantir la qualité de la conception d'un système ou d'un système de systèmes, les modèles doivent être vérifiés à toutes les phases de la conception, y compris dans les phases amonts du processus d'ingénierie système. Des techniques de simulation peuvent être utilisées pour analyser les modèles et les vérifier par rapport aux besoins des clients. Cependant, la simulation utilise des formalismes, et donc des modèles, qui sont différents de ceux, plus conceptuels,

utilisés pour la modélisation de système. De plus, la modélisation de systèmes ou de systèmes de systèmes repose sur l'utilisation de plusieurs langages de modélisation. Les modèles exprimés dans ces différents langages sont utilisés pour la simulation. Le problème du passage des modèles conceptuels réalisés dans les différents langages de modélisation système à une simulation est problématique.

Des approches par intégration ou par unification ou par fédération de modèles ont été proposées pour permettre le passage de différents modèles conceptuels vers un modèle de simulation. Cependant, ces approches sont basées sur la notion de type et peinent à s'adapter aux modifications de la définition des langages de modélisation.

Nous proposons une approche originale fondée sur l'utilisation du concept de rôle pour modéliser des points de vue sur les modèles en fonction des formalismes de simulation. Les rôles peuvent être utilisés sur des éléments de modèles indépendant de leur type et donc indépendamment du formalisme de modélisation utilisé pour modéliser l'élément. Les rôles peuvent être associés et dissociés des éléments de modèles ce qui permet de modifier le point de vue modélisé en fonction des modifications dans la définition des langages de modélisation ou des formalismes de simulation. Nous avons sélectionné un ensemble de propriétés des rôles pour définir un langage permettant de créer des types de rôles, d'instancier des rôles et d'associer et dissocier les rôles. L'ensemble a fait l'objet d'une implémentation en Smalltalk qui nous a permis d'utiliser notre approche sur des modèles de l'observatoire sous-marin MeDON. Nous avons été capables de simuler des modèles SysML et eFFBD à l'aide d'un simulateur que nous avions développé ainsi qu'avec deux outils de simulation basé événements discrets. Nous avons également pu utiliser la possibilité dissocier les rôles pour suivre une évolution dans notre usage des modèles SysML en ajoutant à certains blocs SysML un diagramme d'activité.

L'approche proposée possède des opportunités d'évolution riches et variées. La fédération de modèles et la création de simulation pour la vérification d'architectures physiques font parties de ces perspectives.

A

Role4All utilisé en parallèle de Pharo

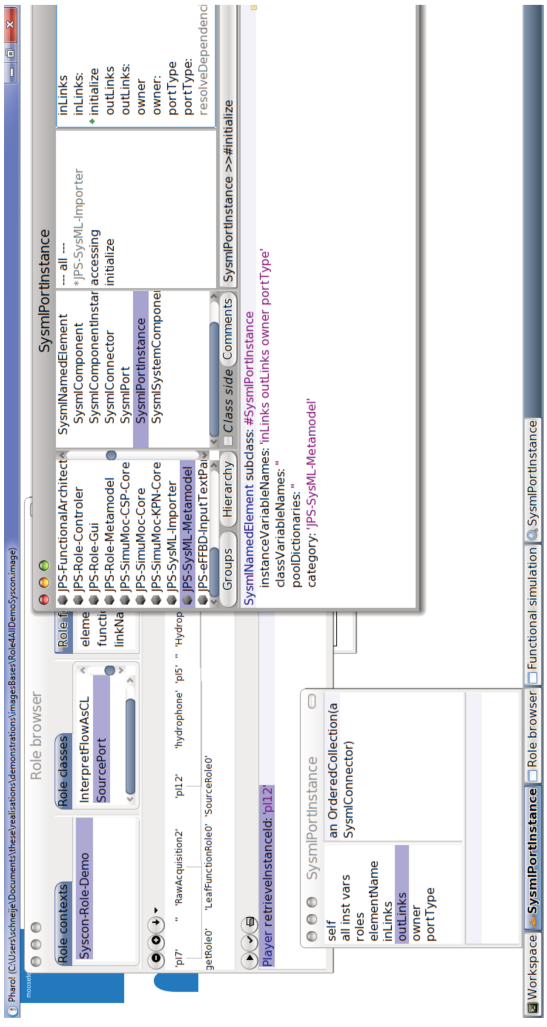


FIGURE A.1. – Role4All peut être utilisé en même temps que tous les outils Smalltalk natifs disponibles dans l'environnement Pharo

