HAL open science

Contributions à la conception rigoureuse des systèmes à base de composants exploitant des modèles SysML et des approches formelles

Samir Chouali

▶ To cite this version:

Samir Chouali. Contributions à la conception rigoureuse des systèmes à base de composants exploitant des modèles SysML et des approches formelles. Génie logiciel [cs.SE]. Université Bourgogne Franche-Comté, 2020. tel-03126592

HAL Id: tel-03126592
https://hal.science/tel-03126592

Submitted on 31 Jan 2021

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

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

SPIM

Habilitation à Diriger des Recherches

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Contributions à la conception rigoureuse des systèmes à base de composants exploitant des modèles SysML et des approches formelles

SAMIR CHOUALI

SPIM

Habilitation à Diriger des Recherches

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

HABILITATION À DIRIGER DES RECHERCHES

de l'Université de Franche-Comté

préparée au sein de l'Université Bourgogne Franche-Comté

Spécialité : **Informatique**

présentée par

SAMIR CHOUALI

Contributions à la conception rigoureuse des systèmes à base de composants exploitant des modèles SysML et des approches formelles

Soutenue publiquement le 10 juillet 2020 devant le Jury composé de :

YAMINE AIT-AMEUR	Rapporteur	Professeur à l'INPT-ENSEEIHT de Toulouse
CHRISTIAN ATTIOGBÉ	Rapporteur	Professeur à l'Université de Nantes
GWEN SALAÜN	Rapporteur	Professeur à l'Université Grenoble Alpes
PIERRE-CYRILLE HEAM	Examinateur	Professeur à l'Université de Bourgogne Franche-Comté
OLGA KOUCHNARENKO	Examinatrice	Professeur à l'Université de Bourgogne Franche-Comté
HASSAN MOUNTASSIR	Examinateur	Professeur à l'Université de Bourgogne Franche-Comté

REMERCIEMENTS

Tout d'abord, je remercie chaleureusement les trois rapporteurs de cette HDR, les professeurs Yamine AIT-AMEUR, Christian ATTIOGBÉ et Gwen SALAÜN, pour le temps consacré à la relecture et à l'évaluation de ce manuscrit.

Je tiens également à remercier tous mes collègues du DISC qui m'ont soutenu et encouragé dans mes activités scientifiques. Je remercie particulièrement Hassan Moutassir qui m'a donné l'opportunité pour co-encadrer avec lui des travaux de thèse. Mes remerciements vont également à Pierre-Cyrille Héam qui m'a accompagné et soutenu pour la préparation de cette habilitation. Je tiens à remercier Olga Kouchnarenko qui m'a encouragé, ces dernières années, pour soutenir l'HDR. Aussi, je remercie les collègues avec qui j'ai eu des collaborations enrichissantes sur des thématiques variées : Ahmed hammad (développement de systèmes complexes en exploitant des modèles SysML) et Ahmed Mostefaoui (validation des protocoles d'interaction dans les véhicules connectés). Merci à Julien Bourgeois, directeur du DISC, qui m'a soutenu et encouragé, ces dernières années, pour effectuer des séjours scientifiques à l'étranger et initier des collaborations.

Je voudrais aussi remercier les doctorants que j'ai co-encadrés, qui ont eu une place centrale dans ce travail.

Je remercie enfin ma famille pour son soutien permanent et pour avoir accepté, ces derniers mois, le partage du temps familial avec mes activités professionnelles.

TABLE DES MATIÈRES

Table des matières vii

I Introduction 1

1 Introduction 3

1.1 Contexte scientifique : développement des systèmes à base de composants critiques . . . 3

1.2 Problématiques scientifiques . . . 4

1.3 Contributions . . . 8

1.4 Mes publications . . . 11

II Conception incrémentale des systèmes à base de composants guidée par SysML 13

2 Méthodologie pour l'assemblage de composants guidée par SysML 15

2.1 Introduction . . . 15

2.2 État de l'art et contributions . . . 16

2.3 Formalisme des automates d'interface . . . 17

2.3.1 L'approche optimiste pour la vérification de la compatibilité . . . 17

2.3.2 Composabilité, compatibilité et composition . . . 18

2.4 Spécification de l'architecture d'un SBC avec SysML . . . 19

2.5 Dérivation automatique des automates d'interface à partir de SysML . . . 21

2.6 Vérification de l'assemblage dans un SBC modélisé avec SysML . . . 23

2.6.1 Modèles formels des diagrammes SysML spécifiant l'architecture d'un SBC . . . 24

2.6.2 Algorithme de vérification . . . 25

2.7 Conception incrémentale d'un SBC par des raffinements successifs . . . 27

2.8 Conclusion . . . 30

2.9 Bilan . . . 30

3 Adaptation des composants SysML lors du développement d'un SBC 31

3.1 Introduction 31

3.2 Aperçu de l'approche 31

3.3 État de l'art et contributions 33

3.4 L'approche d'adaptation des blocs SysML 34

3.4.1 Spécification SysML de la partie à développer 34

3.4.2 Sélection des blocs réutilisables $\{B_i\}$ 35

3.4.3 Vérification de la validité du contrat d'adaptation 37

3.4.4 Génération de l'adaptateur 38

3.5 Conclusion et perspectives 42

3.6 Bilan 44

4 Spécification incrémentale d'un SBC en exploitant les exigences 45

4.1 Introduction 45

4.2 Aperçu de l'approche 46

4.3 État de l'art et contributions 47

4.4 Étude de cas 49

4.5 Analyse du diagramme des exigences SysML 50

4.6 Vérification formelle des exigences SysML sur des composants 51

4.6.1 Vérification avec SPIN 52

4.6.2 Spécification des exigences avec la LTL 52

4.7 Assemblage des composants et préservation des exigences SysML 54

4.7.1 Les exigences fonctionnelles et les actions d'entrée/sortie 54

4.7.2 Préservation des actions d'entrée/sortie par la composition des automates d'interface 55

4.7.3 Vérification de la préservation des exigences atomiques 55

4.8 Spécification de l'architecture du système 56

4.9 Illustration sur l'étude de cas 57

4.10 Conclusion et perspectives 59

4.11 Bilan 59

III Assemblage des composants basé sur leurs contrats comportementaux 61

5 Conception à base de composants orientés objet 63

5.1 Introduction 63

5.2 État de l'art et contributions 64
5.3 Contrats comportementaux 65
5.3.1 Formalisation des protocoles des composants orientés objet 65
5.3.2 Sémantique des méthodes 67
5.4 Étude de cas d'un système ferroviaire 68
5.4.1 Conception de l'étude de cas ferroviaire 69
5.5 Compatibilité des composants 71
5.5.1 Synchronisation des automates d'interface et compatibilité sémantique 72
5.5.2 L'approche optimiste pour l'assemblage de composants 73
5.6 Raffinement 74
5.6.1 Simulation d'expansion 75
5.6.2 Substituabilité sémantique 76
5.6.3 Propriétés du raffinement 77
5.7 Adaptation des composants 78
5.7.1 Aperçu de l'approche 79
5.7.2 État de l'art et contributions 80
5.7.3 Contrat d'adaptation 81
5.7.4 Adaptabilité sémantique 81
5.7.5 Spécification de l'adaptateur 83
5.8 Génération automatique de l'adaptateur 84
5.9 Conclusion 86
5.10 Bilan 86

6 Vérification de l'interopérabilité des composants avec B 87
6.1 Introduction 87
6.2 État de l'art et contributions 88
6.3 Spécification des interfaces de composants avec B 90
6.3.1 Définition 90
6.3.2 Étude de cas 90
6.4 Vérification de l'interopérabilité avec le raffinement B 94
6.4.1 Définitions 95
6.4.2 Étude de cas 99
6.5 Conclusion 99
6.6 Bilan 100

7 Vérification des protocoles de coordination Reo avec SPIN 101

7.1 Introduction 101

7.2 État de l'art et contributions 102

7.3 Reo 103

7.3.1 Connecteurs synchrones et asynchrones 103

7.3.2 Expression logique du comportement d'un connecteur Reo 104

7.4 Formalisation des contraintes Reo sous forme de commandes gardées . . 106

7.5 Formalisation des connecteurs Reo avec Promela 108

7.5.1 Formalisation des Ports 108

7.5.2 Formalisation du protocole défini par un connecteur 109

7.5.3 Formalisation des propriétés LTL 110

7.6 Conclusion et perspectives 111

7.7 Bilan 111

IV Conclusion et Perspectives 113

8 Conclusion et perspectives 115

8.1 Conclusion 115

8.2 Perspectives de recherche 115

8.2.1 Conception des CPS, correct-par-construction, en exploitant les langages Sysml/Marte/pccsl et les contrats comportementaux des composants 116

8.2.2 Assemblage et adaptation des composants interagissant d'une manière asynchrone 116

8.2.3 Analyse et vérification des protocoles de communication dans les véhicules communicants 117

Bibliographie 119

Table des figures 135

Liste des tables 137

I

INTRODUCTION

1

INTRODUCTION

1.1/ CONTEXTE SCIENTIFIQUE : DÉVELOPPEMENT DES SYSTÈMES À BASE DE COMPOSANTS CRITIQUES

Ce document présente mes travaux de recherche menés après l'obtention de mon doctorat. Ceux-ci concernent principalement l'utilisation des notations semi-formelles et des approches formelles pour la conception rigoureuse des systèmes à base de composants (SBC) critiques.

J'ai commencé à travailler sur la thématique du développement des SBC lors de mon année de post-doctorat au laboratoire LORIA (équipe DEDALE), en 2004. Au cours de cette année, je me suis intéressé à l'utilisation de la méthode B [Abr96b] pour la vérification de l'assemblage des composants. Après mon recrutement à l'université de Franche-Comté en tant que maître de conférences (en 2005), j'ai poursuivi mes travaux sur les SBC lors de ma participation au projet ANR TACOS (Trustworthy Assembling of Components : frOm requirements to Specifications) (2007-2010). L'objectif de ce dernier était de proposer une approche de conception par composants pour le développement de systèmes fiables, de l'expression des besoins jusqu'à une spécification formelle. Le domaine d'application choisi était celui d'un véhicule en libre service, appelé CyCab, destiné a être utilisé dans les villes du futur pour le transport des personnes d'une manière autonome. La nature complexe du CyCab (composé de composants électroniques, mécaniques, informatiques), nous a motivé pour le considérer comme étude cas pour illustrer notre de démarche de conception à base de composants.

En effet, la conception à base de composants [HC01, Szy02] a montré son efficacité pour répondre à la complexité croissante des systèmes informatiques et pour s'adapter à leur évolution rapide. Ainsi, en s'inspirant du concept des composants utilisés dans plusieurs domaines (électronique, mécanique,...), un logiciel peut aussi être construit par l'assemblage de briques logiciels réutilisables appelées composants logiciels. D'après la définition de Clemens Szyperski [Szy02], un composant logiciel est une unité de composition avec des interfaces définies contractuellement et des dépendances au contexte explicites. Un composant est donc spécifié par ses interfaces, qui exhibent généralement ses services fournis et requis. Elles servent, d'une part à exprimer ce que l'on attend de l'environnement pour le bon fonctionnement d'un composant, et d'autre part à décrire ce qui est offert par un composant. Elles définissent donc une sorte de *contrat* entre le composant et son environnement.

La démarche à base de composants vise donc à concevoir et à développer des systèmes

par l'assemblage de composants préfabriqués hétérogènes (développés éventuellement par des tierces parties), en se basant uniquement sur leurs interfaces, au lieu de faire un développement à partir de zéro. L'objectif est d'établir une sorte de marché commun de composants logiciels, appelés COTS (Commercial-Off-The-Shelf), permettant ainsi aux développeurs de trouver les composants à intégrer dans leurs systèmes. Cette démarche offre principalement deux avantages :

— Réduire les coûts de développement d'une application par la réutilisation et l'assemblage de composants préfabriqués. En effet, il suffit de réutiliser des composants développés par des tiers pour construire de nouveaux systèmes complexes.

— Garantir une maintenabilité à long terme des logiciels, car une propriété intrinsèque aux SBC, c'est leur capacité d'évolution et d'adaptation aux nouveaux environnements. En effet, il suffit de connecter un nouveau composant à un système existant pour lui rajouter une nouvelle fonctionnalité ou un nouveau service. Et il suffit de changer un composant défaillant pour assurer la maintenance du système existant.

Par ailleurs, nous assistons ces dernières années à une demande de logiciels fiables, de plus en plus complexes et capables de supporter de nouvelles fonctionnalités, pour une utilisation dans divers domaines : transport, militaire, médical, domotique, business.... La solution pour concevoir ces systèmes complexes reste la réutilisation des composants et l'exploitation de l'approche de conception par composants. En effet, c'est une approche qui est applicable à divers domaines : il existe de nombreux modèles de composants [CSVC11, LW07], ciblant des domaines d'applications variés. De plus, c'est une approche qui a montré son efficacité au niveau industriel. Nous citons par exemple, le modèle à composants Robus [HMN+08, MMTL+16], développé dans le cadre d'une collaboration entre industriels (Arcticus Systems, Volvo Construction Equipment and BAE Systems Hägglunds) et l'université de Mälardalen (Suède), est utilisé avec succès dans l'industrie automobile pour le développement des logiciels embarqués. Néanmoins, le modèle de composants idéal n'existe pas [LW07], et les techniques de conception à base de composants continuent de soulever de nouvelles problématiques.

Dans la section suivante, nous présentons les principales problématiques que nous avons abordées dans nos travaux. Elles sont liées à l'exploitation de SysML [sys] et les contrats des composants, basés sur le formalisme des automates d'interface [dAH01, AH05], pour le conception rigoureuse des SBC. Les réponses à ces problématiques, sont présentées dans la section contributions.

1.2/ Problématiques scientifiques

Modélisation semi-formelle des SBC avec SysML

Certes, comme évoqué ci-dessus, l'approche à composants présente plusieurs avantages, néanmoins concevoir un SBC répondant aux exigences de l'utilisateur reste une tâche difficile. Cette difficulté est inhérente à la nature complexe de ces systèmes, constitués d'un ensemble de composants réutilisables, donc par définition, destinés à être utilisés dans différentes applications, dont certaines peuvent encore être inconnues et dont les exigences ne peuvent pas être prévues. Ainsi, pour comprendre ces systèmes en vue de leur conception, il est nécessaire d'utiliser un langage de modélisation permettant d'avoir une vue générale du système, tout en décrivant les niveaux liés à sa struc-

ture, son comportement, et ses exigences. Pour cela, le langage SysML (Systems Modeling Language) [sys], qui est un profil UML [uml] pour l'ingénierie des systèmes, nous semble le plus approprié pour modéliser les systèmes complexes. En effet, en plus de la modélisation structurelle et comportementale d'un système, SysML offre la possibilité de décrire ses exigences, ainsi que ses composants physiques et logiciels.

Et comme les SBC, sont de plus en plus exploités dans des domaines critiques, leur sûreté de fonctionnement exige l'emploi de méthodes formelles pour vérifier l'assemblage de leurs composants lors de la conception. Dans ce contexte, l'utilisation de SysML comme langage de modélisation, pose les problématiques suivantes :

(M1) Quels modèles SysML utiliser pour spécifier, sans ambiguïté, l'architecture, le comportement, et les exigences d'un SBC, et comment prendre en considération ces modèles lors du passage au niveau formel pour vérifier l'assemblage des composants ?

(M2) Comment garantir la satisfaction des exigences SysML lors de la conception d'un SBC ?

INTEROPÉRABILITÉ DES COMPOSANTS ET LEURS INTERFACES

Un SBC, comme tout système informatique, n'échappe pas également aux problèmes de dysfonctionnements. Néanmoins, dans le domaine des composants, ces dysfonctionnements sont principalement la cause des problèmes d'interopérabilité des composants (appelée aussi compatibilité des composants). L'interopérabilité est définie par la capacité de deux (ou plusieurs) composants à communiquer et à coopérer malgré les différences dans leurs langages d'implémentation, leurs environnements d'exécution, ou leurs modèles d'abstraction [Weg96, Kon95]. La vérification de l'interopérabilité des composants est donc cruciale pour le développement des SBC, car cela permet aux concepteurs de déterminer si des composants peuvent interagir malgré leur hétérogénéité. Cette vérification repose sur la compatibilité des interfaces des composants. Celles-ci spécifient généralement des informations aux niveaux signature (profil des opérations), sémantique (des opérations), protocole (ordre d'appel des opérations d'un composants), et qualité de services (temps de réponse, coûts,...). Et généralement, plus les interfaces sont expressives, plus fiable est le résultat de la vérification de l'interopérabilité.

En plus de l'interopérabilité, la substituabilité des composants est également importante lors du développement des SBC. Sa vérification, en se basant sur une relation de raffinement entre les interfaces des composants, permet de décider si un composant peut être remplacé par un autre plus évolué, sans corrompre le fonctionnement du système global. Cela permet aussi de construire des SBC, progressivement, par des raffinements successifs.

Dans la littérature, plusieurs formalismes ont été proposés pour spécifier les interfaces des composants. Par exemple dans [CFTV00], un formalisme basé sur le π-calcul est utilisé pour enrichir les langages de description d'interfaces (IDL) classiques (limités au niveau signature), avec la prise en compte des protocoles des composants. Par ailleurs, plusieurs travaux proposent de spécifier les interfaces des composants avec des automates, comme ceux proposés dans [dAH01, AH05] basés sur le formalisme des automates d'interface, ainsi que ceux proposant les automates d'interface modales [RBB+09, LNW07]. Ces travaux se focalisent principalement sur la spécification des pro-

tocoles des composants et la vérification de la compatibilité et du raffinement des interfaces.

Le concept de contrat est également utilisé pour compléter les spécifications de composants, pour y introduire principalement la sémantique des opérations. Il a été introduit par B. Meyer dans [Mey92], pour définir la sémantique des méthodes sous forme de pré et post-conditions, et d'un invariant. Par exemple, la sémantique des méthodes a été prise en compte dans l'approche proposée dans [CD01], basée sur UML et OCL, pour la modélisation des composants. Dans le modèle à composants Kmelia [AAA06, AAA08], la sémantique des services des composants est également prise en compte dans la spécification de leurs interfaces. Dans [BJPW99], une classification des contrats a été proposée : contrat syntaxique (donne la signature des opérations) ; contrat comportemental (spécifie les pré- et post-conditions des opérations, ainsi qu'un invariant) ; contrat de synchronisation (l'ordre d'invocation des opérations) ; contrat de qualité de service (permet de quantifier le service fourni).

Au début de nos travaux, nous nous sommes intéressés à l'utilisation de la méthode B [Abr96a] pour vérifier l'interopérabilité des composants. L'objectif était d'exploiter les outils de cette méthode pour vérifier formellement l'interopérabilité des composants. D'où la problématique suivante : **(I1) comment utiliser B pour spécifier les interfaces des composants et ensuite comment vérifier leur compatibilité avec la méthode B ?**

Ensuite, nous nous sommes focalisés sur l'utilisation du formalisme des automates d'interface [dAH01] pour spécifier les interfaces des composants et vérifier leur assemblage. Notre choix a été motivé par l'approche optimiste des automates d'interface pour vérifier l'assemblage des composants, particulièrement adaptée aux systèmes ouverts (approche présentée dans le chapitre 2). Un automate d'interface permet d'exhiber les informations des composants aux niveaux signature et protocole. Donc la problématique qui s'est posée par rapport à notre choix de SysML comme langage de modélisation est : **(I2) comment prendre en considération les modèles SysML spécifiant un SBC, lors de la vérification de l'interopérabilité de ses composants en se basant sur leurs automates d'interface ?**

Par ailleurs, dans nos travaux, nous avons principalement traité des études de cas de systèmes à composants dans le domaine des transports (véhicules autonomes, transport ferroviaire,...). Et pour les interfaces des composants de ces systèmes, nous avons besoin d'exhiber des informations aux niveaux signature, sémantique, et protocole. Or, le formalisme des automates d'interface ne prend pas en compte le niveau sémantique, donc son expressivité est insuffisante pour vérifier correctement l'interopérabilité des composants. D'où la problématique suivante : **(I3) comment prendre en considération le niveau sémantique dans le formalisme des automates d'interface pour vérifier l'interopérabilité et la substituabilité des composants ?**

ADAPTATION DES COMPOSANTS

L'assemblage de composants réutilisables est souvent contraint par des problèmes d'inadéquations (*mismatches*) aux niveaux des noms (ou signatures) de leurs services échangés (par exemple, lors de l'association de noms différents au même service fourni et requis dans deux composants devant interagir), ou de leurs protocoles d'interaction (par exemple, lorsque l'ordre des messages échangés entre les composants n'est pas adéquat). Ce qui provoque des blocages lors de leur assemblage. Ces incompatibi-

lités résultent généralement de la réutilisation des composants, qu'il faut donc adapter à chaque fois qu'il y a un besoin de les intégrer dans un nouveau système. Pour y remédier, une solution existe, c'est l'introduction d'un composant intermédiaire, appelé *adaptateur* [YS97, CMP06, CPS06a], qui joue le rôle de médiateur entre les composants réutilisés. Ce faisant, toutes les interactions transitent par cet adaptateur, qui prend le rôle d'orchestrateur en résolvant les inadéquations, et permet ainsi aux composants concernés d'interagir correctement. Dans notre contexte, les problématiques qui se sont posées sont les suivantes :

(A1) Comment générer automatiquement un adaptateur en partant des modèles SysML d'un SBC présentant des inadéquations entre ses composants ?