B

Modèle SysML sérialisé pour les hydrophones

```
SysmlSystem system.1 {
name: Hydrophones
components:
connectors:
}

Component component.1 {
name: hydrophone
ports: port.1
subComponents: component.2, component.3
}

Component component.2 {
name: RawAcquisition
ports: port.2
subComponents:
}

Component component.3 {
name: Detection
ports: port.3, port.4
subComponents:
}

Port port.1 {
name: outHydrophone
isBehavioral: false
}

Port port.2 {
name: outRawAcquisition
isBehavioral: true
```

```
}

Port port.3 {
name: inRawAcquisition
isBehavioral: true
}

Port port.4 {
name: outDetection
isBehavioral: true
}

ComponentInstance componentInstance.1 {
name: DataAcquisition1
type: component.1
ports: portInstance.1
subComponents: componentInstance.2, componentInstance.3
connectors: link.1, link.2
}

ComponentInstance componentInstance.2 {
name: RawAcquisition1
type: component.2
ports: portInstance.2
subComponents:
connectors:
}

ComponentInstance componentInstance.3 {
name: Detection1
type: component.3
ports: portInstance.3, portInstance.4
subComponents:
connectors:
}

ComponentInstance componentInstance.4 {
name: DataAcquisition2
type: component.1
ports: portInstance.5
subComponents: componentInstance.5, componentInstance.6
connectors: link.3, link.4
}

ComponentInstance componentInstance.5 {
name: RawAcquisition2
type: component.2
```

```
ports: portInstance.6
subComponents:
connectors:
}

ComponentInstance componentInstance.6 {
name: Detection2
type: component.3
ports: portInstance.7, portInstance.8
subComponents:
connectors:
}

ComponentInstance componentInstance.7 {
name: DataAcquisition3
type: component.1
ports: portInstance.9
subComponents: componentInstance.8, componentInstance.9
connectors: link.5, link.6
}

ComponentInstance componentInstance.8 {
name: RawAcquisition3
type: component.2
ports: portInstance.10
subComponents:
connectors:
}

ComponentInstance componentInstance.9 {
name: Detection3
type: component.3
ports: portInstance.11, portInstance.12
subComponents:
connectors:
}

PortInstance portInstance.1 {
name: pI1
type: port.1
inLinks: link.2
outLinks:
}

PortInstance portInstance.2 {
name: pI2
type: port.2
```

```
inLinks:
outLinks: link.1
}

PortInstance portInstance.3 {
name: pI3
type: port.3
inLinks: link.1
outLinks:
}

PortInstance portInstance.4 {
name: pI4
type: port.4
inLinks:
outLinks: link.2
}

PortInstance portInstance.5 {
name: pI5
type: port.1
inLinks: link.4
outLinks:
}

PortInstance portInstance.6 {
name: pI6
type: port.2
inLinks:
outLinks: link.3
}

PortInstance portInstance.7 {
name: pI7
type: port.3
inLinks: link.3
outLinks:
}

PortInstance portInstance.8 {
name: pI8
type: port.4
inLinks:
outLinks: link.4
}

PortInstance portInstance.9 {
```

```
name: pI9
type: port.1
inLinks: link.6
outLinks:
}

PortInstance portInstance.10 {
name: pI10
type: port.2
inLinks:
outLinks: link.5
}

PortInstance portInstance.11 {
name: pI11
type: port.3
inLinks: link.5
outLinks:
}

PortInstance portInstance.12 {
name: pI12
type: port.4
inLinks:
outLinks: link.6
}

Connector link.1 {
source: portInstance.2
target: portInstance.3
}

Connector link.2 {
source: portInstance.4
target: portInstance.1
}

Connector link.3 {
source: portInstance.6
target: portInstance.7
}

Connector link.4 {
source: portInstance.8
target: portInstance.5
}
```

```
Connector link.5 {
source: portInstance.10
target: portInstance.11
}

Connector link.6 {
source: portInstance.12
target: portInstance.9
}
```

C

Modèle eFFBD sérialisé pour le système d'informations de l'observatoire MeDON

```
EffbdSystem observatoryIt.1 {
name: observatory
branch: b.1
}

Branch b.1 {
elements: and.1
}

And and.1 {
branches: b.2, b.3, b.4, b.5
}

Branch b.2 {
elements: loop.1
}

Branch b.3 {
elements: loop.2
}

Branch b.4 {
elements: loop.3
}

Branch b.5 {
elements: loop.4
}

Loop loop.1 {
branch: b.6
```

```
}

Branch b.6 {
elements: function.1
}

Function function.1 {
name: ReadEntry1
inputs:
outputs: flow.1
branch:
}

Flow flow.1 {
name: dataEntry1
source: function.1
target: function.4
isTrigger: true
}

Loop loop.2 {
branch: b.7
}

Branch b.7 {
elements: function.2
}

Function function.2 {
name: ReadEntry2
inputs:
outputs: flow.2
branch:
}

Flow flow.2 {
name: dataEntry2
source: function.2
target: function.4
isTrigger: true
}

Loop loop.3 {
branch: b.8
}

Branch b.8 {
```

```
elements: function.3
}

Function function.3 {
name: ReadEntry3
inputs:
outputs: flow.3
branch:
}

Flow flow.3 {
name: dataEntry3
source: function.3
target: function.4
isTrigger: true
}

Loop loop.4 {
branch: b.9
}

Branch b.9 {
elements: function.4
}

Function function.4 {
name: ComputeDTOA
inputs: flow.1, flow.2, flow.3
outputs:
branch:
}
```

Bibliographie

[1] ADEVS. Adevs website. http://web.ornl.gov/~1qn/adevs/. 22, 153

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 137

[3] Sherman R Alpert, Kyle Brown, and Bobby Woolf. *The design patterns Smalltalk companion*. Addison-Wesley Longman Publishing Co., Inc., 1998. 136, 150

[4] Charles W Bachman and Manilal Daya. The role concept in data models. In *Proceedings of the third international conference on Very large data bases-Volume 3*, pages 464–476. VLDB Endowment, 1977. 35

[5] Ian Bailey. Reorganising modaf & naf. Technical report, Model Futures, 2013. 18

[6] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Lulu. com, 2013. 133

[7] A Biswas, J Hayden, MS Phillips, KB Bhasin, C Putt, and T Sartwell. Applying dodaf to nasa orion mission communication and navigation architecture. In *Aerospace Conference, 2008 IEEE*, pages 1–9. IEEE, 2008. 17

[8] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Model Driven Architecture*, pages 17–32. Springer, 2005. 26, 31, 32

[9] John Boardman and Brian Sauser. System of systems-the meaning of of. In *System of Systems Engineering, 2006 IEEE/SMC International Conference on*, pages 6–pp. IEEE, 2006. 10

[10] Frédéric Boulanger, Daniel Krob, Gérard Morel, and Jean-Claude Roussel, editors. *Proceedings of the Poster Workshop at the 2014 Complex Systems Design & Management International Conference co-located with 5th International Conference on Complex System Design & Management (CSD&M 2014), Paris, France, November 12th, 2014*, volume 1234 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014. 135, 189

[11] Henry Broodney, Michael Masin, Evgeny Shindin, Uri Shani, Roy Kalawsky, Demetrios Joannou, Yingchun Tian, Antara Bhatt, and Imad Sanduka. Leveraging domain expertise in architectural exploration. In *Complex Systems Design & Management*, pages 87–103. Springer, 2015. 16

[12] Alan W Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004. 16

[13] Philipp Bunge. Scripting browsers with glamour. *Master's thesis, University of Bern*, 2009. 108