(A2) Et concernant le formalisme des automates d'interface, une autre problématique s'est posée : **comment générer automatiquement un adaptateur pour des composants spécifiés par des contrats comportementaux, basés sur le formalisme des automates d'interface, en considérant les niveaux signature, sémantique, et protocole des composants ?** En effet, dans l'approche des automates d'interface, l'adaptation des composants n'est pas traitée, ce qui justifie cette problématique.

COORDINATION

L'une des solutions contribuant à résoudre les problèmes d'assemblage des composants dans les systèmes concurrents et distribués, est l'exploitation des modèles et des langages de coordinations [PA98, BCGZ01]. Ces derniers offrent principalement des primitives pour spécifier les interactions synchronisées entre les composants. Ainsi, étant donné un ensemble d'entités en interaction, le modèle de coordination a pour but de les faire interagir correctement. Donc, cela permet de définir une entité, implémentant un protocole de coordination, pour coordonner les interactions des composants d'une manière exogène. Autrement dit, en faisant abstraction de leurs comportements internes. Ce protocole définit une sorte de contrat d'interaction entre l'ensemble des composants du système.

Dans ce contexte, nous nous sommes intéressés à l'utilisation du langage de coordination Reo [Arb04], développé par le Professeur F. Arbab et son équipe du centre de recherche CWI d'Amsterdam. Reo, offre un ensemble de primitives de synchronisation, basées sur les canaux de communication, pour construire d'une manière compositionnelle des protocoles de coordination. Et comme ces protocoles définissent l'ensemble des interactions des composants dans un SBC, leur sûreté de fonctionnement doit donc être garantie afin de garantir celle du système global. Ainsi, pour vérifier des propriétés sur les protocoles Reo, nous avons choisi d'exploiter le Model-Checker SPIN [Hol03], pour ses qualités pour faire face au problème d'explosion combinatoire de l'espace d'états lors de la vérification et aussi pour sa popularité. La problématique suivante s'est alors posée : **(C1) comment traduire les protocoles Reo et leurs propriétés à vérifier, vers, respectivement, le langage Promela, de SPIN, et la logique LTL (*Linear Temporal Logic*), tout en préservant les propriétés de synchronisation des primitives Reo ?**

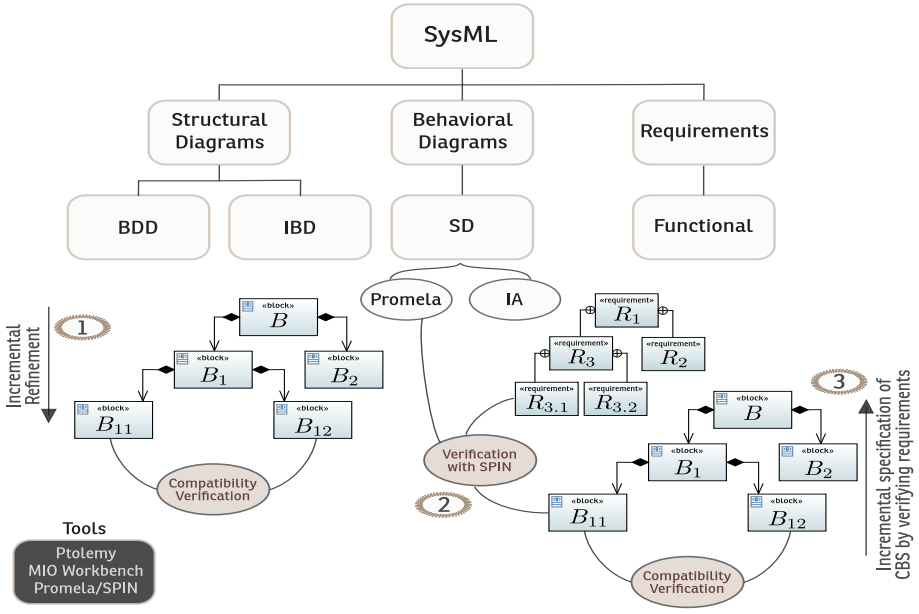


FIGURE 1.1 – Conception incrémentale d'un SBC guidée par SysML

1.3/ CONTRIBUTIONS

Conception incrémentale des SBC fiables guidée par des modèles SysML : Nous avons obtenu plusieurs résultats concernant la conception rigoureuse des SBC guidée par des modèles SysML, spécifiant la structure, le comportement, et les exigences du système à développer.

— **Modélisation des SBC avec SysML :** Pour réponde à la problématique **(M1)**, nous avons proposé une démarche pour décrire l'architecture (avec un diagramme de définition de blocs SysML BDD, avec la définition des ports proxy et des blocs d'interface pour modéliser les interfaces des composants, et un diagramme de blocs internes IBD), le comportement (avec un diagramme de séquence SD), et les exigences (avec un diagramme des exigences), des SBC (voir les diagrammes utilisés dans la figure 1.1 [1]). Ces diagrammes sont ensuite transformés automatiquement au niveau formel pour les prendre en considération dans la vérification de l'assemblage des composants en exploitant le formalisme des automates d'interface (travaux présentés dans le chapitre 2). Ces travaux concernent certaines parties des thèses encadrées [Roz15, Bou16] et les publications [CH11a, BCHM19].

— **Assemblage des composants guidé par SysML :** Pour répondre à la problématique **(I2)**, nous avons proposé une approche incrémentale pour construire un SBC en partant d'une spécification abstraite du système global, décrit par des diagrammes SysML. Le système est construit progressivement en sélectionnant des composants adéquats tels que leur composition respecte les contraintes définies dans la spécification abstraite. La vérification de ces contraintes est effectuée en spécifiant et vérifiant une relation de raffinement entre les interfaces des composants. Les outils Ptolemy [LX04] et MIO Workbench [BMSH10] sont utilisés

1. Figure extraite de la thèse de Oscar Carillo

pour vérifier respectivement la compatibilité et le raffinement des interfaces (voir dans la figure 1.1 (1)). Cette contribution exploite les résultats de la contribution précédente, elle est également en lien avec la problématique **(M1)**. Ces travaux sont résumés dans le chapitre 2. Ils concernent une partie de la thèse encadrée [Roz15] et la publication [CCM12a].

— **Construction incrémentale de l'architecture d'un SBC guidée par les exigences SysML :** Pour répondre à la problématique **(M2)**, nous avons proposé une approche pour construire l'architecture d'un SBC d'une manière incrémentale en analysant son diagramme des exigences SysML (fonctionnelles). Ces dernières sont vérifiées une par une, sur le comportement des composants élémentaires, en utilisant le Model-Checker SPIN (voir la figure 1.1 (2)). Les composants vérifiés sont ensuite intégrés dans l'architecture partielle du système après avoir vérifié leur compatibilité grâce à leurs automates d'interface (voir la figure 1.1 (3)). Ces travaux rentrent dans le cadre de la thèse [Roz15] et sont synthétisés dans le chapitre 4. Ils sont publiés dans [CCM13a, CCM13b]. Dans [CCM13b], nous avons également proposé une approche pour vérifier l'assemblage des composants en considérant des exigences non-fonctionnelles spécifiées avec SysML.

— **Génération automatique des adaptateurs lors de la conception des SBC modélisés avec SysML :** Pour répondre à la problématique **(A1)**, nous avons proposé une approche pour générer automatiquement un adaptateur permettant aux composants, modélisés avec SysML, d'interagir malgré l'incohérence existante entre les noms de leurs services échangés. La génération de l'adaptateur est guidée par les spécifications SysML du composite à développer (spécification initiale du système, sous forme d'un composite), initialement défini par le concepteur. Notre approche se distingue par rapport aux travaux existants par son exploitation des modèles semi-formels (SysML) et formels (automates d'interface), pour traiter les incohérences entre les composants et corriger leur incompatibilité grâce à l'adaptation. Ces travaux concernent une partie de la thèse [Bou16] et sont présentés dans le chapitre 3. Ils sont publiés principalement dans [BCZ15, BCHM16b].

Conception rigoureuse de systèmes critiques à base de composants orientés objet : Pour répondre à la problématique **(I3)**, nous avons proposé une démarche pour développer des SBC corrects par la conception en exploitant leurs interfaces spécifiées par des contrats comportementaux. Ce formalisme est basé sur les automates d'interface enrichis avec la sémantique des méthodes (voir la figure 1.2 [2]). Nous avons donc proposé un cadre formel pour définir et vérifier l'interopérabilité des composants orientés objet (utilisés dans le domaine des transports). Nous avons également défini la relation de raffinement des composants, pour vérifier leur implémentabilité indépendante (ou leur substituabilité). Autrement dit, vérifier si les interfaces des composants compatibles peuvent être raffinées séparément tout en maintenant leur compatibilité. Les études de cas traitées dans ces travaux concernent le domaine des transports, en particulier celui du ferroviaire. Et enfin, pour répondre à la problématique **(A2)**, nous avons proposé une approche d'adaptation des composants, décrits par leurs contrats comportementaux et présentant des inadéquations entre leurs services échangés. Ces travaux concernent, en partie, la thèse [Mou11] et sont présentés dans le chapitre 5. Ils sont principalement publiés dans [CMM10e, MCM09, MACM15b, CMM12a, CMM10b].

2. Figure extraite de la thèse de Sebti Mouelhi

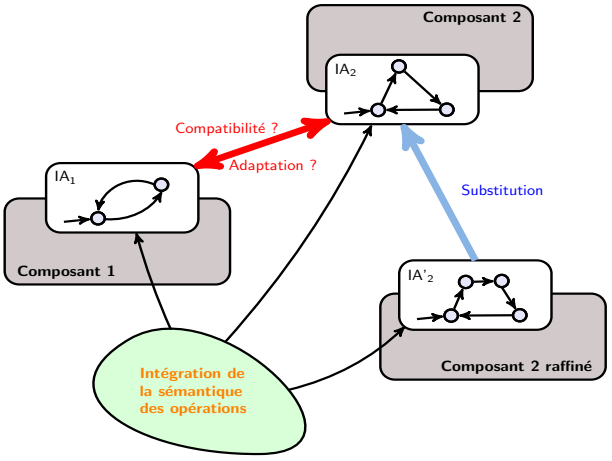


FIGURE 1.2 – Assemblage des composants basé sur les Automates d'interfaces

Exploitation de la méthode B pour spécifier les interfaces des composants et vérifier leur assemblage : Pour répondre à la problématique **(I1)**, nous avons proposé, dans [CHS06], une approche pour spécifier formellement les interfaces des composants en utilisant le formalisme de la méthode B. En se basant sur les spécifications des interfaces, nous avons montré qu'il est possible d'utiliser la méthode B, et particulièrement, son raffinement pour prouver formellement la compatibilité des composants aux niveaux signature et sémantique de leurs opérations. Dans [Cho06], nous avons étendu ce travail pour considérer, en plus, les protocoles des composants dans la vérification de l'assemblage. Le chapitre 6 présente ces travaux.

Transformation automatique des protocoles de coordination Reo vers Promela pour vérifier leurs propriétés : Depuis fin 2017, je travaille sur la spécification et la vérification des protocoles de coordination dans les systèmes à base de composants/services, en collaboration avec le groupe méthodes formelles (avec le Professeur F.Arbab) du centre de recherche en informatique d'Amsterdam CWI. Ainsi, pour répondre à la problématique **(C1)**, nous avons proposé une approche permettant de traduire automatiquement les protocoles de coordination spécifiés avec le langage Reo (développé au CWI), en Promela, afin de vérifier des propriétés temporelles avec le Model-Checker SPIN. Ce travail rentre dans le cadre de la thèse de Benjamin Lion du CWI. Il est présenté dans le chapitre 7.

Vérification de protocoles de communication dans le domaine des véhicules autonomes/communicants : Depuis 2017, je collabore avec des collègues de l'équipe AND (Algorithmique Numérique Distribuée) du DISC (FEMTO-ST) et le Professeur A.Boukerche de l'université d'Ottawa (Canada), sur l'utilisation des approches formelles pour la validation des protocoles de communication dans les véhicules autonomes et/ou communicants (et aussi dans le domaine des réseaux de capteurs). Ce qui représente pour moi une ouverture vers de nouvelles thématiques de recherche. Dans ce cadre, nous avons obtenu des premiers résultats prometteurs (qui ne sont pas détaillés dans ce manuscrit mais accessibles par les articles référencés ci-dessous) :

— Spécification et vérification du protocole MQTT-CV : nous avons adapté le protocole

MQTT (Message Queuing Telemetry Transport), utilisé dans les applications IOT (internet des objets) pour les véhicules communicants, et proposé une approche pour le vérifier formellement [CBM17b].

— Vérification de l'interaction des composants en considérant leur consommation de l'énergie dans les véhicules autonomes : nous avons adapté le formalisme des automates d'interface pour modéliser et vérifier le protocole d'interaction entre les différents composants dans un véhicule autonome en considérant les contraintes de la consommation de l'énergie [CBM17a] (papier, best short paper de la conférence MSWIM, classée A).

— Proposition d'une nouvelle approche d'agrégation de données en série, dans les réseaux de capteurs, permettant d'optimiser les chemins parcourus et aussi de réduire les communications entre les capteurs [MBMC19].

1.4/ MES PUBLICATIONS

La liste ci-dessous synthétise les publications auxquelles j'ai participé depuis 2005, année de mon recrutement en tant qu'enseignant chercheur.

— **Journaux internationaux (8)** : [BCHM19, MBMC19, CHM13, CCM12a, CH11b, CMM10e, CHS06, CJMB05]

— **Journaux nationaux (3)** : [BCHM16a, CMM12a, CDH+10]

— **Conférences internationales (22)** : [FMCB19, LCA18, CMM18, DBMC17, CBM17b, CBM17a, MMC17, BCHM16b, BCZ15, MACM15b, BCHM15a, CCM13a, CCM13b, HMC13b, HMC13a, CMM10b, MCM11, MCM09, CMM10c, SC05, BCJ05, Cho06]

— **Conférences nationales (5)** : [BCHM15b, CCM14b, CCM12b, CMM10a, CMM10d]

II

Conception incrémentale des systèmes à base de composants guidée par SysML

2

Méthodologie pour l'assemblage de composants guidée par SysML

2.1/ Introduction

Ce chapitre présente notre méthodologie permettant à l'architecte d'un SBC de le modéliser avec SysML et de vérifier formellement l'assemblage de ses composants. Ainsi, nous proposons de combiner des modèles SysML et le formalisme des automates d'interface afin de vérifier formellement l'assemblage des composants spécifiés avec des diagrammes SysML. Pour cela, nous proposons, tout d'abord, de modéliser l'architecture d'un SBC avec un ensemble de modèles SysML, qui sont ensuite traduits automatiquement vers des modèles formels, qui sont à leur tour exploités pour vérifier l'assemblage des composants constituant le système global. Ce qui nous permet de valider ou de corriger l'architecture proposée. Par ailleurs, nous proposons également une approche pour construire un SBC en se basant sur sa spécification abstraite. C'est à dire, nous commençons la conception du système, à développer, par sa spécification SysML abstraite, décrivant sa structure et son comportement, et nous proposons ensuite un ensemble de composants réutilisables, telles que leur composition vérifie les contraintes imposées par la spécification de départ. Nous exploitons une relation de raffinement entre les interfaces des composants pour vérifier la cohérence entre le niveau abstrait et le niveau concret du système.

Dans les approches proposées, la conception du système se fait d'une manière incrémentale : nous proposons d'assembler les composants d'un système progressivement, et grâce à l'approche des automates d'interface, nous avons la possibilité de vérifier l'assemblage des composants pour une vue partielle du système sans connaître les interfaces de tous les composants.

Ce chapitre est structuré de la manière suivante. La section 2.2 porte sur les travaux connexes et nos contributions. Dans la section 2.3, nous présentons le formalisme des automates d'interface. Dans la section 2.4, nous montrons comment modéliser l'architecture d'un SBC avec SysML. Dans la section 2.5, nous présentons nos travaux pour le passage des modèles SysML spécifiant les protocoles des composants vers les automates interfaces. La section 2.6 est dédiée à la vérification de l'assemblage de l'ensemble des composants constituant le SBC décrit avec SysML, en exploitant le formalisme des automates d'interface. Notre approche pour construire un SBC par raffinement est décrite dans la section 2.7. Nous terminons ce chapitre par une conclusion et un bilan présentés respectivement dans les sections 2.8 et 2.9.

2.2/ ÉTAT DE L'ART ET CONTRIBUTIONS

Plusieurs travaux existent dans la dérivation des modèles formels à partir des modèles semi-formels comme UML et SysML. Par exemple, les auteurs dans [KBSB10, RF06, Wan08, ES09, Mer14] se sont focalisés sur la transformation des diagrammes de séquence vers des réseaux de Petri ou des réseaux de Petri colorés, en adoptant diverses méthodes. Une approche permettant de vérifier des diagrammes de séquence avec le Model-Checker SPIN, a également été proposée dans [LTM+09]. Contrairement à ces travaux, dans notre cas, nous proposons de générer des automates d'interface à partir des diagrammes de séquence en vue de vérifier l'assemblage des composants.

Il existe également quelques travaux exploitant d'autres diagrammes SysML dans le but de vérifier formellement des propriétés. Par exemple, dans [JDB09], les auteurs proposent une syntaxe et une sémantique formelle pour les diagrammes d'activité SysML. Leur contribution principale est la définition du langage appelé *Activity Calculus*, avec une sémantique opérationnelle formelle. Ce qui permet de détecter des erreurs de conception dés le début du processus de développement. Les diagrammes d'activité ont également été considérés dans [OMD13, OMD14], où les auteurs proposent une approche permettant de vérifier formellement ces diagrammes avec le Model-Checker PRISM [KNP11]. Cet outil a été également exploité dans [Ali18] pour vérifier des systèmes embarqués, modélisés principalement avec le diagramme de blocs SysML . Dans [CMC+08], les auteurs proposent une solution pour la modélisation et la vérification de systèmes embarqués en temps réel avec des contraintes d'énergie. Pour cela, ils combinent les fonctionnalités des modèles SysML et des annotations du profil MARTE avec les avantages d'utiliser l'outil Time Petri Net. Ce formalisme permet l'analyse et la vérification des exigences fonctionnels, temporels et énergétiques dans les premières phases du développement. Dans [KAdSS11], les auteurs présentent TEPE (TEmporal Property Expression), un langage graphique, basé sur des diagrammes paramétriques SysML, pour l'expression des propriétés. Ces dernières sont construites sur des relations logiques et temporelles entre les attributs et les signaux des blocs. TEPE peut être intégré à un profil temps réel de SysML comme AVATAR. Ce dernier est un profil orienté vérification, permettant d'exprimer et de vérifier des propriétés temporelles. Il est supporté par la boîte à outils open source TTool. Elle comprend un éditeur de diagramme, un générateur de code UPPAAL et une interface de bouton-poussoir pour la vérification formelle. Contrairement à notre approche, ces travaux ne se situent pas dans le contexte du développement des SBC, donc les questions liées à l'interopérabilité des composants ne sont pas traitées.

Contributions : L'originalité de notre approche par rapport aux travaux présentés, est la connexion formelle entre des modèles SysML, spécifiant l'architecture des SBC, et le formalisme des automates d'interface, pour vérifier l'interopérabilité des sous-composants dans le système complet. Notre proposition est, à notre connaissance, la seule contribution proposant de concevoir des SBC en exploitant les notations SysML et le formalisme des automates d'interface. Ainsi, nous proposons des moyens aux concepteurs de SBC, pour exploiter SysML afin de concevoir rigoureusement leurs systèmes. En outre, nous proposons deux démarches de conception : l'une permet l'assemblage des composants selon une architecture SysML définie, pour construire le système global, l'autre repose sur une spécification abstraite du système à construire, et propose de sélectionner, en vérifiant une relation de raffinement, un ensemble de composants tels que leur composi-

tion respecte les contraintes imposées par la spécification abstraite.

2.3/ FORMALISME DES AUTOMATES D'INTERFACE

Les automates d'interface (IAs) ont été introduits par Luca de Alfaro et Thomas Henzinger [dAH01, AH05, AH01] pour spécifier les interfaces des composants, dans le but de vérifier leur compatibilité en considérant leurs protocoles d'interaction. Ces protocoles sont décrits par des séquences d'actions, décomposées en trois sous ensembles : les actions d'entrée (identifiées par ' ?'), de sortie (identifiées par ' !'), et internes (identifiées par ' !').