[14] Centre National de Ressources Textuelles et Lexicales. Définition de modélisation. http://www.cnrtl.fr/definition/mod%C3%A9lisation. 16

[15] Deniz Cetinkaya, Alexander Verbraeck, and Mamadou D Seck. Mdd4ms: a model driven development framework for modeling and simulation. In *Proceedings of the 2011 Summer Computer Simulation Conference*, pages 113–121. Society for Modeling & Simulation International, 2011. 21, 22

[16] Deniz Cetinkaya, Alexander Verbraeck, and Mamadou D Seck. Model transformation from bpmn to devs in the mdd4ms framework. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, page 28. Society for Computer Simulation International, 2012. 24, 25

[17] Susan Rose Childers and James E. Long. A concurrent methodology for the system engineering design process. *INCOSE International Symposium*, 4(1):226–231, 1994. 19

[18] North Atlantic Council. Nato architecture framework (naf) ver 3. *North Atlantic Council*, 2007. 18

[19] North Atlantic Council. Nato architecture framework (naf) ver 4 (draft). http://nafdocs.org/, 2015. 18

[20] Judith Dahmann, Kristen J. Baldwin, Åse Jakobson, and Denis JSRD Bertrand. Recommended practice: Systems of systems considerations in the development of systems. In *9th Annual IEEE International System Conference*, 2015. 10

[21] Julien Deantoni, Issa Papa Diallo, Ciprian Teodorov, Joël Champeau, and Benoit Combemale. Towards a meta-language for the concurrency concern in dsls. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 313–316. EDA Consortium, 2015. 23, 31

[22] Papa Issa Diallo. *A Framework for the definition of a System Model MoC-based Semantics in the context of Tool Integration*. PhD thesis, ENSTA Bretagne, 2014. 27, 31

[23] Edsger W Dijkstra. *Selected writings on computing: a personal perspective*. Springer Verlag, 1982. 17

[24] Délégation Générale de l'Armement. Référentiel agate. http://www.ixarm.com/article33349. 18

[25] Ecore. Ecore website. http://wiki.eclipse.org/Ecore. 38

[26] Jeff A Estefan. Survey of model-based systems engineering methodologies (mbse) rev b. *International Council on Systems Engineering. INCOSE, Seattle*, 2008. 19

[27] Serge Fiorèse and Jean-Pierre Meinadier. *Découvrir et comprendre l'Ingénierie Système*. Editions Cépaduès, 2012. 15, 16, 20

[28] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. 89

[29] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Elsevier, 2011. 18, 19, 21, 55, 119

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994. 72

[31] Alfredo Garro, Francesco Parisi, and Wilma Russo. A process based on the model-driven architecture to enable the definition of platform-independent simulation models. In *Simulation and Modeling Methodologies, Technologies and Applications*, pages 113–129. Springer, 2013. 24, 25, 31

[32] Howard Seth Gitlow, Shelly Jan Gitlow, and Monique Sperry. *Le guide Deming pour la qualité et la compétitivité*. Afnor, 1992. 14

[33] Kasper Bilsted Graversen. *The nature of roles*. PhD thesis, IT University of Copenhagen, 2006. 37

[34] Sebastian Günther and Thomas Cleenewerck. Design principles for internal domain-specific languages: a pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM, 2010. 89

[35] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. Conceptual interoperability through models federation. In *Semantic Information Federation Community Workshop*, 2013. 28, 29, 31, 32

[36] Terry Halpin. Orm 2. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, pages 676–687. Springer, 2005. 36

[37] Van Haren. *TOGAF Version 9.1*. Van Haren Publishing, 2011. 18

[38] Cecilia Haskins, Kevin Forsberg, Michael Krueger, David Walden, and R. Douglas Hamelin. Systems engineering handbook. *INCOSE. Version*, 3.2, 2010. 9, 11, 117

[39] Matthew Hause. Model-based system of systems engineering with updm. http://www.omg.org/ocsmp/Model-Based_System_of_Systems_Engineering_with_UPDM.pdf, 2010 (accessed October 3, 2014). 18