Définition 2.1 (*Automate d'interface*)**.** *Un* automate d'interface *A est représenté par le tuple* $\langle S_A, \iota_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *tel que :*

— S_A *est l'ensemble fini d'états. A est dit "vide" si* $S_A = \emptyset$*.*

— $\iota_A \in S_A$ *est l'état initial.*

— Σ_A^I, Σ_A^O *et* Σ_A^H *représentent, respectivement, les ensembles des actions d'entrée, de sortie, et internes. L'ensemble des actions de A est noté par* $\Sigma_A = \Sigma_A^I \cup \Sigma_A^O \cup \Sigma_A^H$*. Ces ensemble sont mutuellement disjoints.*

— $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions.*

2.3.1/ L'APPROCHE OPTIMISTE POUR LA VÉRIFICATION DE LA COMPATIBILITÉ

Pour assembler correctement deux composants, C_1 et C_2, spécifiés respectivement avec leurs automates d'interface, A_1 et A_2, il est nécessaire de vérifier leur compatibilité en calculant leur composition. Elle est réalisée en synchronisant leurs actions partagées d'entrée/sortie (calcul de leur produit synchrone). Cette composition peut contenir des états *bloquants*, dans lesquels l'un des deux automates sollicite une action d'entrée qui n'est pas acceptée par l'autre (concrètement, un composant qui demande un service qui n'est pas disponible dans l'autre). Dans l'approche des automates d'interface la compatibilité est établie s'il existe au moins un environnement avec lequel C_1 et C_2 interagissent sans blocage. Donc la présence d'un état bloquant n'est pas une preuve suffisante pour décider de leur incompatibilité. En effet, il suffit de trouver un environnement, appelé *légal*, donc un troisième composant C_3, spécifié avec l'automate A_3 tel que ce dernier active uniquement certaines actions d'entrée dans la composition de A_1 et A_2, permettant d'éviter leurs états bloquants. Cependant, les deux automates sont dits incompatibles s'il n'y a aucun environnement permettant d'éviter que leur composition atteigne des états bloquants. Cette approche est considérée optimiste contrairement à certaines approches, comme celle présentée dans [BMSH10], dites pessimistes, qui décide de l'incompatibilité de deux composants, si au moins un état bloquant est détecté dans leur produit synchrone. Lors de la conception des SBC, la vision optimiste est plus naturelle, car plus adaptée à la construction incrémentale des systèmes, ce qui justifie notre choix pour le formalisme des automates d'interface.

La vérification de la compatibilité de A_1 et A_2 dans l'approche optimiste est effectuée en suivant les étapes suivantes :

1. vérifier que A_1 et A_2 sont composables ;
2. calculer le produit synchrone $A_1 \otimes A_2$;
3. calculer l'ensemble des états bloquants dans $A_1 \otimes A_2$;
4. calculer l'ensemble des états incompatibles dans $A_1 \otimes A_2$: les états à partir desquels l'environnement ne peut pas éviter d'atteindre les états bloquants en une ou plusieurs étapes ;
5. calculer $A_1 \parallel A_2$ en enlevant de $A_1 \otimes A_2$, les états bloquants, les états incompatibles et les états non atteignables à partir de l'état initial ;
6. si $A_1 \parallel A_2$ n'est pas vide alors A_1 et A_2 sont compatibles, sinon ils sont incompatibles.

L'algorithme pour calculer la composition de deux automates d'interface A_1 et A_2 (dont les étapes sont décrites ci-dessus), est d'une complexité linéaire par rapport à la taille du produit des deux automates [dAH01]. Cet algorithme est implémenté dans l'outil Ptolemy [LX04], développé à l'université de Berkeley.

Les étapes de cet algorithmes sont détaillées dans la section suivante.

2.3.2/ Composabilité, compatibilité et composition

Avant de calculer le produit synchrone de deux automates d'interface, il faut vérifier leur composabilité.

Définition 2.2 (*Composabilité*). *Deux automates d'interface A_1 et A_2 sont composables si*

$$\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma^{\mathrm{H}}_{A_2} = \emptyset.$$

Nous notons par $Shared(A_1, A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$ l'ensemble des actions partagées par A_1 et A_2.

Définition 2.3 (*Produit synchrone* $\otimes$). *Soient A_1 et A_2 deux automates d'interface composables, le* produit synchrone *de A_1 et A_2 est l'automate d'interface $A_1 \otimes A_2$ défini par*

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ *et* $\iota_{A_1 \otimes A_2} = (\iota_{A_1}, \iota_{A_2})$;
- $\Sigma^{\mathrm{I}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^{\mathrm{O}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus Shared(A_1, A_2)$;
- $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup Shared(A_1, A_2)$;
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ *si*
  - $a \notin Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$ *ou*
  - $a \notin Shared(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$ *ou*
  - $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$.

L'incompatibilité entre deux automates d'interface A_1 et A_2 pourrait se produire à cause de l'existence des états (s_1, s_2) dans le produit $A_1 \otimes A_2$ tels qu'il existe au moins une action a dans $Shared(A_1,A_2)$ activable à partir de s_1 et elle ne l'est pas à partir de s_2 ou vice versa. Ces états sont dits *bloquants* dans le produit $A_1 \otimes A_2$.

Définition 2.4 (*États bloquants*)**.** *L'ensemble des* états bloquants $Dead(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ *dans* $A_1 \otimes A_2$ *est défini par* $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid \exists\, a \in Shared(A_1, A_2)$ *telle que la condition suivante est satisfaite :* $(a \in \Sigma^{O}_{A_1}(s_1) \wedge a \notin \Sigma^{I}_{A_2}(s_2)) \vee (a \in \Sigma^{O}_{A_2}(s_2) \wedge a \notin \Sigma^{I}_{A_1}(s_1))\ \}$

Pour décider de la compatibilité de A_1 et A_2, il faut pour prouver l'existence d'un environnement légal, sans pour autant le définir. Pour cela, il suffit de calculer la composition de A_1 et A_2 [AH05] et de montrer qu'elle n'est pas vide. Cette composition est calculée en enlevant de l'ensemble des transitions du produit toutes les transitions étiquetées avec une action d'entrée, qui ont un état cible incompatible (état permettant d'atteindre les états bloquants) [AH05]. Autrement dit, on obtient la composition en limitant le produit $A_1 \otimes A_2$, à l'ensemble des états compatibles et atteignables, noté $Cmp(A_1, A_2)$, après la suppression des états incompatibles. Cet ensemble contient les états dans lesquels les états bloquants ne sont pas atteignables en activant des actions de sortie ou internes (parce qu'elles sont localement contrôlables par le composant).

Définition 2.5 (*Composition* ||)**.** *La* composition $A_1 \parallel A_2$ *est l'automate d'interface tel que :*

— $S_{A_1 \| A_2} = Cmp(A_1, A_2)$;

— $\iota_{A_1 \| A_2} = (\iota_{A_1}, \iota_{A_2})$, $\iota_{A_1 \| A_2} \in S_{A_1 \| A_2}$;

— $\Sigma^{I}_{A_1 \| A_2} = \Sigma^{I}_{A_1 \otimes A_2}$, $\Sigma^{O}_{A_1 \| A_2} = \Sigma^{O}_{A_1 \otimes A_2}$, *et* $\Sigma^{H}_{A_1 \| A_2} = \Sigma^{H}_{A_1 \otimes A_2}$;

— $\delta_{A_1 \| A_2} = \delta_{A_1 \otimes A_2} \cap (Cmp(A_1, A_2) \times \Sigma_{A_1 \| A_2} \times Cmp(A_1, A_2))$.

Définition 2.6 (*Compatibilité*)**.** A_1 *et* A_2 *sont* compatibles *s'ils sont composables et si* $A_1 \parallel A_2 \neq \emptyset$

2.4/ SPÉCIFICATION DE L'ARCHITECTURE D'UN SBC AVEC SYSML

Dans cette section, nous présentons nos modèles SysML pour spécifier l'architecture d'un SBC. Cette proposition tient compte de la définition du langage SysML à partir de la version 1.3, et des caractéristiques d'un SBC.

Pour illustrer notre proposition, nous présentons l'étude de cas d'une voiture CyCab, qui est un système à base de composants ([BGMPG99]), développé par l'INRIA [1], et considéré comme étude de cas dans le projet ANR TACOS. Le CyCab est un petit véhicule, électrique et automatique, utilisé comme moyen de transport conçus essentiellement pour le transport autonome. Il permet aux utilisateurs de se déplacer via un ensemble de stations préinstallées. Il est totalement contrôlé par un système informatique et peut être piloté automatiquement selon de nombreux modes. Pour illustrer notre approche, nous considérons les contraintes suivantes :

— Un CyCab a une route dédiée où les stations sont équipées de capteurs et d'unités de calcul. Il n'y a pas d'obstacle sur la route.

— Nous proposons que la conduite du CyCab soit guidée par les informations reçues des stations, ce qui permet de localiser le CyCab par rapport aux stations.

— Le véhicule est activé par le composant de démarrage (*Starter*). Le véhicule est également doté d'un bouton d'arrêt d'urgence, associé au composant d'arrêt d'urgence (*Emergency Halt (EH)*). Le bouton d'arrêt d'urgence peut être activé à tout moment pendant les mouvements du CyCab.

1. Institut National de Recherche en Informatique et Automatique

Le CyCab est un système composé de deux composants composites : le véhicule et la station. Le véhicule est, à son tour, composé des composants élémentaires : *Vehicle Core*, *Starter* et *EH*. La station est composée des composants : *Sensors* (capteurs) et *CU* (unité de calcul).

Donc pour spécifier l'architecture d'un SBC, en l'occurrence ici le CyCab, nous proposons d'utiliser les diagrammes SysML suivants : le bloc, le diagramme de définition de blocs (BDD), le diagramme de blocs internes (IBD), et le diagramme de séquence.

Le bloc : Permet de définir un composant en décrivant, ses opérations internes, comme opérations privées du bloc, et aussi ses services requis et offerts (sous forme d'opérations également), de la façon suivante. Chaque bloc est décoré par deux ports proxy : un port d'entrée pour décrire les services offerts qui sont listés dans un bloc d'interface qui défini le type du port, et un port de sortie pour décrire de la même manière les services requis. Par ailleurs, nous distinguons deux types de blocs : les composites et les élémentaires. Par exemple dans la figure2.1, est présenté le bloc composite *Station* et ses sous-blocs élémentaires *Sensors* et *CU*. Dans la même figure, nous constatons également que le bloc *Sensors* reçoit des informations de la position du véhicule via l'opération *spos()* sur son port d'entrée *ps_in*, comme indiqué dans le bloc d'interface *IinSensors*, et renvoie la position géographique calculée via un appel à l'opération *pos()* décrite dans le bloc d'interface *IoutSensors*

Le BDD : Permet de décrire la structure du système ou d'un bloc composite, en exhibant les relations de hiérarchie entre ses blocs (relation de composition). Par exemple, la figure 2.1 représente le BDD du bloc composite *Station*, qui est relié par des relations de composition avec ses sous-blocs. Le BDD complet du CyCab est présenté dans notre papier [CH11b].

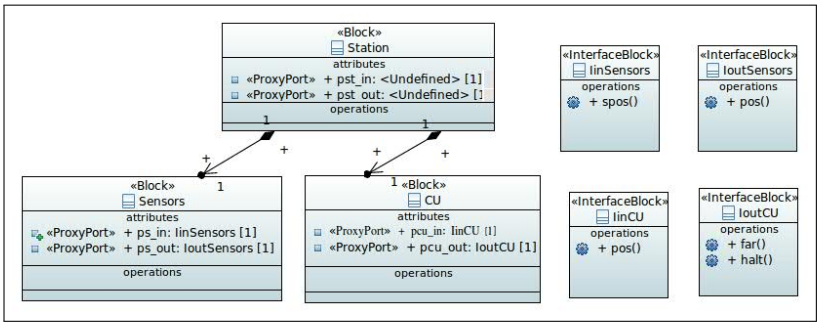


FIGURE 2.1 – Le BDD du bloc composite *Station*

IBD : Permet au concepteur d'affiner l'architecture du système en détaillant les interactions entre les sous-blocs dans un bloc composite. Dans l'IBD, les parties (*parts*) correspondent aux sous-blocs. L'interaction entre les sous-blocs est modélisée par des connecteurs entre leurs ports. Par exemple dans la figure 2.2, nous spécifions la composition interne du bloc *Station* en montrant l'interaction entre les sous-blocs *Sensors* et *CU*. Ils sont liés via un connecteur entre le port *ps_out* du bloc *Sensors* et le port *pcu_in* du bloc *CU*.

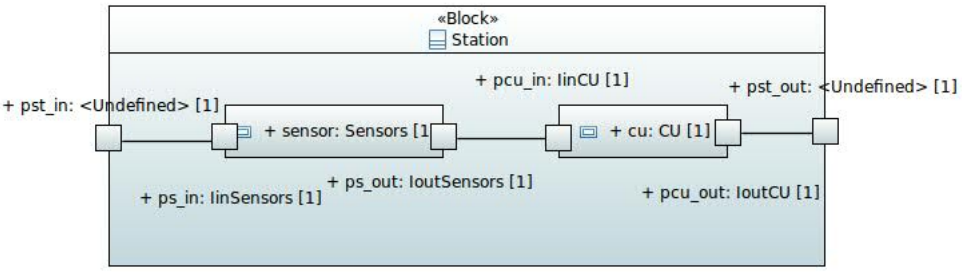


FIGURE 2.2 – Le IBD du bloc *Station*

Le diagramme de séquence : Permet de représenter les protocoles d'interaction entre chaque composant et son environnement (les autres composants), sous forme d'une séquence d'échanges de messages. Nos diagrammes de séquence doivent être composés de deux lignes. Une représentant le composant, et une autre pour l'environnement. Par exemple, dans la figure 2.3, est modélisé le diagramme de séquence du composant *Sensors* modélisant son protocole d'interaction. Ce composant reçoit le message *spos()* de l'environnement, représentant les informations de la position du véhicule, et renvoie ensuite le message *pos()*, indiquant les coordonnées géographiques, à l'unité de calcul (CU), représentée par l'environnement.

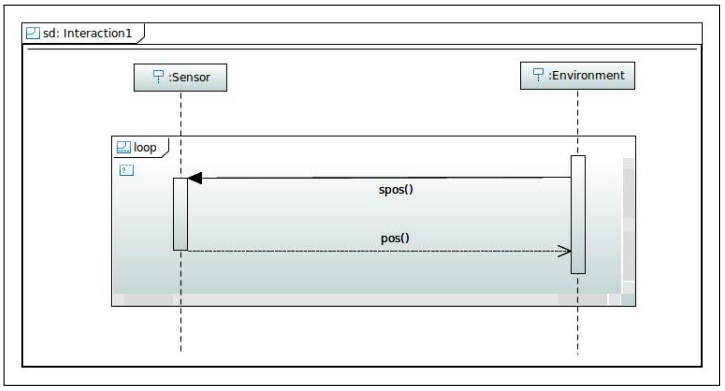


FIGURE 2.3 – Le diagramme de séquence du bloc Sensors

2.5/ DÉRIVATION AUTOMATIQUE DES AUTOMATES D'INTERFACE À PARTIR DE SYSML

Dans cette section, nous présentons notre démarche pour la dérivation des automates d'interface à partir des diagrammes de séquence spécifiant les protocoles des composants. Nous proposons une approche de transformation de modèles exploitant *Atlas Transformation Language (ATL)* [atl], qui repose principalement sur la méta-modélisation [dLVA04] et les transformations de méta-modèles [CH03]. Elle consiste à définir les méta-modèles des modèles source et cible, puis à spécifier les correspondances entre eux dans le méta-niveau. Pour automatiser la transformation, nous avons défini un ensemble de règles ATL permettant d'effectuer cette transformation.

La figure 2.4 présente un aperçu de notre approche. Partant, des diagrammes de

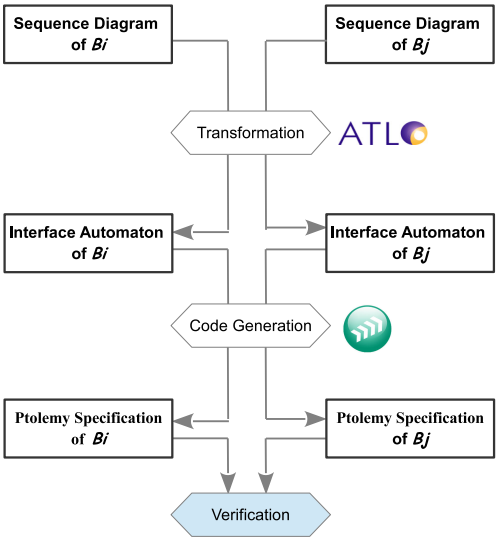


FIGURE 2.4 – Illustration de notre approche.

séquence des composants, et grâce à un ensemble de règles ATL, définies sur les méta-modèles des diagrammes de séquence et des automates d'interface, nous obtenons les automates d'interface correspondant aux diagrammes de séquence. Ensuite pour vérifier la compatibilité des composants, nous proposons de générer automatiquement, grâce à un ensemble de modèles Acceleo [acc], les spécifications d'entrée pour l'outil Ptolemy pour la vérification de la compatibilité.

Méta-modèle : Dans la figure 2.5, est présenté le méta modèle des automates d'interface. Ainsi, chaque automate d'interface est composé d'un ensemble d'état et transitions. Les ports *Inport* et *outport* sont associés respectivement aux actions d'entrée et de sortie. Le méta modèle des diagramme de séquence n'est pas présenté ici (disponible dans [BCHM19]).

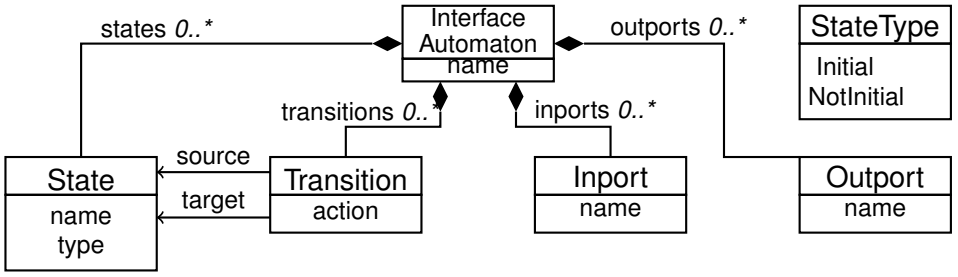


FIGURE 2.5 – Le méta modèle d'un automate d'interface.

Règles ATL : Pour traduire le modèle source correspondant à un diagramme de séquence vers le modèle cible correspondant à un automate d'interface, nous avons défini un ensemble de règles ATL permettant l'automatisation de cette transformation. Ces règles permettent de définir l'ensemble des états, de transitions, et des actions d'un automate d'interface, en transformant les éléments constitutifs d'un diagramme de

séquence. Par exemple, ci-dessous est présentée une des règles ATL (l'ensemble des règles est présenté dans [BCHM19]) :

— **Rule :** *Message2Transition*

Cette règle permet de créer une transition pour chaque message échangé entre un bloc et son environnement, décrit dans le diagramme de séquence. Cette transition sera étiquetée avec l'action *mes*, correspondant au nom du message. Et, en fonction de la position du message, la transition sera typée en tant qu'action d'entrée, de sortie ou interne.

```
rule message2Transition {
from mes : SD!Message, mos : SD!MessageOccurrenceSpecification
.            (mos.getCovered().name <> 'ENV' )
to t : IA!Transition (
action <- mes.name.concat(
      if(mes.sendEvent.getCovered()=mes.receiveEvent.getCovered())then ';'
       else    if (mes.sendEvent=mos)then '!' else '?'endif  endif),
source <- thisModule.resolveTemp(mos, 's'),
target <- thisModule.resolveTemp(
            thisModule.NextMsgOcSpec(mos.getCovered(), mos), 's')
) }
```

Templates Acceleo pour Ptolemy : Après avoir dérivé les automates d'interface du diagramme de séquence, nous proposons un ensemble de *templates* Acceleo pour générer automatiquement la spécification d'entrée Ptolemy, afin de vérifier la compatibilité des composants. En analysant un fichier d'entrée de Ptolemy spécifiant un automate d'interface, nous avons défini six *templates* Acceleo permettant de créer automatiquement le fichier de la spécification Ptolemy. Ces derniers sont détaillés dans notre papier [BCHM19] et la thèse [Bou16].

En appliquant notre approche sur les diagrammes de séquence des bloc *Sensors* et *CU*, on obtient leurs automates d'interfaces respectifs illustrés dans la figure 2.6.

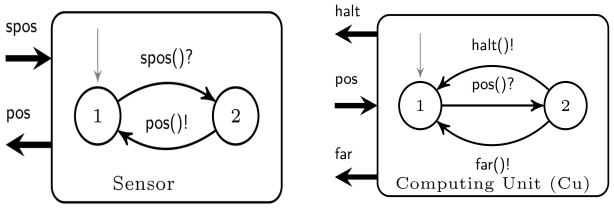


FIGURE 2.6 – Les automates d'interface des sous-composants *Sensors* et *CU*

2.6/ VÉRIFICATION DE L'ASSEMBLAGE DANS UN SBC MODÉLISÉ AVEC SYSML

Dans cette section, nous présentons notre approche pour vérifier la compatibilité d'un ensemble de composants qui forment un SBC dont l'architecture est décrite avec SysML. Cette approche exploite les travaux présentés précédemment, et prend donc en considération les protocoles d'interaction spécifiés avec des diagrammes de séquence et

traduits en automates d'interface. Elle exploite également les modèles formels des BDD et IBD, que nous présentons dans la section suivante.

2.6.1/ MODÈLES FORMELS DES DIAGRAMMES SYSML SPÉCIFIANT L'ARCHITECTURE D'UN SBC

Pour pouvoir analyser automatiquement l'architecture d'un SBC, spécifiée avec SysML, afin de vérifier l'assemblage de l'ensemble de ses composants, nous proposons les modèles formels pour spécifier l'ensemble des diagrammes SysML utilisés pour spécifier l'architecture. Nous allons ainsi définir formellement le bloc, le diagramme de définition de blocs, le bloc d'interface, et le diagramme de blocs internes. Ces modèles sont exploités par notre algorithme de vérification de l'assemblage des composants, décrit dans la section 2.6.2.

Définition 2.7 (BDD)**.** Un modèle formel pour le BDD d'un système S est : $BDD_S = \langle IB_S, SB_S, SubB_S \rangle$ *où :*

- IB_S *: est le bloc initial du système ;*
- SB_S *: est l'ensemble de blocs ;*
- *la fonction* $SubB_S : SB_S \rightarrow 2^{SB_S}$ *qui associe à chaque bloc son ensemble de sous-blocs.*

Définition 2.8 (Port d'un bloc)**.** *Un port d'un bloc SysML,* P*, est défini par le tuple* $\langle name_P, type_P, direction_P \rangle$*, tel que :*

- $name_P$ *est le nom du port,*
- $type_P$ *est le nom du bloc d'interface qui type le port,*
- $direction_P$ *indique la nature du port :* in *pour un port d'entrée et* out *pour un port de sortie. Nous notons par* $P.direction$ *la direction du port.*

Pour la définition suivante, nous avons besoin de considérer l'ensemble $IntB$ qui représente les blocs d'interface.

Définition 2.9 (Bloc SysML)**.** *Un bloc SysML* B *est un tuple* $\langle name_B, Op_B, P_{inB}, P_{outB}, TypePort_B \rangle$*, tel que :*

- $name_B$ *est le nom du bloc,*
- Op_B *est l'ensemble fini des opérations privées dans* B*,*
- P_{inB} *le port proxy d'entrée de* B*,*
- P_{outB} *le port proxy de sortie de* B*.*
- *La fonction* $TypePort_B : \{P_{inB}, P_{outB}\} \rightarrow IntB$*, qui détermine l'interface qui définit le type de chaque port.*

Nous notons par $B.name$, $B.Op$, $B.P_{in}$, et $B.P_{out}$, respectivement, le nom du bloc B, son ensemble des opérations internes, son port d'entrée, et son port de sortie.

Définition 2.10 (Bloc d'interface)**.** *Soit B un bloc SysML et P_x un port de B, où x prend la valeur de in pour un port d'entrée, et out pour un port de sortie. Un bloc d'interface de B, tel que B définit le type du port P_x, et représente l'interface requise ou fournie de B, est défini par $IxB = \langle name_{xB}, Op_{xB} \rangle$, tel que $name_{xB}$ est le nom de l'interface, et Op_{xB} définit l'ensemble des services requis par B quand $x = out$, et dans le cas contraire ($x = in$) il définit l'ensemble des services fournis.*

Soit I un bloc d'interface, nous notons par $I.Op$ l'ensemble des opérations définies dans I.

Par exemple, dans la figure 2.1,l'interface requise du bloc CU est $IoutCU$, telle que $IoutCU.Op = \{far(), halt()\}$. Son interface offerte est $IinCU$, telle que $IinCU.Op = \{pos()\}$.

Définition 2.11 (IBD)**.** *Soit $SetB_B = SubB_B(B)$ l'ensemble des sous-blocs du bloc B. Le diagramme de blocs internes de B, est le tuple*

$IBD_B = \langle Parts_B, BlockPart_B, Ports_B, Pextin_B, Pextout_B, PinPart_B, PoutPart_B, Connectors_{inB}, Connectors_{outB} \rangle$ *tel que :*

- $Parts_B$ *est l'ensemble des parties (Parts), tel que chaque partie représente une instance d'un sous-bloc dans B.*
- *La fonction $BlockPart_B : Parts_B \rightarrow SetB_B$ détermine pour chaque Part dans l'IBD son bloc.*
- $Ports_B$ *est l'ensemble des ports associés aux parties dans l'IBD.*
- $Pextin_B$ *et $Pextout_B$ sont respectivement le port extérieur d'entrée et le port extérieur de sortie.*
- *La fonction $PinPart_B : Parts_B \rightarrow Ports_B$ détermine le port d'entrée pour chaque Part dans l'IBD. Et la fonction $PoutPart_B : Parts_B \rightarrow Ports_B$ détermine le port de sortie pour chaque Part dans l'IBD.*
- $Connectors_{inB} = \{(p_i, p_j)\ tel\ que\ (p_i, p_j) \in Ports_B \times Ports_B \wedge p_i.direction \neq p_j.direction\}$, *l'ensemble des connecteurs internes qui relient les sous-blocs. Ils sont également appelés connecteurs d'assemblage.*
- $Connectors_{outB} = \{(p_i, p_j)\ tel\ que\ (p_i, p_j) \in Ports \times Ports \wedge p_i.direction = p_j.direction\}$, *l'ensemble des connecteurs qui relient les ports des sous-blocs (parties) avec les ports du bloc composite. Ils sont appelés connecteurs de délégation.*

Les définitions du BDD et de l'IBD sont exploitées dans l'algorithme de la section suivante, pour identifier et assembler les sous-composants connectés dans le système global ou dans un composant composite. Les autres définitions sont exploitées dans la section 2.7 et les chapitres suivants.

2.6.2/ ALGORITHME DE VÉRIFICATION

L'algorithme 1, $SysCompos$, exploite les modèles formels décrits dans la section précédente et l'approche des automates d'interface pour vérifier l'assemblage de l'ensemble des composants d'un SBC décrit avec SysML. Nous notons par $Compos(A_1, A_2)$, l'algorithme permettant de vérifier la comptabilité de deux automates d'interface, dont les étapes sont décrites dans la section 2.3.1. Cet algorithme est détaillé dans [dAH01].

L'algorithme 1 accepte en entrée le modèle formel du BDD du SBC (il exploite aussi le IBD de chaque bloc composite et l'automate d'interface de chaque bloc élémentaire), et produit en sortie l'automate d'interface de bloc composite du système décrit par son BDD si les composants sont compatibles, ou un automate vide dans la cas contraire. L'algorithme analyse récursivement l'architecture du système en exploitant les liens hiérarchiques entre les blocs (dans le BDD) et leurs liens internes (connecteurs) dans les IBD, et vérifie au fur et à mesure, de son avancée, la compatibilité des blocs SysML.

Algorithme 1 : SysCompos(BDD_B)

ENTREES :
BDD_B le BDD du système ou du bloc composite B
SORTIES :
l'automate d'interface du composant composite si l'assemblage est correct, ou un automate d'interface vide sinon

DEBUT
Soit bc, le bloc courant, tel que $bc = IB_B$
Si $SuB_B(bc) = \emptyset$ **Alors**// LE BLOC COURANT N'A PAS DE SOUS-BLOCS
  Soit IA_{bc}, l'automate d'interface obtenu à partir du diagramme de séquence SD_{bc}
  Retourner IA_{bc};
SINON
  //ANALYSE DE L'IBD DU BLOC COURANT POUR CALCULER LA COMPOSITION DES SOUS-BLOCS DU BLOC COURANT
  Soit $IBD_{bc} = \langle Parts_{bc}, BlockPart_{bc}, Ports_{bc}, Pextin_{bc}, Pextout_{bc}, PinPart_{bc}, PoutPart_{bc}, Connectors_{inbc}, Connectors_{outbc}\rangle$ le IBD de bc
  et $setBlocs = \{bi \mid \exists p_i \in Parts_{bc} \wedge (p_i, b_i) \in BlockPart_{bc}\}$; $setCon = Connectors_{inbc}$;
  RÉPÉTER
    Soit $b_i \in setBlocs$,
    SI $(SuB_B(b_i) \neq \emptyset)$ **ALORS**
      $IA_{b_i} = SysCompos(BDD_{b_i})$;
    SINON
      Soit IA_{b_i}, l'automate d'interface obtenu à partir du diagramme de séquence SD_{b_i}
    FIN SI
    TANT QUE $\exists\{p_i, p'_i\} \in SetCon$ tel que $(p'_i, b'_i) \in BlockPart_{bc}$ **FAIRE**
      SI $(SuB(b'_i) \neq \emptyset)$ **ALORS**
        $IA_{b'_i} = SysCompos(BDD_{b'_i})$
      SINON
        Soit $IA_{b'_i}$, l'automate d'interface obtenus du diagramme de séquence $SD_{b'_i}$
      FIN SI
      $IA_{b_i} = Compos(IA_{b_i}, IA_{b'_i})$
      $IA_{b'_i} = IA_{b_i}$ // L'AUTOMATE DE b'_i DEVIENT ÉGALEMENT CELUI DE LA COMPOSITION
      SI $(IA_{b_i} = \emptyset)$ **ALORS** Exit();**FIN SI** // INCOMPATIBILITÉ DES SOUS-BLOCS
        $SetCon = SetCon - \{(p_i, p'_i)\}$;
      FIN TANT QUE;
      $setBlocs = setBlocs - b_i$;
  JUSQU'A $setBlocs = \emptyset$;
  Retourner IA_{b_i};
Fin Si
FIN

En appliquant l'algorithme 1 sur le bloc $Station$, avec le paramètre approprié

($SysCompos(BDD_{Station})$), on obtient l'automate d'interface décrit dans la figure 2.7, qui représente l'automate de la composition des automates d'interface des sous-blocs $Sensors$ et CU. Ce qui montre que les sous-blocs dans le bloc composite $Station$, sont compatibles.

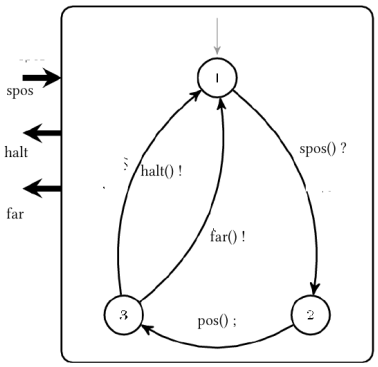


FIGURE 2.7 – L'automate d'interface de la composition de *Sensors* et *CU*

2.7/ CONCEPTION INCRÉMENTALE D'UN SBC PAR DES RAFFINEMENTS SUCCESSIFS

Dans cette section, nous présentons notre approche pour le développement d'un SBC (ou d'un composant composite) ou d'un composant composite, en considérant sa spécification SysML abstraite, modélisant sa structure et son comportement (ce qui marque une différence par rapport à l'approche présentée dans la section précédente). Et ensuite, grâce à une relation de raffinement, nous proposons un ensemble de composants élémentaires réutilisables, ou abstraits (à construire également), tels que leur composition satisfait la spécification abstraite du système. La sélection de l'ensemble de composants peut se faire, soit en une seule étape, ou progressivement en plusieurs étapes. Dans le premier cas, le système à construire n'est pas très complexe, donc il suffit d'assembler d'une manière incrémentale des composants élémentaires pour le construire. Par contre, le deuxième cas concerne la construction d'un système complexe, où l'on a besoin de définir des sous-composants abstraits et de les construire en assemblant d'autres composants, et enfin les assembler à leur tour pour construire le système final. Dans ce cas la démarche de conception est aussi incrémentale.

Spécification SysML du composant composite abstrait et des autres composants : Pour modéliser le composite à construire, nous proposons les modèles SysML suivants : le BDD et le IBD, pour définir son architecture ; des blocs d'interface pour décrire ses services offerts et requis ; un diagramme de séquence pour décrire son protocole d'interaction. Ces diagrammes sont utilisés également pour spécifier les composants abstraits à utiliser pour construire le système final. Par contre, pour les composants élémentaires à utiliser, nous avons besoin uniquement de leurs diagrammes de séquence et de leurs blocs d'interface.

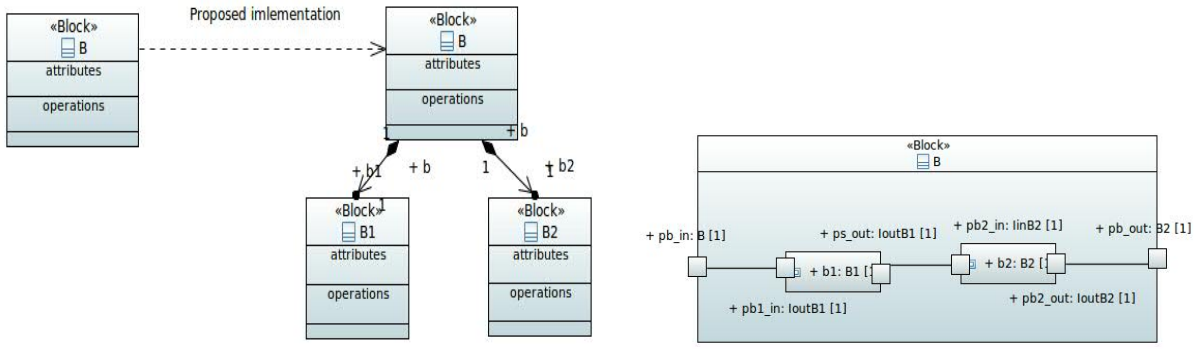


FIGURE 2.8 – Illustration de l'approche

Démarche de vérification par raffinement : Pour vérifier si une composition d'un ensemble de composants satisfait les contraintes imposées par la spécification abstraite d'un composant composite, nous avons proposé dans [Roz15], de vérifier une relation de **raffinement** entre le bloc SysML du composite abstrait et les blocs SysML correspondant aux composants faisant partie de la composition.

Définition 2.12 (Raffinement d'un bloc SysML composite avec une composition de blocs)**.** *Soit B un bloc abstrait décrit avec les diagrammes BDD_B, IBD_B, et un diagramme de séquence. Soient $B_1, ..., B_n$ les sous-blocs du B, décrits avec leurs modèles SysML. La composition de $B_1, ..., B_n$ selon le diagramme IBD_B raffine le bloc abstrait B ssi, il y a une cohérence au niveau structurel et comportemental entre la composition des sous-blocs B_i et le composite B.*

Les deux conditions de la relation de raffinement sont détaillées ci-dessous :

1. **Cohérence au niveau structurel** entre la spécification SysML abstraite du composant composite et la composition des composants sélectionnés : vérifier que l'assemblage des composants est compatible, et le composite obtenu de leur composition offre au moins les mêmes services et requiert au plus les mêmes services, que le composite abstrait. Par exemple, dans la figure 2.8, nous proposons de construire le bloc composite B avec la composition des blocs élémentaires $B1$ et $B2$. Donc il faut vérifier que $B1$ et $B2$ sont compatibles, que que la composition des blocs $B1$ et $B2$ est cohérente avec le bloc composite B, vis à vis des services requis et offerts.
2. **Cohérence au niveau comportemental** : il faut vérifier la cohérence entre le protocole d'interaction du composant composite abstrait et celui de la composition des composant utilisés.

Il faut noter que si la première condition n'est pas vérifiée, alors la deuxième ne l'est pas également.

Ces deux conditions sont détaillées ci-dessous.

Cohérence au niveau structurel : la cohérence au niveau structurel est formalisée par la définition suivante.

Définition 2.13 (Cohérence d'un bloc composite avec ses sous-blocs)**.** *Soit B un bloc abstrait décrit avec les diagrammes BDD_B, et IBD_B. Soient $B_1, ..., B_n$ les sous-blocs du B.*

La composition de $B_1, ..., B_n$ selon le diagramme IBD_B est cohérente au niveau structurel avec B ssi :

- ***Condition 1 (Composabilité)*** *:*
  Pour chaque couple de blocs connectés $\{B_i, B_j\}$, nous avons :
  $IinB_i.Op \cap IinB_j.Op = IoutB_i.Op \cap IoutB_j.Op = B_i.Op \cap (B_j.Op \cup IinB_j.Op \cup IoutB_j.Op) = B_j.Op \cap (B_i.Op \cup IinB_i.Op \cup IoutB_i.Op) = \emptyset$.
- ***Condition 2 (Au moins les mêmes entrées)*** *:*
  Pour chaque sous-bloc B_i, de B, tel que B_i (d'après IBD_B) est connecté, par délégation à un port d'entrée de B, nous avons : $IinB.Op \subseteq IinB_i.Op$.
- ***Condition 3 (Au plus les mêmes sorties)*** *:*
  Pour chaque sous-bloc B_i, de B, tel que B_i (d'après IBD_B) est connecté, par délégation à un port de sortie de B, nous avons : $IoutB_i.Op \subseteq IoutB.Op$.
- ***Condition 4 (Compatibilité)*** *:*
  Les sous-blocs $B_1, ..., B_n$ sont compatibles, d'après leurs connections décrites dans IBD_B .

La première condition garantit la composabilité des blocs connectés pour pouvoir utiliser l'approches des automates d'interfaces pour vérifier leurs assemblage. La deuxième garantit que la composition des sous-blocs offre au moins les mêmes services que le bloc composite. Quant à la troisième, elle garantit que la composition des sous-blocs requiert au plus les mêmes services que le bloc composite. Et enfin, la dernière condition garantit la compatibilité de l'ensemble des sous-blocs (cette condition est vérifiée grâce aux automates d'interface obtenus à partir des diagrammes de séquence des sous-blocs).

Dans [Roz15, CCM12a], nous avons proposé un algorithme, qui exploite les contributions présentées dans les sections précédentes (passage d'un diagramme de séquence vers un automate d'interface), permettant de vérifier l'ensemble des conditions définies. L'algorithme exploite les modèles formels du BDD, de l'IBD du bloc composite, et des blocs d'interface du bloc composite et des sous-blocs.

Cohérence au niveau comportemental : Pour vérifier la cohérence au niveau comportemental, nous proposons de vérifier la relation de raffinement entre l'automate d'interface correspondant au diagramme de séquence du composant composite, et celui obtenu de la composition des composants utilisés. En effet, le raffinement des automates d'interface est basé sur la relation de simulation alternée définie dans [dAH01, AH05]. Par conséquent, il est défini de manière contravariante pour les actions d'entée/sortie : une implémentation doit accepter toutes les entrées acceptées par la spécification et ne produire que les sorties autorisées par celle-ci [PdAHSV02]. Ainsi, le composant raffiné peut être utilisé dans un environnement plus contraint que celui de son abstraction. Autrement dit, le composant abstrait est substituable par sa version raffinée.

Avant de définir la relation de raffinement, nous définissons la simulation alternée en se basant sur la définition proposée dans [AH05]. Nous utilisons la notation $s \xrightarrow{\tau}^{*} s'$, pour désigner dans un automate d'interface une séquence de transitions étiquetées par des actions internes entre les états s et s'.

Définition 2.14 (Simulation alternée [AH05])**.** *Soient deux automates d'interface Q et P. Une relation binaire $\succeq \subseteq S_P \times S_Q$ est une simulation alternée de P par Q, si pour tout*

$(p, q) \in S_P \times S_Q$ *tel que* $p \succeq q$*, nous avons :*

Si $p \xrightarrow{a?} p'$ *et* $a \in \Sigma_P^I$ *alors* $\exists q'.q \xrightarrow{a?} q'$ *et* $(p', q') \in \succeq$

Si $q \xrightarrow{a!} q'$ *et* $a \in \Sigma_q^O$ *alors* $\exists p'.p \xrightarrow{\tau}^* p'.\exists p''.p' \xrightarrow{a!} p''$ *et* $(p'', q') \in \succeq$

Si $q \xrightarrow{a;} q'$ *et* $a \in \Sigma_Q^H$ *alors* $\exists p'.p \xrightarrow{\tau}^* p'$ *et* $(p', q') \in \succeq$

Définition 2.15 (*Raffinement*[AH05])**.** *L'automate d'interface* Q *raffine l' automate d'interface* P*, noté* $P \succeq Q$*, si les conditions suivantes sont satisfaites :*

— $\Sigma_P^{\mathrm{I}} \subseteq \Sigma_Q^{\mathrm{I}}$ *et* $\Sigma_P^{\mathrm{O}} \supseteq \Sigma_Q^{\mathrm{O}}$ *;*

— *Il existe une simulation alternée* $\succeq$ *entre* Q *et* P *telle que leurs états initiaux sont reliés par cette relation :* $i_P \succeq i_Q$*.*

Nous avons montré dans [Roz15], qu'il est possible d'utiliser l'outil MIO Workbench [BMSH10] pour vérifier la relation de raffinement.

2.8/ Conclusion

Dans ce chapitre nous avons présenté nos approches qui combinent des modèles SysML et l'approche des automates d'interface pour concevoir rigoureusement un SBC dans lequel l'interopérabilité entre ses composants est garantie. Ainsi, nous apportons des solutions méthodologiques pour les concepteurs des SBC, souhaitant utiliser SysML comme langage de modélisation, et ayant le besoin de prouver formellement leur fiabilité vis à vis de l'interopérabilité de leurs composants.