[40] Philipp Sebastian Helle, Ian Giblett, and Pascal Levier. An integrated executable architecture framework for system of systems development. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, page 1548512913477259, 2013. 26, 31

[41] Jonas Hellgren. Modelling of hybrid electric vehicles in modelica for virtual prototyping. In *Proceedings of the 2nd International Modelica Conference*, pages 247–256, 2002. 20

[42] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. 138

[43] Hans—Peter HoffMann. Ibm rational harmony deskbook rel 3. 1. 1. 19

[44] Edward Huang, Randeep Ramamurthy, and Leon F McGinnis. System and simulation modeling using sysml. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pages 796–803. IEEE Press, 2007. 25

[45] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report, School of Computer Science and IT, 1996. 49

[46] Takashi Iba, Yoshiaki Matsuzawa, and Nozomu Aoyama. From conceptual models to simulation models: Model driven development of agent-based simulations. In *9th Workshop on Economics and Heterogeneous Interacting Agents*, volume 28, page 149, 2004. 22, 26, 31

[47] ISO ISO. Iso-14258: 1998, industrial automation systems – concepts and rules for enterprise models. *International Organization for Standardization*, 1998. 28, 29

[48] ISO ISO. Iec 15288: 2008, systems and software engineering–system life cycle processes. *2nd International Organization for Standardization/International Electrotechnical Commission, 18th March*, 2008. 9, 12, 13, 15, 20

[49] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008. 26

[50] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. 138

[51] Gilles Kahn and David Macqueen. Coroutines and Networks of Parallel Processes. Rapport de recherche, 1976. 138

[52] Stephen J Kapurch. *NASA Systems Engineering Handbook*. DIANE Publishing, 2010. 12

[53] D Küçükkeçeci Çetinkaya. *Model Driven Development of Simulation Models: Defining and Transforming Conceptual Models into Simulation Models by Using Metamodels and Model Transformations*. TU Delft, Delft University of Technology, 2013. 25, 31

[54] Thomas Kühn, Max Leutäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In Benoït Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer International Publishing, 2014. 33, 34, 37, 50, 70

[55] Jim Long. Relationships between common graphical representations in systems engineering. *Vitech white paper, Vitech Corporation, Vienna, VA*, 2002. 21, 55, 121

[56] Dominique Luzeaux and Jean-Rene Ruault. *Systems of systems*. Wiley Online Library, 2010. 10

[57] Mark W Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998. 10, 118

[58] CHAOS Manifesto. Think big, act small. *The Standish Group International Inc*, 2013. 8

[59] Leon McGinnis and Volkan Ustun. A simple example of sysml-driven simulation. In *Simulation Conference (WSC), Proceedings of the 2009 Winter*, pages 1703–1710. IEEE, 2009. 22, 25, 31

[60] MeDON. Medon data website. http://medon.ensta-bretagne.fr/doku.php. 3, 119

[61] MeDON. Medon project website. http://www.medon.info. 2, 114

[62] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *Workshop on Revival of Dynamic Languages*, 2005. 88

[63] Christian Morel. *Les décisions absurdes: Comment les éviter*, volume 2. Gallimard, 2013. 16

[64] Blazo Nastov, Vincent Chapurlat, Christophe Dony, and François Pfister. A verification approach from mde applied to model based systems engineering: xeffbd dynamic semantics. In *Complex Systems Design & Management*, pages 225–238. Springer, 2015. 23, 31, 127

[65] Object Management Group. Common warehouse metamodel (cwm) specification version 1.1. *Object Management Group*, 2003. 18

[66] Object Management Group. Mda guide version 1.0.1. *Object Management Group*, 2003. 16

[67] Object Management Group. Omg systems modeling language (omg sysml) version 1.3. *Object Management Group*, 2012. 126

[68] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML)*, August 2013. 23, 31

[69] Object Management Group. Mda guide version 2. *Object Management Group*, 2014. 24

[70] Thomas Paviot. *Méthodologie de résolution des problèmes d'interopérabilité dans le domaine du Product Lifecycle Management.* PhD thesis, École Centrale de Paris, 2010. 28