2.9/ Bilan

Les résultats de ce chapitre concernent les éléments suivants :

1. **Projets** :
   — Le projet Tassili SYSVAP (Spécifier en vue de Vérifier et d'évaluer les performances des systèmes complexes), coopération avec l'université USTHB Alger (Algérie), 2013-2016.
   — Le projet région SyVAD (Modélisation SysML pour la Validation et la Vérification de micro-systèmes), période 2012-2014.
2. **Encadrement de thèses de doctorat** :
   — La thèse de Oscar Carillo [Roz15], 2015.
   — La thèse de Hamida Bouaziz [Bou16], 2016.
3. **Encadrement de stage de Master 2** :
   Stage de Samir Berrani, concernant la combinaison de SysML et Modelica pour spécifier et valider les réseaux de capteurs, 2012.
4. **Publications principales** : [CH11b, CCM12a, BCHM19].

3

ADAPTATION DES COMPOSANTS SYSML LORS DU DÉVELOPPEMENT D'UN SBC

3.1/ INTRODUCTION

Dans ce chapitre, nous proposons une approche ascendante (*Bottom-up*) pour construire un SBC, en assemblant et adaptant des composants spécifiés avec SysML (blocs), lorsque ils présentent des inadéquations (*Mismatchs*) entre leurs services. C'est une approche qui exploite les spécifications formelles des diagrammes SysML, définies dans le chapitre précédent, afin de construire d'une manière incrémentale le système dans son ensemble, en générant des adaptateurs pour faire interagir correctement les composants. Notre proposition apporte une nouvelle contribution par rapport aux travaux existants dans le domaine de l'adaptation, car nous nous focalisons sur l'utilisation conjointe de SysML et de l'approche formelle des automates d'interface, pour l'adaptation d'un ensemble de composants, pour construire des composants composites, qui sont ensuite assemblés et adaptés à leur tour, pour construire progressivement l'ensemble du système.

Le reste de ce chapitre est organisé comme suit. Dans la section suivante, nous présentons un aperçu de notre approche. La section 3.3 présente l'état de l'art et nos contributions. Dans la section 3.2, est présenté l'aperçu de notre approche. Notre proposition est détaillée dans la section 3.4. Nous terminons ce chapitre par une conclusion et un bilan dans les deux dernières sections.

3.2/ APERÇU DE L'APPROCHE

Notre objectif est de construire un SBC d'une manière ascendante, en assemblant et adaptant des composants réutilisables pour obtenir progressivement l'ensemble des composants composites qui forment le système global (voir la figure 3.1). Donc, nous générons progressivement des adaptateurs pour, d'une part orchestrer les interactions entre les composants réutilisés, et d'autre part pour répondre aux contraintes imposées par les composants composites à construire.

Dans la figure 3.2 est décrit un aperçu de notre approche que nous résumons dans les points suivants :

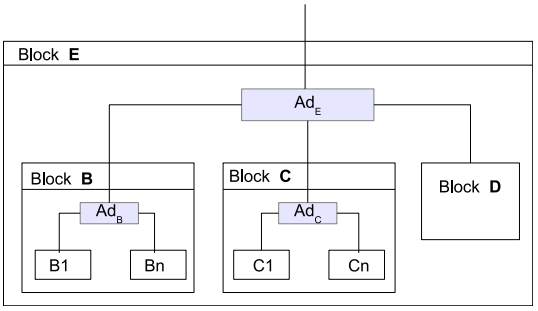


FIGURE 3.1 – Approche d'adaptation ascendante des blocs SysML

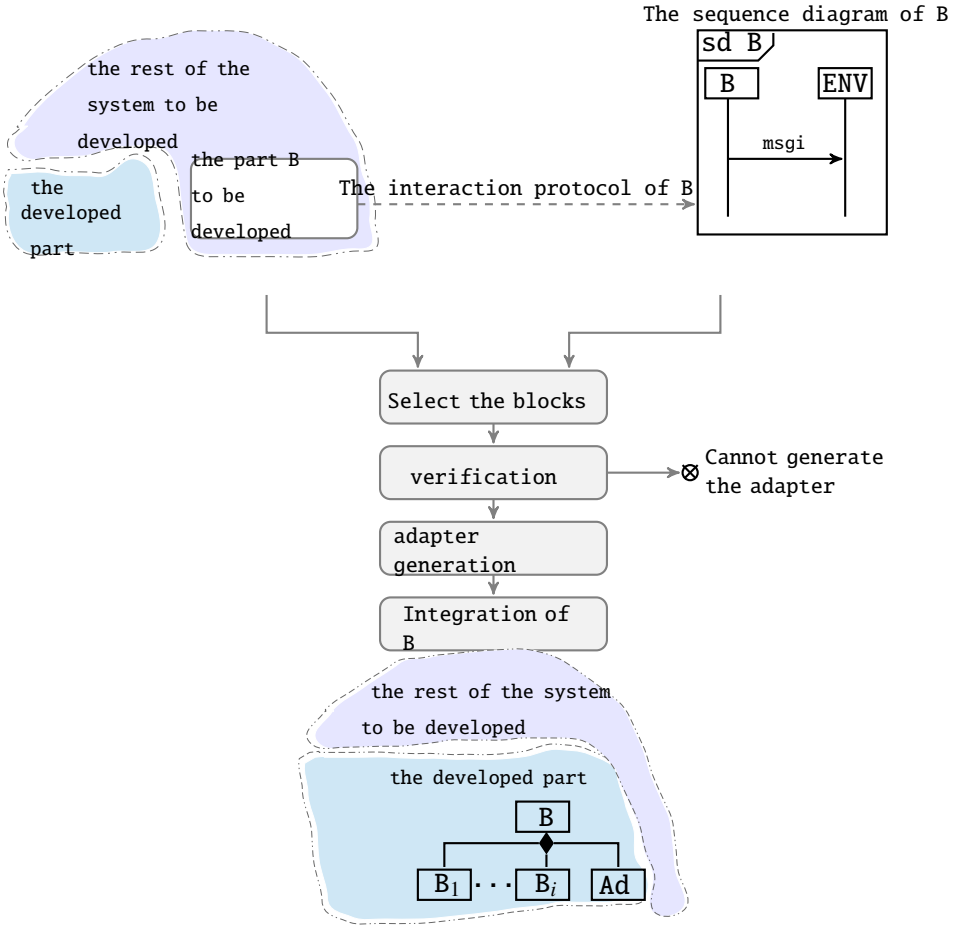


FIGURE 3.2 – Illustration de l'approche d'adaptation des blocs SysML.

— Nous commençons par spécifier la partie du système que nous souhaitons développer (le composant composite), qui sera ensuite intégrée dans le système global, en la modélisant sous forme d'un bloc composite SysML *B*, permettant d'exhiber ses ensembles de services requis et offerts. Son protocole d'interaction modélisé avec un diagramme de séquence est également fourni. Notre intérêt dans cette étape est de développer *B* avec un ensemble de blocs réutilisables assemblés avec leur adaptateur.

— Après cela, un ensemble de blocs $\{B_i\}$, modélisant des composants réutilisables, sont sélectionnés (pour construire *B*), en considérant les contraintes déduites de la

spécification du bloc *B*.

— Ensuite, en fonction de chaque bloc B_i et de la spécification SysML du bloc composite, nous pouvons décider si B_i est approprié pour intégrer l'ensemble des blocs qui vont composer *B*. Évidemment cette étape est répétée pour chaque bloc sélectionné B_i. Et à la fin, le protocole d'interaction de l'adaptateur et sa structure, modélisée avec SyML, sont calculés, en fonction des modèles SysML des blocs *B* et $\{B_i\}$. L'adaptateur permet, d'une part de résoudre les inadéquations entre les blocs B_i, et d'autre part de les compléter pour répondre aux exigences de *B* et de son environnement.

— Enfin, le bloc adaptateur est intégré aux blocs sélectionnés pour construire automatiquement le BDD et l'IBD du bloc parent *B*.

Notre approche permet de générer automatiquement l'adaptateur qui se distingue par ses deux rôles : (i) il joue un premier rôle de convertisseur, d'une part entre les composants à assembler pour construire le composant composite (la partie du système), et d'autre part entre les composants et leur environnement ; (ii) il joue un second rôle en tant que complément pour les autres composants, c'est-à-dire que lorsque l'environnement des composants demande un service qui n'est offert par aucun des composants réutilisés, l'adaptateur fourni ce service en l'implémentant. De même, lorsqu'un composant réutilisé a besoin d'un service et que celui-ci ne peut être proposé ni par les autres composants ni par l'environnement de son composant parent, l'adaptateur permet l'implémentation du service requis.

3.3/ ÉTAT DE L'ART ET CONTRIBUTIONS

Dans la littérature, plusieurs études ont été effectuées pour présenter les travaux existants ayant proposé des solutions dans le domaine de l'adaptation de logiciels [CMP06, SEG09, BBG+06, IST11]. La plupart des approches proposées sont généralement classées en deux catégories : les restrictives et les génératives. Les approches restrictives, comme celles proposés dans [Reu03, IT03], permettent de résoudre le problème d'inadéquation entre les composants en supprimant une partie de leurs comportements conduisant vers des états de blocage. Ce qui a comme conséquence la réduction des fonctionnalités des composants. Et les génératives, comme celles proposées dans [YS97, BBC05, BP06], proposent de générer un composant médiateur, appelé adaptateur, pour permettre aux composants d'interagir correctement, en renommant les noms des messages ou des services échangés, et/ou les réordonnant. La génération de l'adaptateur repose sur un contrat d'adaptation qui définit une relation entre les services des composants présentant des inadéquations. Ce sont des approches qui sont capables de résoudre plus de problèmes d'inadéquation par rapport aux approches restrictives. Néanmoins, dans le cas du réordonnement des messages, ces approches sont contraintes par des problèmes de complexité.

Les approches d'adaptation existantes présentent généralement des différences dans le formalisme utilisé pour représenter les interfaces des composants et pour modéliser leurs protocoles d'interaction. L'algèbre des processus est utilisée dans de nombreux travaux, par exemple ceux présentés dans [BBC05, AG97]. Ce choix est justifié par la capacité de l'algèbre des processus à présenter des descriptions plus expressives des protocoles et à permettre une analyse plus sophistiquée des systèmes concurrents, et

aussi à prendre en charge la dérivation formelle des propriétés de sécurité et de vivacité. Dans [KEP07], les auteurs utilisent PRES+ [CEP00], qui est une variante des réseaux de Petri temporisés, pour modéliser les interactions des composants et vérifier certaines propriétés sur le protocole de l'adaptateur. Ils ont justifié ce choix par la capacité de PRES + de capturer intuitivement les aspects de la concurrence et du temps réel. Dans [CPS06b, CPS08a, CS14, PST07, TI08], les auteurs utilisent les systèmes de transitions étiquetées pour modéliser les protocoles des composants à adapter. Dans cette catégorie, nous soulignons le travail présenté dans [TI08], qui est proche de notre proposition, car les auteurs présentent une approche exploitant les LTS , modélisant les comportements des composants, qui sont dérivés des MSC (*Message Sequence Charts*). Dans notre cas les protocoles sont spécifiés avec des automates d'interface, dérivés des diagramme de séquence.

Contributions : Notre approche comparée à celles présentées précédemment, se situe dans le contexte de la communication synchrone des composants modélisés avec SysML, où une approche générative-restrictive est adoptée pour construire le composant adaptateur capable de résoudre les inadéquations liées aux noms des services échangés entre les composants. Le formalisme des automates d'interface est utilisé pour spécifier formellement les interactions des composants réutilisés et exploiter leur composition pour la génération des adaptateurs évitant le blocage des composants. De plus, l'adaptateur est généré en fonction d'une spécification abstraite d'un composant composite contenant un ensemble de composants réutilisables, ce qui n'est pas le cas pour les travaux présentés ci-dessus. Nous pouvons considérer que notre approche introduit une nouvelle branche dans la taxonomie de l'adaptation de composants.

3.4/ L'APPROCHE D'ADAPTATION DES BLOCS SYSML

Dans les sections suivantes, nous détaillons les différentes étapes de l'approche.

3.4.1/ SPÉCIFICATION SYSML DE LA PARTIE À DÉVELOPPER

Cette étape est dédiée à la spécification de la structure et du protocole d'interaction de la partie *B*, considérée comme composant composite, que nous souhaitons développer. Structurellement, nous modélisons cette partie sous la forme d'un bloc SysML, où il faut préciser ses ports d'entrée/sortie et les blocs d'interface permettant de typer ces derniers. Ce qui permet de décrire les services requis et offerts de *B*. Pour spécifier les différentes interactions de notre bloc composite avec son environnement, nous utilisons un diagramme de séquence respectant le même formalisme décrit dans le chapitre précédent. Ce diagramme est transformé ensuite en automate d'interface avec l'approche proposée dans [CH11a], pour pouvoir l'exploiter dans les étapes suivantes dans la vérification formelle de la compatibilité et la déduction du protocole d'interaction de l'adaptateur.

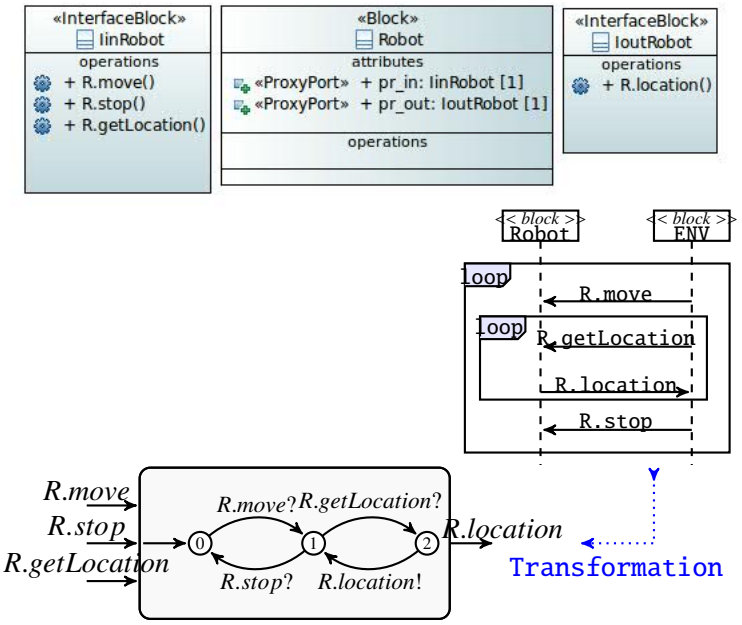


FIGURE 3.3 – Le bloc composite Robot et son protocole d'interaction.

ILLUSTRATION

Pour illustrer notre approche d'adaptation, nous proposons un exemple, simple, d'un robot guidé par une station. Pour développer ce système, composé du robot et de la station, nous commençons par construire le robot qui est une partie du système global et qui est un composant composite. Ce dernier interagit avec son environnement de la façon suivante : il commence le déplacement dès qu'il reçoit le message de son environnement, ensuite soit il reçoit le message d'arrêt ou la demande de localisation. Dans ce dernier cas, le robot envoie son emplacement à son environnement (en l'occurrence la station).

Dans la figure 3.3, est décrit le bloc *Robot*, qui représente la partie du système que nous souhaitons développer. Les blocs d'interface qui typent les ports du bloc *Robot*, montrent que les services offerts sont représentés par les opérations : *R.location()*. Ceux qui sont requis sont représentés par les opérations : *R.move(), R.stop(), R.getLocation()*. Dans la même figure, le protocole d'interaction est décrit par un diagramme de séquence qui est transformé en automate d'interface.

3.4.2/ SÉLECTION DES BLOCS RÉUTILISABLES $\{B_i\}$

Au cours de cette étape, l'architecte du système doit sélectionner un ensemble de blocs $\{B_i\}$ pour construire le bloc composite B (défini lors de l'étape précédente). Chaque bloc sélectionné doit être spécifié avec un diagramme de séquence décrivant ses interactions avec son environnement. Le résultat de cette étape est un ensemble de blocs SysML à réutiliser avec leurs diagrammes de séquence et un contrat d'adaptation C, qui spécifie les correspondances entre les services des blocs présentant des inadéquations. Cette opération doit être réalisée par l'architecte du système, en tant que connaisseur des différents composants sélectionnés ainsi que de leurs services requis et fournis. Ainsi, il pourra définir le contrat C, en indiquant pour chaque action dans un bloc, son action

correspondante dans un autre bloc.

Services fournis et requis d'un bloc : Pour spécifier certaines conditions sur les blocs, nécessaires à notre processus d'adaptation, et formaliser le contrat d'adaptation, nous avons besoin de définir pour chaque bloc B_i, ses ensembles de services requis et fournis. Soit $SetPorts$ l'ensemble de tous les ports, nous proposons de définir les ensembles suivants en considérant les formalisations des diagrammes SysML proposées dans le chapitre précédent.

— PS_{B_i} : l'ensemble des services fournis,
PS_{B_i}={ $ps \mid \exists p \in SetPorts\ et\ p = B_i.P_{in} \wedge ps \in TypePort_{B_i}(p).Op$ }

— RS_{B_i} : l'ensemble des services requis,
RS_B={ $rs \mid \exists p \in SetPorts\ et\ p = B_i.P_{out} \wedge rs \in TypePort_{B_i}(p).Op$ }

— IOp_{B_i} : l'ensemble des actions internes, IOp_{B_i}= { $o \mid o \in B_i.Op$}

Contrat d'adaptation : Le contrat d'adaptation dans un SBC sert essentiellement à résoudre les problèmes d'inadéquation entre les services des composants. Pour le spécifier, nous nous sommes inspiré du travail présenté dans [CPS08a], où un contrat d'adaptation est spécifié par un ensemble de règles. Une règle prend la forme d'un vecteur synchrone v_i, tel que le nombre d'éléments de chaque vecteur correspond au nombre de composants. Un vecteur synchrone v_i pour un ensemble de composants ($\{C_i\}_{i\in\{1..n\}}$) est un tuple :

$$\langle\ e_1, \ldots, e_i, \ldots, e_n\ \rangle.$$

Avec e_i appartenant à l'ensemble des actions du composant C_i. Elle est égale, soit à ε, dans le cas où C_i ne participe pas à cette synchronisation (n'est pas concerné par l'adaptation), soit à e_j, une action de v_i, appartenant au composant C_j, dans le cas où l'adaptation concerne ces deux actions.

Dans notre cas, le contrat d'adaptation C est construit progressivement : à chaque fois qu'un nouveau bloc B_i est intégré pour construire le bloc B, l'architecte doit, d'une part, spécifier les correspondances entre les services de B_i et les services de la spécification de B, et d'autre part, spécifier les correspondances entre les services de B_i et les services des blocs déjà choisis ($\{B_j\}_{j<i}$). Ces correspondances représentent le contrat C ={ v_i } $_{i=1..m}$. Chaque élément v_i du contrat d'adaptation C prend la forme d'un vecteur synchrone :

$\langle\ a_1, a_2, \ldots, a_n, s\ \rangle$, tel que :

$s \in PS_B \cup RS_B \cup \{\varepsilon\}\ et\ a_i \in PS_{B_i} \cup RS_{B_i} \cup \{\varepsilon\}$.

Chaque vecteur contient deux éléments a_i et a_j qui sont différents de ε, cela signifie que le service a_i du bloc B_i correspond au service a_j du bloc B_j. Le contrat d'adaptation C est l'union des éléments de deux sous-contrats : $C = C_{subBlocks} \cup C_{spec}$, tel que :

— $C_{subBlocks}$: spécifie les correspondances entre les sous-blocs $\{B_i\}$,
$C_{subBlocks}$=$\{\langle a_1, a_2, \ldots, a_n, s\ \rangle\}$, tel que $s = \varepsilon\ et\ a_i \in PS_{B_i} \cup RS_{B_i} \cup \{\varepsilon\}$

— C_{spec} : spécifie les correspondances entre les sous-blocs $\{B_i\}$ et la spécification du bloc parent,
C_{spec}=$\{\langle a_1, a_2, ..., a_n, s \rangle\}$, tel que $s \in PS_B \cup RS_B$ *et* $s \neq \varepsilon$ *et* $a_i \in PS_{B_i} \cup RS_{B_i} \cup \{\varepsilon\}$.

ILLUSTRATION

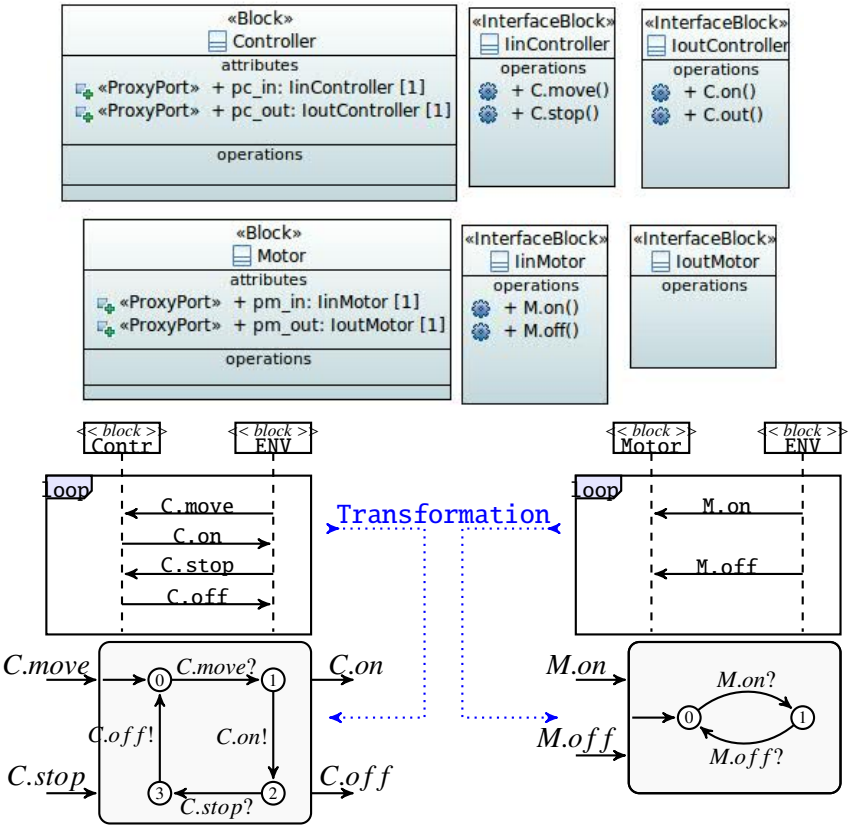


FIGURE 3.4 – Les blocs *Controller* et *Motor* et leurs protocoles

Pour construire la partie du système concernant le robot, nous proposons de réutiliser les composants *Motor* et *Controller* (voir la figure 3.4). Et nous définissons le contrat d'adaptation $C_{Contr \leftrightarrow Mot}$, pour spécifier les correspondances entre les actions inadéquates entre les sous-blocs, *Motor* et *Controller*, et entre ces sous-blocs et le bloc composite *Robot*.
$C_{Contr \leftrightarrow Mot}$ =

$\{\langle C.on, M.on, \varepsilon\rangle, \langle C.off, M.off, \varepsilon\rangle, \langle C.move, \varepsilon, R.move \rangle, \langle C.stop, \varepsilon, R.stop \rangle\}$

Par exemple, la première règle de ce contrat indique que l'action $C.on$ dans le bloc *Controller* correspond à l'action $M.on$ dans le bloc *Motor*. Et la troisième règle indique que l'action $C.move$ dans *Controller* correspond à l'action $R.move$ du bloc composite *Robot*.

3.4.3/ VÉRIFICATION DE LA VALIDITÉ DU CONTRAT D'ADAPTATION

Un contrat d'adaptation C doit respecter plusieurs conditions de validité afin de le prendre en considération pour la génération de l'adaptateur. Ses conditions sont à vérifier sur ses

sous-contrats.

Comme dans notre approche, nous traitons uniquement les correspondances de type un-à-un, notre sous-contrat $C_{subBlocks}$ doit vérifier la condition 1 et le sous-contrat C_{spec} doit vérifier la condition 2.

Condition 1 (validité de $C_{subBlocks}$) :

Condition 1.1 : un service requis d'un bloc correspond au plus à un service fourni d'un autre bloc.
$\forall\ v_i = \langle\ e_{i1}, ..., e_{in}, \varepsilon\ \rangle \in C_{subBlocks}, e_{ik} = a, a \in RS_{B_k}$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, ..., e_{jn}, \varepsilon\ \rangle \in C_{subBlocks}, e_{jk} \neq a$

Condition 1.2 : un service fourni d'un bloc correspond au plus à un service requis d'un autre bloc.
$\forall\ v_i = \langle\ e_{i1}, ..., e_{in}, \varepsilon\ \rangle \in C_{subBlocks}$, si $e_{ik} = a, a \in PS_{B_k}$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, ..., e_{jn}, \varepsilon\ \rangle \in C_{subBlocks}, e_{jk} \neq a.$

Le sous contrat C_{spec} doit satisfaire la condition 2 :

Condition 2 (validité de C_{spec}) :

Condition 2.1 : un service fourni a de la spécification peut correspondre au plus à un service fourni b des sous-blocs.
$\forall\ v_i = \langle\ e_{i1}, ..., e_{in}, a\rangle \in C_{spec}, a \in PS_B \wedge e_{ik} = b$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, ..., e_{jn}, c\rangle \in C_{spec}\ , c \neq a\ et\ e_{jk} \neq b\ .$

Condition 2.2 : un service requis a de la spécification peut correspondre au plus à un service requis b des sous-blocs.
$\forall\ v_i = \langle\ e_{i1}, ..., e_{in}, a\rangle \in C_{spec}, a \in RS_B \wedge e_{ik} = b$
$\Rightarrow \forall\ v_{j \neq i} = \langle\ e_{j1}, ..., e_{jn}, c\rangle \in C_{spec}\ , c \neq a\ et\ e_{jk} \neq b\ .$

Les blocs réutilisés et le bloc composite parent doivent également vérifier les conditions de cohérence.

Condition 3 (Vérification de la cohérence des sous-blocs sélectionnés et du bloc parent) : cette condition doit être vérifiée par le bloc parent B et les blocs réutilisés $\{B_i\ \}$ qui vont composer B.

- Un service fourni d'un sous-bloc ne peut pas être un service requis du bloc parent : $\forall\ a \in PS_{B_i}$, nous avons $a \notin RS_B$.
- Un service requis d'un sous-bloc ne peut pas être un service fourni du bloc parent : $\forall\ a \in RS_{B_i}$, nous avons $a \notin PS_B$.

3.4.4/ GÉNÉRATION DE L'ADAPTATEUR

Dans cette section, nous présentons notre démarche pour générer automatiquement l'adaptateur en fonction des spécifications des blocs B_i et du bloc composite B. Nous présentons, en premier, l'approche permettant de générer le protocole d'interaction de l'adaptateur, ensuite sont présentés les algorithmes pour générer sa spécification SysML et celle du bloc composite.

Pour déduire le protocole de l'adaptateur, nous avons besoin des automates d'interface des blocs B et B_i. Et nous avons également besoin d'exploiter la relation de raffinement des automates d'interface définie dans le chapitre 2 (voir la définition 2.15).

Donc, pour déduire le protocole de l'adaptateur, que nous spécifions par l'automate d'interface A_{ad}, nous avons besoin des automates d'interface de l'ensemble des blocs réutilisables et du bloc parent. Ces automates son obtenus à partir des diagrammes de séquence des blocs concernés. Ainsi, nous notons par A_B l'automate d'interface du bloc B et par $\{A_i\}_{i=1..n}$, les automates des blocs B_i. Et pour calculer A_{ad}, nous proposons d'exploiter la relation de raffinement entre les automates d'interface exprimée par la formule suivante :

$$(1)\ A_B \geq (A_G \parallel_c A_{ad})$$

Explication : si on considère que A_G est l'automate d'interface correspondant à la composition de tous les blocs dans l'ensemble $\{B_i\}$, donc l'automate d'interface obtenu avec le composition de A_G et A_{ad} doit raffiner l'automate d'interface du bloc composite A_B. Ce qui se traduit au niveau des composants par le fait que le composant qui sera obtenu de l'assemblage des blocs B_i avec l'adaptateur, doit fournir au moins les même services que ceux spécifiés dans le bloc parent B et requérir au plus les même services que ceux requis par B (ce qui est garantit par la relation de raffinement).

Dans ce qui suit, nous présentons la relation de composition des blocs réutilisés ($\parallel_c$) et ensuite montrons comment déduire l'automate A_{ad} de la formule (1).

Pour calculer la composition des automates d'interface des blocs réutilisés, nous devons adapter les opérations de produit synchrone et de composition entre les automates d'interface, définies dans [dAH01], afin de prendre en compte le contrat d'adaptation et les actions correspondantes au lieu des actions partagées. Pour cela, nous devons définir préalablement $Corresponding(A_i, A_j)$, l'ensemble des actions correspondantes entre deux automates d'interface A_i et A_j, et la fonction $corresp(a)$ qui retourne l'action qui correspond à l'action *a* dans le contrat d'adaptation :

Corresponding(A_i, A_j)**=** $\{a \in \Sigma^I_{A_i} \cup \Sigma^O_{A_i} \cup \Sigma^I_{A_j} \cup \Sigma^O_{A_j} \mid \exists v = \langle\ e_1, \ldots, e_n, \varepsilon \rangle \in\ C_{subBlocks}\ \wedge\ e_k = a\}$
corresp(a)$=\{a' \mid \exists\ V \in\ et\ \exists (i, j) \in \mathbb{N} \times \mathbb{N}\ et\ v = \langle\ a_1,...,a_n\rangle\ et\ a_i = a\ et\ a_j = a'\}$

Définition 3.1 (Produit synchrone sous contrat d'adaptation)**.** *Soient A_i et A_j deux automates d'interface composables, reliés par un contrat d'adaptation* C*, valide (satisfait la condition 1). Nous définissons le produit synchrone de A_i et A_j, sous un contrat d'adaptation, par le tuple :*

$$A_i \otimes_c A_j = \langle\ S_{A_i \otimes_c A_j}, \iota_{A_i \otimes_c A_j}, \Sigma^I_{A_i \otimes_c A_j}, \Sigma^O_{A_i \otimes_c A_j}, \Sigma^H_{A_i \otimes_c A_j}, \delta_{A_i \otimes_c A_j} \rangle$$

— $S_{A_i \otimes A_j} = S_{A_i} \times S_{A_j}$ *et* $\iota_{A_i \otimes_c A_j} = (\iota_{A_i}, \iota_{A_j})$ *;*
— $\Sigma^I_{A_i \otimes_c A_j} = (\Sigma^I_{A_i} \cup \Sigma^I_{A_j}) \setminus$ *Corresponding*(A_i, A_j) *;*
— $\Sigma^O_{A_i \otimes_c A_j} = (\Sigma^O_{A_i} \cup \Sigma^O_{A_j}) \setminus$ *Corresponding*(A_i, A_j) *;*
— $\Sigma^H_{A_i \otimes_c A_j} = \Sigma^H_{A_i} \cup \Sigma^H_{A_j} \cup$ *Corresponding*(A_i, A_j) *;*
— $((s_i, s_j), a, (s'_i, s'_j)) \in \delta_{A_i \otimes_c A_j}$ *si :*
  — $a \notin$ *Corresponding*$(A_i, A_j) \wedge (s_i, a, s'_i) \in \delta_{A_i} \wedge s_j = s'_j$
  — $a \notin$ *Corresponding*$(A_i, A_j) \wedge (s_j, a, s'_j) \in \delta_{A_j} \wedge s_i = s'_i$
  — $a \in$ *Corresponding*$(A_i, A_j) \wedge a \in \Sigma^O_{A_i} \wedge (s_i, a, s'_i) \in \delta_{A_i} \wedge (s_j, corresp(a), s'_j) \in \delta_{A_j}$

— $a \in Corresponding(A_i,A_j) \wedge a \in \Sigma^O_{A_j} \wedge (s_j,a,s'_j) \in \delta_{A_j} \wedge (s_i,corresp(a),s'_i) \in \delta_{A_i}$

Ce produit absorbe les transitions $(s_i, s_j) \xrightarrow{a!} (s'_i, s_j) \xrightarrow{corresp(a)} (s'_i, s'_j)$ et les transitions $(s_i, s_j) \xrightarrow{a!} (s_i, s'_j) \xrightarrow{corresp(a)} (s'_i, s'_j)$ en les remplaçant par une transition étiquetée par une action interne, $(s_i, s_j) \xrightarrow{a;} (s'_i, s'_j)$.

Définition 3.2 (Composition sous contrat d'adaptation)**.** *La composition sous contrat d'adaptation, C, de deux automates d'interface A_i et A_j est définie par : $A_i \|_c A_j = A_i \otimes_c A_j$ après la suppression des états bloquants et de tous les états atteignables depuis ces derniers en activant des actions de sortie ou internes. L'ensemble des états bloquants est défini comme suit :*

$$Dead(A_i,A_j)= \left\{ \begin{array}{c} (s_i, s_j) \in S_{A_i} \times S_{A_j} \mid \exists a \in Corresponding(A_i, A_j) \\ \left(\begin{array}{c} a \in \Sigma^O_{A_i}(s_i) \wedge corresp(a) \notin \Sigma^I_{A_j}(s_j) \\ \vee \\ a \in \Sigma^O_{A_j}(s_j) \wedge corresp(a) \notin \Sigma^I_{A_i}(s_i) \end{array} \right) \end{array} \right\}$$

Ainsi, le protocole d'interaction global A_G des sous-blocs $\{B_i\}_{i=1..n}$ est obtenu en composant leurs automates d'interface $\{A_i\}_{i=1..n}$, en utilisant la composition sous contrat d'adaptation : $A_G = A_1 \|_c A_2 \|_c ... \|_c A_n$. A chaque étape *i* du calcul de la composition, nous calculons la composition entre l'automate d'interface A_c (où $A_c = A_1 \|_c ... \|_c A_{i-1}$) et l'automate d'interface A_i du bloc B_i. Et à chaque étape *i*, il faut vérifier la condition 4 :

Condition 4 : *(les blocs doivent être compatibles)* A_c ne doit pas être vide.

Cette condition est inspirée du théorème 3.1 présenté dans [BR08]. C'est une condition indispensable pour pouvoir déduire l'automate A_{ad}. En effet pour déduire le protocole d'interaction de l'adaptateur en utilisant l'automate d'interface A_G et la relation de raffinement, nous proposons d'exploiter la formule (1), décrite précédemment, et d'utiliser les théorèmes 3.1 et 3.2 :

Théorème 3.1 ([BR08])**.** *Une solution R existe pour $Q \geq (P \parallel R)$, ssi $mirror(P)$ et Q sont compatibles.*

$mirror(P)$ est l'automate d'interface P où les actions d'entrée et de sortie sont inversées : l'action d'entrée devient de sortie et inversement. Nous définissons formellement la fonction $mirror$ comme suit :

$mirror(Q)$={ $Q' \mid S_{Q'} = S_Q$, pour toute transition $(s, a!, s') \in \delta_Q$, il existe $(s, a?, s') \in \delta_{Q'}$, et pour toute transition $(s, a?, s') \in \delta_Q$, il existe $(s, a!, s') \in \delta_{Q'}$, et pour toute transition $(s, a;, s')$ $\in \delta_Q$, il existe $(s, a;, s') \in \delta_{Q'}$}

D'après le théorème 3.1 et en faisant le parallèle avec notre formule (1), l'automate A_{Ad} existe ssi mirror(A_B) et A_G sont compatibles. Et cette compatibilité dépend également de la compatibilité des automates composant A_G, d'où la condition 4.

Ainsi, pour déduire A_{ad}, nous nous référons à la formule proposée dans le deuxième théorème présenté dans [BR08] :

Théorème 3.2 ([BR08])**.** *Lorsque la condition énoncée dans le théorème3.1 est remplie, la solution la plus générale à $Q \geq (P \parallel R)$ existe et est donnée par $R = mirror(P \parallel mirror(Q))$.*

Ainsi, dans notre cas, comme nous avons des actions correspondantes entre les automates d'interface au lieu des actions partagées, l'automate A_{ad} doit être calculé comme suit :

$$A_{ad} = mirror(A_G \|_c mirror(A_B)) = mirror(A_1 \|_c ... \|_c A_n \|_c mirror(A_B))$$

D'après les théorèmes 3.1 et 3.2, nous proposons la condition 5 :

Condition 5 : A_G et $mirror(A_B)$ doivent être compatibles.

Algorithme 2 : Déduction du protocole d'interaction de l'adaptateur

1 **ENTRÉES :** $A_{ad} = \langle A_{ad}, \iota_{ad}, \Sigma^I_{ad}, \Sigma^O_{ad}, \Sigma^H_{ad}, \delta_{ad} \rangle, C$

2 **SORTIES :** $A_{adapter} = \langle S_{adapter}, \iota_{adapter}, \Sigma^I_{adapter}, \Sigma^O_{adapter}, \Sigma^H_{adapter}, \delta_{adapter} \rangle$

- Créer une copie $A_{adapter}$ de A_{ad}.
- Définir l'ensemble T de toutes les transitions ($s \xrightarrow{a;} s' \in \delta_{adapter}$), telle que ***a*** appartient à une règle dans le contrat C.
- Remplacer toutes les transitions $s \xrightarrow{a;} s' \in \delta_{adapter}$ telle que $s \xrightarrow{a;} s' \in T$, par $s \xrightarrow{a?} s'' \xrightarrow{corresp(a)!} s'$.

La condition 5 est nécessaire pour garantir l'existence d'un adaptateur (pour avoir A_{ad} non vide). Donc, si la condition 5 est vérifiée, nous pouvons déduire le protocole d'interaction réel de l'adaptateur en appliquant l'algorithme 2. En effet, selon le produit synchrone sous contrat d'adaptation, les transitions étiquetées avec des actions internes *a ;* dans A_{ad}, qui apparaissent dans le contrat, représentent les transitions dans lesquelles l'adaptateur joue le rôle de convertisseur (*Converter*). Ainsi, pour déduire le protocole réel de l'adaptateur, chaque transition de cet ensemble doit être remplacée par deux transitions : la première est étiquetée avec l'action d'entrée *a ?* et le seconde avec l'action correspondante *corresp(a) !*. Cela signifie que lorsque l'adaptateur reçoit le message *a ?* d'un bloc, il le convertit en entrée appropriée d'un autre bloc et le transmet à l'aide d'une action de sortie *corresp(a) !*. Les transitions qui ne sont pas sélectionnées par l'algorithme 2 sont celles où l'adaptateur joue le rôle de complément pour répondre à la spécification du bloc parent. Donc, pour déduire le comportement réel de l'adaptateur, cet algorithme accepte comme entrée l'automate A_{ad} et calcule l'automate d'interface de l'adaptateur $A_{adapter}$, en déduisant les transitions absorbées par le produit synchrone sous un contrat d'adaptation.

ILLUSTRATION DE LA GÉNÉRATION DU PROTOCOLE DE L'ADAPTATEUR

Pour illustrer notre approche, nous proposons de générer l'adapteurs des blocs *Controller* et *Motor* en se basant sur leur contrat d'adaptation $C_{Contr \leftrightarrow Mot}$. Nous adaptons ces deux blocs pour répondre à la spécification du bloc parent *Robot* en introduisant l'adaptateur $ADContrMot$ (voir la figure 3.5), où le protocole d'interaction de l'adaptateur $ADContrMot$ est représenté à l'aide d'un automate d'interface. Pour calculer ce dernier nous appliquons la formule (1), présentée précédemment, sur l'étude de cas et nous obtenons :

$$A_{Robot} \geq A_{Controller} \|_c A_{Motor} \|_c A_{ADContrMot}$$

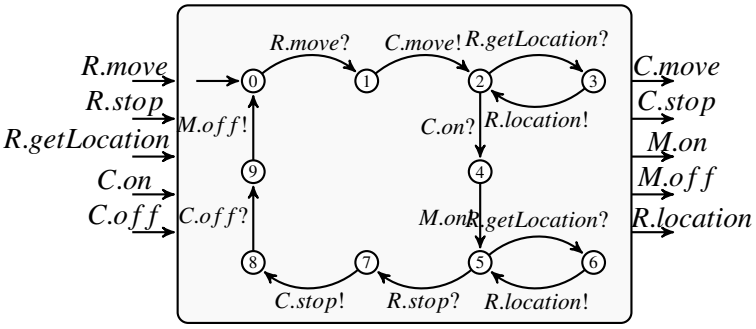


FIGURE 3.5 – Le protocole du bloc adaptateur $ADContrMot$

Et en se basant sur les théorème 3.1 et 3.2, nous déduisons :

$A_{ADContrMot} = mirror(A_{Controller} \parallel_c A_{Motor} \parallel_c mirror(A_{Robot}))$

Pour déduire l'automate d'interface de l'adaptateur, nous appliquons l'algorithme 2 sur l'automate $A_{ADContrMot}$ pour obtenir l'automate décrit dans la figure 3.5.

3.4.4.1/ GÉNÉRATION DE L'ARCHITECTURE DU BLOC ADAPTATEUR

Dans cette section, nous présentons notre démarche pour générer automatiquement la spécification SysML du bloc adaptateur, ainsi que l'architecture du bloc composite B, après l'étape du calcul du protocole d'interaction (l'automate d'interface) du bloc adaptateur.

Pour déduire la spécification SysML du bloc d'adaptateur $B_{adapter}$, nous proposons l'algorithme 3 permettant de déduire l'ensemble des ports de $B_{adapter}$ en analysant l'automate d'interface du bloc adaptateur. Cet algorithme parcourt l'ensemble des actions de l'automate d'interface de l'adaptateur pour générer les opérations internes, requises, et fournies du bloc adaptateur. Il permet aussi de générer ses blocs d'interface. Pour construire le BDD et l'IBD de la partie B, nous appliquons l'algorithme 4 et l'algorithme 5. Le rôle de l'algorithme 4 est d'établir les relations de composition entre le bloc parent B, ses sous-blocs $\{B_i\}$, et le bloc adaptateur $B_{adapter}$. Quant à l'algorithme 5, nous l'utilisons pour générer l'IBD du bloc B. Sa tâche est de lier les ports de l'adaptateur aux ports des sous-blocs $\{B_i\}$ et du bloc parent B.