[71] Paul Pearce and Matthew Hause. Iso-15288, oosem and model-based submarine design. In *Systems Engineering Test and Evaluation Conference*, 2012. 19

[72] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. 22

[73] Alain Plantec and Vincent Ribaud. Platypus: A step-based integration framework. In *IDIMT 2006*, pages 261–274, 2006. 27, 31, 32

[74] Ahsan Qamar, Sebastian JI Herzig, Christiaan JJ Paredis, and Martin Torngren. Analyzing semantic relationships between multiformalism models for inconsistency management. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*, pages 84–89. IEEE, 2015. 27

[75] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, 2010. 89, 104

[76] Stewart Robinson. Conceptual modeling for simulation: issues and research requirements. In *Proceedings of the 38th conference on Winter simulation*, pages 792–800. Winter Simulation Conference, 2006. 21

[77] James C Schaaf Jr and Faye Lynn Thompson. System concept development with virtual prototyping. In *Proceedings of the 29th conference on Winter simulation*, pages 941–947. IEEE Computer Society, 1997. 20

[78] Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006. 23

[79] Jean-Philippe Schneider, Joel Champeau, Loic LAGADEC, and Eric Senn. Role framework to support collaborative virtual prototyping of system of systems. In *WETICE 2015*, 2015. 75, 84, 114

[80] Jean-Philippe Schneider, Joel Champeau, Ciprian Teodorov, Eric Senn, and Loic LAGADEC. A role language to interpret multi-formalism system of systems models. In *Syscon 2015*, 2015. 65, 84, 114

[81] Jean-Philippe Schneider, Zoé Drey, and Jean-Christophe Le Lann. Early exploring design alternatives of smart sensor software with model of computation implemented with actors. *IWST 2013*, page 37, 2013. 135

[82] Jean-Philippe Schneider, Eric Senn, Joël Champeau, and Loïc Lagadec. Flexible model-based simulation as a system's design driver. In Boulanger et al. [10], pages 25–35. 135

[83] Jean-Philippe Schneider, Ciprian Teodorov, Eric Senn, and Joël Champeau. Towards a dynamic infrastructure for playing with systems of systems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, page 31. ACM, 2014. 75, 84, 114

[84] Charlotte Seidner. *Vérification des EFFBDs: Model-checking en Ingénierie Système*. PhD thesis, Thèse de doctorat. Université de Nantes, 2009. 23

[85] Mirko Seifert, Christian Wende, and Uwe Aßmann. Anticipating unanticipated tool interoperability using role models. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, pages 52–60. ACM, 2010. 39

[86] Uri Shani and Henry Broodney. Reuse in model-based systems engineering. In *9th Annual IEEE International System Conference*, 2015. 27, 31, 32

[87] Sodius. Mdworkbench. http://sodius.com/products-overview/mdworkbench. 26

[88] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000. 28, 33, 48, 50, 70

[89] Friedrich Steimann. Role= interface: a merger of concepts. *Journal of object oriented programming: JOOP*, 14(4):23–32, 2001. 36

[90] Friedrich Steimann. The role data model revisited. *Applied Ontology*, 2(2):89–103, 2007. 35, 36

[91] Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Principles for the realization of an open simulation framework based on fuml (wip). In *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*, page 4. Society for Computer Simulation International, 2013. 23, 31

[92] The GeMOC Initiative. The gemoc initiative. http://gemoc.org/. 23, 31

[93] Jean-Pierre Treuil, Alexis Drogoul, and Jean-Daniel Zucker. *Modélisation et simulation à base d'agents: exemples commentés, outils informatiques et questions théoriques*. Dunod, 2008. 25, 59

[94] UK Ministry of Defence. Uk ministry of defence architecture framework. https://www.gov.uk/guidance/mod-architecture-framework. 18

[95] United States Department of Defense. Systems engineering guide for systems of systems, August 2008. 10, 11, 15

[96] United States Department of Defense. Dod architecture framework version 2.02 change 1, January 2015. 17, 18

[97] United States of America Department of Defense. Acquisition modeling and simulation plan. Technical report, United States of America Department of Defense, 2006. 21