3.5/ CONCLUSION ET PERSPECTIVES

Dans ce chapitre, nous avons présenté une approche ascendante pour construire un SBC décrit par ses spécifications partielles (composants composites). Elle est fondée sur la réutilisation et l'adaptation de blocs SysML à l'aide de blocs adaptateurs. Pour cela, nous avons proposé d'exploiter le formalisme des automates d'interface pour la spécification formelle des protocoles d'interaction des blocs et la dérivation du protocole d'interaction de l'adaptateur. Ce dernier se distingue, par rapport aux approches existantes, par ses deux rôles. D'une part, il joue son rôle de convertisseur entre les blocs réutilisés, et entre les blocs réutilisés et leur environnement. D'autre part, il joue le deuxième rôle en tant que complément en implémentant les services manquants offerts ou requis par

Algorithme 3 : Définition du bloc SysML de l'adaptateur

1 **ENTRÉES :** automate d'interface $A_{adapter}$=⟨$S_{Adapter}$,$I_{adapter}$,$\Sigma^{I}_{adapter}$,$\Sigma^{O}_{adapter}$,$\Sigma^{H}_{adapter}$,$\delta_{adapter}$⟩

2 **SORTIES :** bloc $B_{adapter}$ =⟨ $name_{adapter}$, $Op_{adapter}$, $P_{inadapter}$, $P_{outadapter}$, $TypePort_{adapter}$ ⟩,

3 bloc d'interface $Ioutadapter$ =⟨ $name_{outadapter}$, $Op_{outadapter}$ ⟩,

4 bloc d'interface $Iinadapter$ =⟨ $name_{inadapter}$, $Op_{inadapter}$ ⟩.

// DÉFINITION DES BLOCS D'INTERFACE

$name_{outadapter} = Ioutadapter$; $name_{inadapter} = Iinadapter$;

Pour tout $a_i \in \Sigma^{I}_{adapter} \cup \Sigma^{O}_{adapter}$ **Faire**

  Si $a_i \in \Sigma^{I}_{adapter}$ **Alors**

    $Op_{inadapter} = Op_{inadapter} \cup \{a_i\}$

  Sinon

    -$Op_{inadapter} = Op_{outadapter} \cup \{a_i\}$

// DÉFINITION DU BLOC ADAPTATEUR $B_{adapter}$

$name_{adapter} = adapter$;$Op_{adapter} = \emptyset$; $P_{inadapter} = adapterp_in$; $P_{outadapter} = adapterp_out$;

$TypePort_{adapter} = \{(adapterp_in, Iinadapter), (adapterp_out, Ioutadapter)\}$;

Algorithme 4 : Construction du BDD du bloc parent B

1 **ENTRÉES :** B, $\{B_i\}$, $B_{adapter}$

2 **SORTIES :** BDD_B =⟨ IB_B, SB_B, $SubB_B$ ⟩

$IB_B = B$;

$SB_B = \{B_i\}_{i=1..n} \cup \{B, B_{adapter}\}$

$SubB_B = \{(B, \{B_1, ...B_i, ..B_n, B_{adapter}\})\}$

Algorithme 5 : Construction de l'IBD du bloc parent B

1 **ENTRÉES :** B, $\{B_i\}$, $B_{adapter}$

2 **SORTIES :** IBD_B =⟨ $Parts_B$, $BlockPart$, $Ports_B$, $Pextin_B$,$Pextout_B$, $PinPart$, $PoutPart$, $Connectors_{inB}$, $Connectors_{outB}$ ⟩

//CRÉER LA part 'ad' DU BLOC $B_{adapter}$.

-$Parts_B = Parts_B \cup \{ad\}$;

-$BlockPart = BlockPart \cup \{(ad, B_{adapter})\}$

-$PinPart = PinPart \cup \{(ad, B_{adapter}.P_{in})\}$;

-$PoutPart = PoutPart \cup \{(ad, B_{adapter}.P_{out})\}$;

-$Pextin_B = B.P_{in}$; $Pextout_B = B.P_{out}$;

-$Ports_B = Ports_B \cup \{B_{adapter}.P_{in}, B_{adapter}.P_{out}, B.P_{in}, B.P_{out}\}$;

-$Connectors_{outB} = \{(Pextin_B, PinPart(ad)), (Pexout_B, PoutPart(ad))\}$;

//CRÉER LES parts b_i POUR LES BLOCS B_i.

Pour tout $B_i \in \{B_i\}$ **faire**

  -$Parts_B = Parts_B \cup \{b_i\}$;

  -créer la part b_i pour le bloc B_i

  -$BlockPart = BlockPart \cup \{(b_i, B_i)\}$

  -$PinPart = PinPart \cup \{(b_i, B_i.P_{in})\}$;

  -$PoutPart = PoutPart \cup \{(b_i, B_i.P_{out})\}$;

  -$Ports_B = Ports_B \cup \{B_i.P_{in}, B_i.P_{out}\}$;

  -$Connectors_{inB} = Connectors_{inB} \cup$

  $\{(PinPart(b_i), PoutPart(ad)), (PoutPart(b_i), PinPart(ad))\}$;

Fin pour

l'environnement ou les blocs réutilisés.

Comme perspective au travail présenté, nous projetons de proposer une démarche d'adaptation des SBC modélisés avec SysML, en considérant plusieurs types d'inadéquation entre les services des composants. En effet, dans ce travail, nous considérons uniquement des inadéquations entre les services des composants de type "un pour un", autrement dit un service dans un composant correspond à un autre service dans un autre composant. Il serait intéressant, d'améliorer notre un approche pour considérer des inadéquations de type, "un-pour-plusieurs", "plusieurs-pour-un", et "plusieurs-pour-plusieurs".

3.6/ BILAN

Les résultats de ce chapitre concernent les éléments suivants :

1. **Projet** :
   - Le projet Tassili SYSVAP, coopération avec l'université USTHB d'Alger (Algérie), 2013-2016.
   - Le projet région SyVAD (Modélisation SysML pour la Validation et la Vérification de micro-systèmes), période 2012-2014.
2. **Encadrement de thèse de doctorat** :
   - La thèse de Hamida Bouaziz [Bou16], 2016.
3. **Publications** : [BCHM16b, BCZ15].

4

SPÉCIFICATION INCRÉMENTALE DE L'ARCHITECTURE D'UN SBC EN EXPLOITANT LES EXIGENCES

4.1/ INTRODUCTION

Généralement les SBC sont construits en assemblant divers composants réutilisables. Cette façon de faire permet de construire des systèmes de plus en plus complexes, avec un moindre coût. Ce qui fait le succès des approches de développement par composants. Néanmoins, concevoir rigoureusement de tels systèmes reste une tâche complexe. En effet, l'une des principales difficultés concerne la prise en compte des exigences du système lors de la conception du SBC avec un ensemble de composants réutilisables. D'où la question suivante : comment spécifier l'architecture d'un SBC répondant à toutes les exigences définies ?

Pour répondre à la problématique posée, nous présentons une approche méthodologique permettant de construire un SBC, directement à partir d'un diagramme des exigences SysML et d'assurer formellement la cohérence de l'architecture du système vis-à-vis des exigences définies. Dans ce travail, nous nous sommes focalisés sur les exigences fonctionnelles d'un SBC, exprimant chacune des contraintes sur le comportement d'un composant. Elles sont spécifiées avec le diagramme des exigences SysML. Nous les exploitons pour construire incrémentalement l'architecture du système global, que nous spécifions avec les diagrammes SysML BDD et IBD. Nous garantissons ainsi, la cohérence entre l'architecture du système obtenu et les exigences, en les formalisant avec la logique temporelle linéaire (LTL) et les vérifiant avec le Model-Cheker SPIN, étape par étape, sur les comportements des composants modélisés avec des diagrammes de séquence. Dans ce travail, nous avons également défini les conditions concernant la préservation des exigences par la composition.

Dans ce qui suit nous présentons un aperçu de l'approche dans la section 4.2, puis nous situons nos contributions par rapport aux travaux existants dans la section 4.3. Pour les illustrations, nous présentons une étude de cas dans la section 4.4. Les principales étapes de notre approche sont présentées dans les sections 4.5-4.8. L'application de ces étapes sur l'étude de cas est présentée dans la section 4.9. Nous terminons ce chapitre par une conclusion et un bilan dans les deux dernières sections.

4.2/ APERÇU DE L'APPROCHE

La contribution de ce travail est une approche pour construire un SBC et spécifier son architecture, en respectant un ensemble des exigences fonctionnelles spécifiées avec le diagramme des exigences SyML. Pour définir cette architecture, le concepteur du système exploite une bibliothèque de composants réutilisables. Ces composants sont considérés comme des boîtes noires, décrits uniquement par leurs interfaces, spécifiées avec des diagrammes de séquence. Nous proposons donc, de spécifier les exigences d'un SBC avec le diagramme des exigences SysML, puis d'analyser ce diagramme afin d'associer une à une ses exigences atomiques à des composants logiciels qui les satisfont. Cette association (exigence/composant) est réalisée après avoir effectué une étape de vérification formelle avec le Model-Checker SPIN [Hol97]. Chaque composant vérifié est ensuite connecté, après avoir vérifié sa compatibilité, aux autres composants du système (qui constituent l'architecture partielle du système), déjà vérifiés vis-à-vis des exigences.

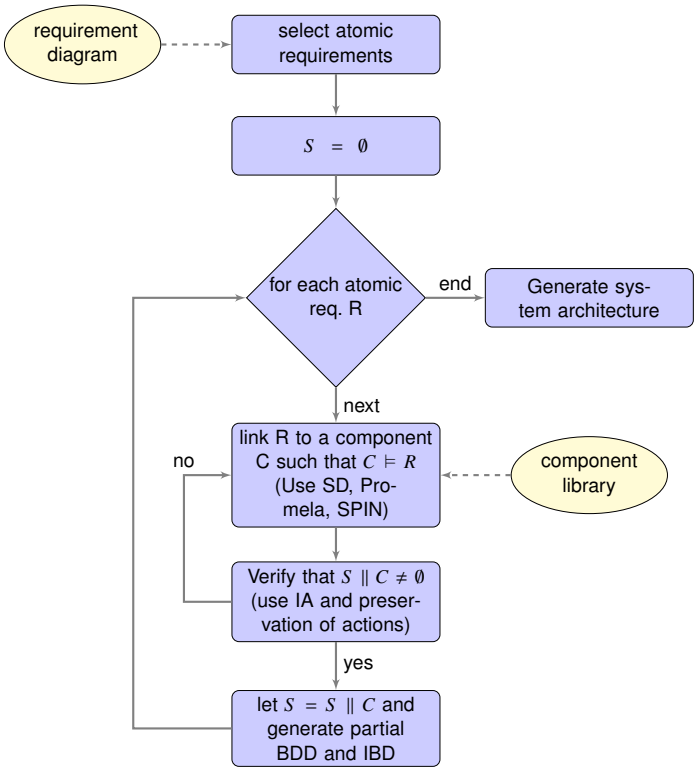


FIGURE 4.1 – Illustration des étapes de l'approche proposée

Les principales étapes de notre approche, présentées dans la figure 4.1, sont décrites ci-dessous :

1. Analyser le diagramme des exigences SysML pour obtenir les exigences atomiques car elles sont plus précises et il est plus facile de trouver les composants qui les satisfont (voir la section 4.5).
2. Soit R_0 la première exigence atomique, soit C_0 un composant de la bibliothèque de composants, décrit par le diagramme de séquence DS_0. Spécifier R_0 avec la formule LTL F_0 et traduire DS_0 en code Promela PRO_0, puis vérifier que C_0 satisfait

R_0 en vérifiant que PRO_0 satisfait F_0 avec le Model-Checker SPIN (voir la section 4.6). Si cette étape retourne *False*, alors C_0 ne satisfait pas R_0, il faut donc obtenir le composant approprié dans d'autres bibliothèques ou il faut le développer à partir de zéro.

3. Soit A_0 l'automate d'interface décrivant le protocole du composant, obtenu à partir du diagramme de séquence DS_0 (voir la section 4.7).
4. Répéter jusqu'à l'analyse de toutes les exigences atomiques.
   a. Soit R_i la prochaine exigence atomique, *connectée* à R_{i-1}, traitée précédemment (voir la définition 4.3). Soit C_i le composant qui satisfait R_i, grâce à la formule LTL F_i et au code Promela PRO_i. Et A_i l'automate d'interface décrivant le protocole du composant.
   b. Vérifier que C_i et C_{i-1} sont compatibles grâce à leurs automates d'interface, donc vérifier que $A_i \parallel A_{i-1} \neq \emptyset$.
   c. Vérifier que les exigences R_i et R_{i-1} sont préservées par la composition. Elles sont donc satisfaites par le composite $C = C_i \parallel C_{i-1}$ (voir la section 4.7).
   d. Définir l'architecture partielle du système, conforme aux exigences analysées, par le composite $C = C_i \parallel C_{i-1}$, conformément à la définition 2.7. Cette architecture est spécifiée avec les diagrammes SysML BDD et IBD.
5. Fin de la répétition

L'architecture finale du système à construire est validée à la fin du traitement de toutes les exigences atomiques.

4.3/ ÉTAT DE L'ART ET CONTRIBUTIONS

Dans le contexte de la vérification des exigences SysML liées aux SBC, nous citons [PEML10], où les auteurs s'intéressent à la vérification des exigences SysML sur des systèmes complexes, développés avec un ensemble de composants hétérogènes. Contrairement à notre approche, les exigences ne sont pas exploitées pour définir l'architecture globale du système. En effet, Ils considèrent que l'architecture du système est donnée et validée, et ils proposent uniquement une solution pour vérifier les exigences sur le comportement du système modélisé avec un diagramme d'activités. La vérification des exigences est également traitée dans [RHI17], où une approche de vérification des exigences SysML est proposée. Les exigences sont formalisées avec la logique temporelle et le comportement du système avec les diagrammes d'activités, qui est ensuite transformé en modèle spécifié avec les réseaux de Petri. Et enfin un algorithme de vérification modulaire est proposé. L'approche ne prend pas en considération les problématiques liées aux SBC, comme la compatibilité des composants, et donc ne traite pas la question de l'architecture du système. Dans [DOP17], une approche permettant de vérifier les exigences sur des SBC temporisés modélisés avec SysML, étendu, pour spécifier les composants avec des contrats. Les principales différences avec notre proposition concernent, d'une part le formalisme utilisé pour spécifier le comportement des composants (les automates temporisé dans cette approche et les automates d'interface dans notre cas), et d'autre part l'exploitation du diagramme des exigences SysML qui n'est pas considérée dans cette approche.

Quelques travaux utilisent la méthode KAOS [vL09] pour intégrer les exigences dans le processus de développement de systèmes. Par exemple, dans [vL03], une approche est proposée pour dériver l'architecture du système en partant des buts définis avec la méthode KAOS. Le processus de dérivation proposé est incrémental, commençant par la dérivation d'une spécification abstraite de l'architecture, qui est ensuite raffinée jusqu'à la satisfaction de l'ensemble des buts modélisés. Les auteurs dans [BDL05], proposent une approche incrémentale en ajoutant des propriétés structurelles et comportementales à une architecture logicielle. Une autre approche basée sur KAOS et SysML est proposée dans [LSM+10, FLB+19]. Dans [LSM+10], les auteurs proposent d'étendre le diagramme des exigences Sysml avec les concepts de la méthode KAOS et proposent ensuite une sémantique formelle des exigences avec la méthode B. Et dans [FLB+19], ils utilisent la méthode combinant KAOS et SysML pour modéliser et vérifier les exigences d'un protocole de communication dans le domaine du ferroviaire. Les questions liées à l'architecture des SBC ne sont pas considérées dans ces derniers travaux.

Dans [CLR+09, KLLS12], les auteurs proposent une approche dirigée par les modèles, appelée rCOS, pour construire des SBC fiables, en partant des exigences, extraites d'un diagramme des cas d'utilisation UML. Les étapes de vérification et de validation sont intégrées dans le processus de développement proposé. Notre approche est similaire à la leur, dans l'utilisation des diagrammes de séquence pour exprimer les interactions des composants, cependant, dans notre cas, nous nous distinguons par l'exploitation du diagramme des exigences SysML pour modéliser et exploiter les exigences pour définir l'architecture du système.

Les travaux proposés dans [Dro03, Dro07] proposent une approche basée sur les arbres de comportements. Ils traduisent les exigences atomiques en arbres de comportements, qui sont des graphes spécifiant les interactions des composants. Chaque exigence est traduite en arbre de comportements, l'ensemble des arbres obtenus sont ensuite fusionnés pour obtenir l'arbre de comportement du système global. Ce dernier est ensuite exploité pour extraire la structure du système. Contrairement à notre approche, celle-ci n'exploite pas les composants existants pour définir l'architecture du système global. Une approche intéressante, proche des derniers travaux cités, qui a inspiré également notre travail a été proposée dans [LNRT10]. Les auteurs proposent une approche pour construire, d'une manière incrémentale, l'architecture d'un SBC à partir des exigences brutes décrites dans un langage naturel. Ils proposent de relier incrémentalement chacune des exigences à une architecture partielle du système, grâce au modèle de composants exploité et aux informations extraites à partir des exigences brutes. Contrairement à notre proposition, le langage SysML n'est pas exploité pour spécifier les exigences.

Contributions : Notre approche se distingue par rapport aux travaux existants, principalement, par les points suivants :

— exploitation de SysML et du formalisme des automates d'interface : nous proposons de relier les exigences, spécifiées et organisées avec des diagrammes SysML (contrairement aux travaux exploitants KAOS), directement dans l'architecture système, en exploitant le formalisme des automates d'interface et la composition des interfaces de composants ;

— construction incrémentale des SBC et vérification de la préservation des exigences par la composition : la construction de l'architecture du système est incrémentale et guidée par les exigences, et la préservation de ces exigences dans le système final

est garantie par la compatibilité des automates d'interface.

En somme, nous apportons une solution aux concepteurs de SBC, souhaitant utiliser SysML comme langage de modélisation et exploiter particulièrement son diagramme des exigences, pour construire un système complexe avec des composants réutilisables, répondant à l'ensemble des exigences définies.

4.4/ ÉTUDE DE CAS

Pour illustrer notre approche nous présentons l'étude de cas, simplifiée, d'un système de sécurité d'un véhicule, relatif au déclenchement des airbags et au verrouillage des ceintures de sécurité, lorsque un choc se produit. Ce système est inspiré de l'étude de Barnes et al.[BMFN01] et les exigences de sécurité dictées par le ministère américain des transports pour améliorer la protection des occupants des véhicules [Adm98]. Dans l'étude, les auteurs définissent les conditions nécessaires pour activer automatiquement les dispositifs de sécurité dans une voiture.

La figure 4.2 illustre un système de sécurité composé de plusieurs capteurs situés tout autour d'un véhicule (accéléromètres, capteurs de choc, capteurs de pression, tachymètres, capteurs de pression de freinage, gyroscopes, etc.) pour la détection des collisions. Quand une voiture entre en collision avec un obstacle, ou les freins sont actionnés, il y a une décélération de la vitesse, ce qui permet aux capteurs de détecter les valeurs de décélération et de les envoyer à une unité centrale portant le numéro 1 dans la figure 4.2. En fonction des données reçues, l'unité centrale active les airbags portant le numéro 2 sur la figure et/ou verrouille les ceintures de sécurité.

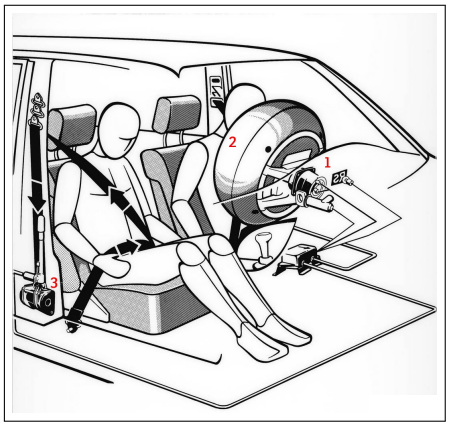


FIGURE 4.2 – Un système de sécurité d'un véhicule [1]

Le diagramme des exigences, correspondant à l'étude de cas, spécifiant les besoins du système est présenté dans la figure 4.3. Dans ce diagramme, l'exigence initiale $R1$ exprime le besoin de préserver la vie des passagers lors des accidents. Elle est décomposée en deux exigences $R1.1$ et $R1.2$ concernant deux dispositifs de sécurité : un système de airbag, qui doit être déployé chaque fois que la voiture rentre en collision et les ceintures de sécurité qui doivent être verrouillées lorsque les capteurs détectent de forts mouvements. Cette dernière exigence est atomique car elle n'est pas décomposable. Par contre, l'exigence $R1.1$ n'est pas atomique, elle est décomposée en

1. Image prise de [Dav10]

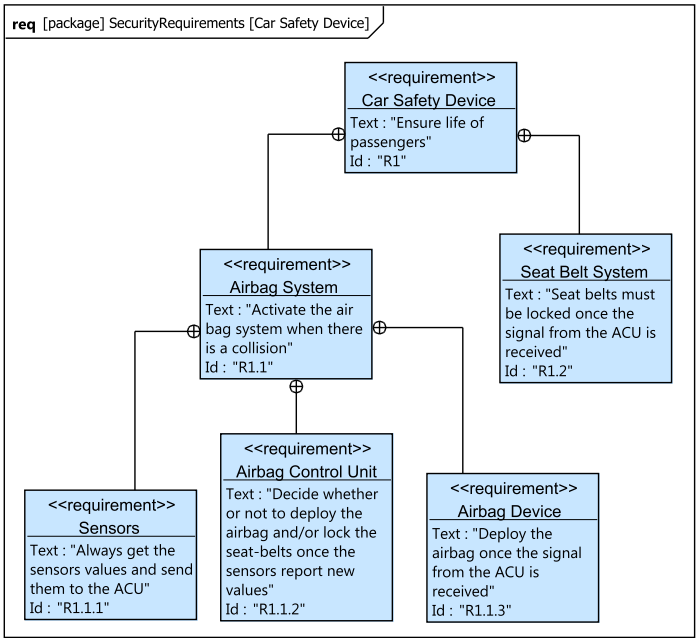


FIGURE 4.3 – Le diagramme des exigences SysML pour le système de sécurité d'un véhicule

exigences atomiques $R1.1.1$, $R1.1.2$ et $R1.1.3$. L'exigence $R1.1.1$ exprime la nécessité de la capture et de l'envoi des valeurs des capteurs à une unité de contrôle de l'airbarg (ACU). L'exigence $R1.1.2$ exprime que le composant ACU décide le déploiement ou non de l'airbag et du verrouillage des ceintures de sécurité, dès que les capteurs envoient de nouvelles données. Enfin, l'exigence $R1.1.3$ exprime la nécessité du déploiement de l'airbag dès la réception du signal de l'ACU.