[98] Hans Vangheluwe, Juan De Lara, and Pieter J Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the AIS'2002 conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal*, pages 9–20, 2002. 21

[99] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008. 22

[100] Jean-Luc Voirin. Modelling languages for functional analysis put to the test of real life. In *Complex Systems Design & Management*, pages 139–150. Springer, 2013. 19

[101] Anthony I Wasserman. Tool integration in software engineering environments. In *Software Engineering Environments*, pages 137–149. Springer, 1990. 28

[102] Christian Wende, Nils Thieme, and Steffen Zschaler. A role-based approach towards modular language engineering. In *Software Language Engineering*, pages 254–273. Springer, 2010. 38

[103] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000. 21, 25, 148, 154

[104] Weiqing Zhang, Vincent Leilde, Birger Moller-Pedersen, Joel Champeau, and Christophe Guychard. Towards tool integration through artifacts and roles. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 603–613. IEEE, 2012. 29

[105] Walter MX Zimmer. *Passive acoustic monitoring of cetaceans*. Cambridge University Press, 2011. 120

Résumé

Les systèmes actuels tendent à être intégrés les uns avec les autres. Mais cette intégration n'est pas forcément prévue à l'origine du système. Cette tendance créée des systèmes de systèmes. Un système de système de systèmes est un système constitué de systèmes qui sont gérés par des équipes indépendantes, qui sont fonctionnellement indépendants, qui collaborent, qui évoluent et qui sont géographiquement distribués. La communication entre les différentes équipes facilite la conception d'un système de systèmes. Cette communication peut être réalisée par l'utilisation de modèles et de simulation.
Cependant, la modélisation du système de systèmes et la modélisation des simulations ne reposent pas sur les mêmes langages. Pour assurer la cohérence des modèles, il faut pouvoir créer les modèles de simulation à partir des modèles système. Cependant, il faut tenir compte des contraintes liées aux propriétés des systèmes de systèmes. Il faut être capable de manipuler des modèles systèmes réalisés dans des langages différents, de réaliser des simulations de natures différentes et suivre les évolutions des langages de modélisation et des outils de simulation.
Pour répondre à ces problématiques, nous avons défini l'environnement Role4All pour la manipulation de modèles systèmes réalisés dans des langages hétérogènes. Role4All est basé sur la notion de rôles. Les rôles permettent de créer des simulations en accédant aux informations contenues dans des éléments de modèles indépendamment de leur type. Role4All est capable de prendre en entrée des modèles sérialisés par différents outils grâce à l'utilisation de parsers combinateurs. Ces derniers apportent modularité et extensibilité aux fonctionnalités d'import. L'environnement Role4All a été utilisé sur un exemple de système de systèmes : l'observatoire sous-marin MeDON.

Abstract

Current systems tend to become integrated with each others. However, this integration may not be designed for the system. This trend raises the concept of system of systems. A system of systems is a system made of systems which are managed independently, functionaly independent, collaborating, evolving and geographically distributed. The communication among the different teams eases the design of the system of systems. This communication may be made through the use of models and simulation.
However, system of systems models and simulation models do not rely on the same modeling languages. In order to ensure coherency between the two types of models, simulation models should be obtained from system models. But this approach should take into account the constraints coming from the properties of system of systems. System models made in different modeling languages should be handled, simulation of different kinds should be generated and the evolution of both modeling languages and simulation tools should be managed.
In order to tackle these issues, we defined the Role4All environment to manipulate system models made in heterogeneous modeling languages. Role4All is based on the concept of roles. Roles enable to create simulations by accessing to information stored in model elements despite their types differences. Role4All is able to take as input serialized models from different modeling tools by using parser combinators. Parser combinators bring modularity and extensibility to the import features. Role4All has been used on a system of system example: the MeDON seafloor observatory.

ENSTA Bretagne

École Nationale Supérieure de Techniques Avancées Bretagne
2 rue François Verny - 29806 Brest Cedex 9
Tél. : 02.98.34.88.00 Fax. : 02.98.34.87.41