4.5/ ANALYSE DU DIAGRAMME DES EXIGENCES SYSML

Dans cette section, nous spécifions formellement le diagramme des exigences SysML afin de l'analyser pour extraire formellement les exigences atomiques. Ensuite, nous montrons qu'il est suffisant que ces exigences soient satisfaites par le SBC correspondant, afin que toutes les autres décrites dans le diagramme soient également satisfaites.

Dans la définition suivante, nous considérons deux relations du diagramme SysML.

— ***Containment*** : utilisée pour décomposer une exigence en d'autres plus précises.

— ***Derivation*** : utilisée pour connecter une exigence avec une autre dont elle dérive.

Définition 4.1 (Spécification formelle d'un diagramme des exigences SysML)**.** *Un diagramme des exigences SysML est défini par le tuple* $RD = \langle IR, SR, RelC, RelD \rangle$ *tel que :*

— IR *: l'ensemble des exigences initiales,*

— SR *: l'ensemble de toutes les exigences,*

— $RelC \subseteq SR \times \mathcal{P}(SR)$ *la relation* containment *telle que* $\mathcal{P}(SR)$ *est l'ensemble des sous-ensembles de* SR*,*

— $RelD \subseteq SR \times \mathcal{P}(SR)$ *la relation* derivation.

Par exemple, dans notre étude de cas, la spécification du diagramme des exigences décrit dans la figure 4.3 est $RD = \langle IR, SR, RelC, RelD \rangle$, tel que :

— $IR = \{R1\}$,

— $SR = \{R1, R1.1, R1.2, R1.1.1, R1.1.2, R1.1.3\}$,

— $RelC = \{(R1, \{R1.1, R1.2\}), (R1.1, \{R1.1.1, R1.1.2, R1.1.3\}), (R1.2, \emptyset)\}$,

— $RelD = \emptyset$, pour désigner qu'il n'y a aucune exigence qui dérive d'une autre.

Définition 4.2 (Exigences atomiques)**.** *L'ensemble des exigences atomiques dans le diagramme des exigences spécifié par* $RD = \langle IR, SR, RelC, RelD \rangle$ *est l'ensemble* $AR = \{R | R \in SR, \nexists (R, \{R_i, ...R_n\}) \in RelC\}$

Une exigence atomique est une exigence qui ne peut pas être décomposée. Elle exprime une contrainte sur les actions d'entrée et de sortie liées à un composant(voir la section 4.7).

L'ensemble des exigences atomiques dans notre étude de cas est (voir la figure 4.3) : $\{R1.1.1, R1.1.2, R1.1.3, R1.2\}$.

Remarque : pour calculer l'ensemble des exigences atomiques, il est nécessaire d'analyser l'ensemble SR de toutes les exigences et d'identifier les exigences non liées par la relation $RelC$ (*containment*).

Proposition 4.1 (SBC satisfaisant toutes les exigences atomiques)**.** *Soit* S *un SBC, soit* $RD = \langle IR, SR, RelC, RelD \rangle$ *la spécification du diagramme des exigences et* AR *l'ensemble des exigences atomique de* RD*. Le système* S *satisfait toutes les exigences dans* SR *ssi il satisfait toutes les exigences atomiques dans* AR*.*

Cette proposition (dont la preuve est disponible dans [CCM14a]) indique qu'il suffit que les exigences atomiques soient satisfaites par le SBC pour décider de la satisfaction de toutes les exigences décrites dans le diagramme des exigences.

4.6/ VÉRIFICATION FORMELLE DES EXIGENCES SYSML SUR DES COMPOSANTS

Pour vérifier si un composant satisfait une exigence donnée, nous proposons de traduire les diagrammes de séquence modélisant les protocoles de composants en Promela et spécifier les exigences SysML avec la LTL pour utiliser le Model-Checker SPIN. Le choix de SPIN est motivé, d'une part par sa puissance en tant que Model-Checker, et d'autre part par son langage Promela disposant de primitives et de concepts qui facilitent l'implémentation des diagrammes de séquence.

Par exemple, les figures 4.4 et 4.6 montrent les interfaces des composants Sensors et ACU, décrites par leurs diagrammes de séquence respectifs. Dans ces diagrammes, les messages échangés spécifient les services offerts en tant qu'appels de l'environnement et les services requis en tant qu'appels à l'environnement.

Eléments d'un diagramme de séquence	**Elément de Promela**	**Instruction Promela**
ligne de vie	processus	`proctype{...}`
message	message	`mtype{m1,...,mn}`
communication	canal de communication	`chan chanName = [1] of {mtype}`
événements d'envoi et de réception	opérations d'envoi et de réception	envoi : `ab!m`, réception : `ab?m`

le fragment combiné Alt	l'instruction if	`if`
		`::(guard)->ab_p?p;`
		`:: else -> ab_q?q;`
		`fi;`

le fragment combiné Loop	l'instruction do	`do`
		`::ab_p?p;`
		`od`

TABLE 4.1 – Implémentation des concepts basiques d'un diagramme de séquence avec Promela

4.6.1/ VÉRIFICATION AVEC SPIN

La table 4.1 montre l'implémentation en Promela des principaux éléments d'un diagramme de séquence. Les figures 4.5 et 4.7 montrent partiellement le code Promela pour implémenter les diagrammes de séquence des composants Sensors et ACU (le code complet est présenté dans le chapitre 7 de la thèse de Oscar Carillo [Roz15]). Dans les deux diagrammes, nous remarquons que leurs deux lignes de vie sont traduites en processus dans le code Promela, un processus pour le composant et un autre pour l'environnement. Les deux processus sont activés dans une instruction atomique dans le processus d'initialisation `init`. Nous remarquons également que les fragments combinés *loop* sont traduits en instructions `do`. Le fragment combiné alternatif, *alt*, dans le diagramme de séquence de ACU est traduit par une instruction `if`. Les trois valeurs possibles pour la décélération sont attribuées dans `init` à l'aide d'une clause `if`, de cette manière, SPIN choisi de manière non déterministe la valeur qui sera utilisée pour simuler le système.

Une fois le diagramme de séquence implémenté en Promela, le composant peut être simulé en tant que système SPIN. Cependant, afin de vérifier si le composant respecte une propriété LTL, nous proposons d'utiliser une série d'indicateurs, appelés drapeaux (*flags* en anglais), pour garder la trace de *qui envoie/reçoit quel message à/de qui*, à tout moment de l'exécution. Ces drapeaux sont nécessaires pour pouvoir spécifier des propriétés LTL sur le comportement décrit par un diagramme de séquence. Ils sont mis à jour ensemble à chaque événement *d'envoi/de réception* à l'aide d'une instruction `d_step`. Les drapeaux de notre exemple dans la figure 4.5 sont `send` et `receive` pour indiquer l'action effectuée, et `msg_get_sensor_values` et `msg_sensor_values` pour indiquer le message échangé, et `sensors` et `environment` pour indiquer qui a effectué l'action.

4.6.2/ SPÉCIFICATION DES EXIGENCES AVEC LA LTL

Une fois que les drapeaux définis, les propriétés à vérifier peuvent être écrites sous forme d'expressions LTL avec les drapeaux. Dans notre approche, nous proposons de traduire les exigences SysML en propriétés LTL en respectant le formalisme avec les drapeaux. Ainsi, par exemple, l'exigence $R1.1.1$ dans la figure 4.3 peut être exprimée sous la forme :

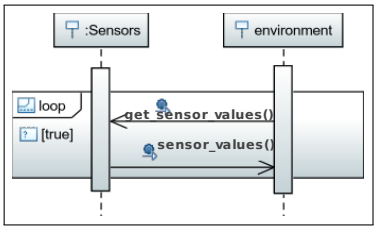


FIGURE 4.4 – Le DS du composant Sensors

```
            ...
proctype proc_sensors(){
     do
      sensors_environment_get_sensor_values?get_sensor_values;
      d_step{send=0; receive=1; msg_get_sensor_values=1;
      msg_sensor_values=0; sensors=1; environment=0;};
      sensors_environment_sensor_values!sensor_values; ...
     od
      }
proctype proc_environment(){
     do
      sensors_environment_get_sensor_values!get_sensor_values;
       ...
      sensors_environment_sensor_values?sensor_values; ...
     od
      }
init{atomic{run proc_sensors();run proc_environment();}}
```

FIGURE 4.5 – Le code Promela pour le composant Sensors

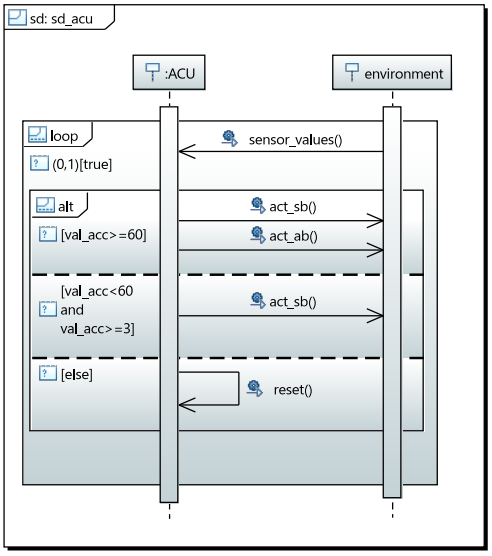


FIGURE 4.6 – Le DS du composant ACU

```
   ...
proctype proc_acu(){
 do
 ::acu_environment_sensor_values?sensor_values;

   if
    ::(val_dec>=60)->{acu_environment_act_sb!act_sb; ...
     acu_environment_act_ab!act_ab;
     d_step{send=0; receive=1; ...};}
   ::((val_dec<60) && (val_dec>=3))->
    acu_environment_act_sb!act_sb; ...
   ::else{acu_reset!reset; ...
       acu_reset?reset; ...}
   fi;

 od }

proctype proc_environment(){
 do
  ::acu_environment_sensor_values!sensor_values; ...
   if
    ::((val_dec<60) && (val_dec>=3))->
     acu_environment_act_sb?act_sb; ...
    ::(val_dec>=60)->{acu_environment_act_sb?act_sb; ...
     acu_environment_act_ab?act_ab; ...}
   fi;

 od}

init{
    if
     ::(true)->val_dec=0;
     ::(true)->val_dec=10;
     ::(true)->val_dec=60;
   fi;

   atomic{run proc_acu();run proc_environment();}

     }
```

FIGURE 4.7 – Le code Promela pour le composant ACU

toujours après avoir reçu un appel get_sensor_values, le bloc Sensors enverra un message avec sensor_values. La formule LTL exprimant cette exigence est :

$$\Box((\texttt{sensors \&\& receive \&\& msg_get_sensor_values}) \rightarrow \Diamond\ (\texttt{sensors \&\& send \&\& msg_sensor_values}))$$

De même, l'exigence $R1.1.2$ peut être exprimée comme suit : *toujours après avoir reçu un message avec le sensor_values, l'ACU enverra un message décidant de verrouiller la ceinture de sécurité, d'activer l'Airbag ou d'attendre un autre appel*. L'expression LTL exprimant cette exigence est :

$\Box((\texttt{acu \&\& receive \&\& msg_sensor_values}) \rightarrow \Diamond\ (\texttt{acu \&\& send \&\& (msg_reset ||}$

```
msg_act_sb || msg_act_ab)))
```

Ces propriétés sont ensuite vérifiées à l'aide SPIN, pour savoir si les composants satisfont les propriétés spécifiant leurs exigences. Une fois la vérification d'une exigence est réalisée sur son composant, le processus continue pour traiter l'ensemble des exigences afin de construire d'une manière incrémentale l'architecture du système.

4.7/ ASSEMBLAGE DES COMPOSANTS ET PRÉSERVATION DES EXIGENCES SYSML

Dans cette section, nous présentons l'étape de notre approche concernant la vérification de l'assemblage des composants. Cette étape vient après la vérification d'une exigence sur un composant. Ainsi, avant d'intégrer le composant vérifié dans l'architecture partielle du système, nous vérifions sa compatibilité avec le composant ou le composite formant cette architecture. Pour cela, nous générons des automates d'interface à partir des diagrammes de séquence, puis nous vérifions leur compatibilité en appliquant l'algorithme présenté dans la section 2.3.1 du chapitre 2.

Dans cette étape, nous proposons également de vérifier, en même temps que la vérification de la compatibilité, la préservation des exigences SysML atomiques lors de la composition. En effet, cette vérification permet d'éviter la vérification de ces exigences sur le composant composite obtenu. Ce qui permet la réduction du problème d'explosion de l'espace d'états lors du Model-Checking.

Avant de présenter l'algorithme pour vérifier la préservation des exigences, nous montrons dans les sections suivantes que les exigences SysML considérées dans cette approche, sont liées aux actions d'entrée/sortie des automates d'interface, et par conséquent, leur préservation dépend de la préservation, par la composition, des actions avec lesquelles elles sont liées.

4.7.1/ LES EXIGENCES FONCTIONNELLES ET LES ACTIONS D'ENTRÉE/SORTIE

Les exigences atomiques considérées dans ce travail concernent les propriétés fonctionnelles d'un SBC. Elles sont directement liées aux actions d'entrée/sortie des composants. En effet, chaque exigence atomique exprime une contrainte sur un ensemble d'actions d'entrée et de sortie d'un composant.

Soit AR l'ensemble des exigences atomiques dans la spécification RD d'un diagramme des exigences. Soit R_i une exigence atomique satisfaite par le composant C_i. Soit A_i l'automate d'interface décrivant le protocole de C_i. Ainsi, R_i est associée aux actions d'entrée $I_{R_i} = \{i_{ri_1}, ..., i_{ri_n}\}$ et aux actions de sortie $O_{R_i} = \{o_{ri_1}, ..., o_{ri_n}\}$.

Par exemple, la première exigence atomique dans notre étude de cas est $R1.1.1$: *toujours obtenir les valeurs du capteur et les envoyer à l'ACU*. Elle est satisfaite par le composant Sensors. L'automate d'interface de ce composant est décrit dans la figure 4.8. L'ensemble des actions d'entrée associées à $R1.1.1$ est $\{get_sensor_values\}$ et celui des actions de sortie est $\{sensor_values\}$.

Dans les automates d'interface modélisant les protocoles des composants, les actions

associées à leurs exigences atomiques sont formalisées par des transitions étiquetées avec ces actions d'entrée/sortie.

Définition 4.3 (Exigences connectées)**.** *Soient R et R' deux exigences atomiques spécifiées dans un diagramme des exigences SysML. R et R' sont liées respectivement à l'ensemble des actions d'entrée I_R et $I_{R'}$ et aux actions de sortie O_R et $O_{R'}$. R et R' sont connectées si $I_R \cap O_{R'} \neq \emptyset$ ou $I_{R'} \cap O_R \neq \emptyset$.*

Selon la définition 4.3 et selon la condition de composabilité des automates d'interface (voir la section 2.3), il est évident de déduire que deux composants satisfaisant deux exigences atomiques connectées sont composables.

Nous exploitons cette dernière définition dans l'étape 4-a (voir la section 4.2) de notre approche, de la façon suivante : à chaque i ème itération de notre approche, il est judicieux de sélectionner une nouvelle exigence atomique à traiter, qui est liée à l'exigence de l'itération $i - 1$, afin de composer leurs composants, sinon la composition n'est pas possible selon l'approche des automates d'interface.

4.7.2/ Préservation des actions d'entrée/sortie par la composition des automates d'interface

Dans cette section, nous montrons que la composition de deux automates d'interface, malgré leur compatibilité, ne garantit pas la préservation de leurs actions d'entrée/sortie non partagées dans l'automate composite obtenu. Dans notre approche, nous considérons q'une action est préservée par la composition si il existe au moins une transition étiquetée par cette action après la composition.

D'après la définition 2.5 de la composition des automates d'interface, les auteurs dans [dAH01], indiquent que l'ensemble des actions de l'automate composite $A = A_1 \parallel A_2$ est identique à la ensemble des actions dans le produit synchrone $A_1 \otimes A_2$, toutefois l'ensemble des transitions dans A n'est pas identique à celui de $A_1 \otimes A_2$ (selon la même définition), car ce dernier est inclus dans celui de $A_1 \otimes A_2$. Donc, il peut y avoir des actions d'entrée/sortie dans Σ_A qui n'étiquettent aucune transition dans A. En effet, selon l'approche optimiste des automates d'interface, même si A_1 et A_2 sont compatibles (donc $A \neq \emptyset$), il peut exister des actions d'entrée/sortie partagées entre A_1 et A_2 qui ne synchronisent pas, ce qui provoque des états bloquants. Par conséquent, les transitions étiquetées avec les actions d'entrée/sortie partagées, qui ne se synchronisent pas, seront éliminées de $A = A_1 \parallel A_2$ car elles mènent vers des états bloquants. Donc la composition ne préserve pas les actions qui étiquettent des transitions qui mènent vers des états bloquants.

4.7.3/ Vérification de la préservation des exigences atomiques

Suite au constat présenté précédemment concernant la préservation des actions d'entrée/sortie par la composition, nous présentons ici la condition que doit respecter la composition des composants, afin de préserver les exigences atomiques correspondant aux composants assemblés. Et nous montrons également comment vérifier cette condition en adaptant l'algorithme de vérification de la compatibilité des automates d'interface.

Condition de la préservation des actions d'entrée/sortie : Une action act (entrée ou sortie) est préservée par la composition de deux automates d'interface, A_1 et A_2, si au moins une transition existe dans l'automate composite, $A = A_1 \parallel A_2$, étiquetée avec act. L'action act appartient à l'ensemble des actions d'entrée/sortie dans A, lorsque act n'est pas partagée entre A_1 et A_2, sinon elle appartient aux actions internes dans A.

La condition de préservation des exigences atomiques par la composition des automates d'interface est énoncée par le théorème suivant.

Théorème 4.2 (Préservation des exigences par la composition)**.** *Soient C_i et C_{i+1} deux composants satisfaisant respectivement les exigences atomiques $R_i$et R_{i+1}. Leurs deux automates d'interface respectifs sont A_i et A_{i+1}. Soient I_i et O_i les actions d'entrée et de sortie liées à R_i. Et I_{i+1} et O_{i+1} celles qui sont liées à R_{i+1}. Le composant composite $S = C_i \parallel C_{i+1}$ préserve les exigences $\{R_i, R_{i+1}\}$ si les automates A_i et A_{i+1} sont compatibles et les actions d'entrée et de sortie I_i, I_{i+1}, O_i et O_{i+1} sont préservées dans l'automate $A_i \parallel A_{i+1}$.*

La preuve de ce théorème est détaillée dans notre papier [CCM13a].

Vue d'ensemble de l'algorithme de vérification : Pour vérifier la préservation des exigences atomiques par la composition, nous proposons d'adapter l'algorithme de vérification de la compatibilité présenté dans la section 2.3.1 du chapitre 2. L'objectif est de vérifier si les transitions étiquetées avec des actions d'entrée/sortie, liées à des exigences atomiques, sont conservées dans l'ensemble de transitions de l'automate composite obtenu, $A_i \parallel A_{i+1}$. Donc l'adaptation consiste à :

— calculer à l'étape (2) de l'algorithme de vérification de la compatibilité, l'ensemble des transitions dans $A_i \otimes A_{i+1}$, noté T, liées aux exigences,

— lorsque nous éliminons les transitions à l'étape (5) de l'algorithme de vérification de compatibilité, nous éliminons également ces transitions dans T,

— enfin, nous vérifions que toutes les actions liées aux exigences sont associées à au moins une transition dans T, après l'étape (6).

4.8/ SPÉCIFICATION DE L'ARCHITECTURE DU SYSTÈME

La construction d'un SBC avec notre approche est basée sur la construction, à chaque étape, d'un composant composite, qui définit une architecture partielle du système qui satisfait un ensemble des exigences atomiques. Nous décrivons cette architecture avec les diagrammes SysML BDD et IBD, commme indiqué ci-dessous.

Soit C_i le composant de l'étape i respectant l'exigence R_i. Ce composant et compatible avec le composant composite C_{i-1} de l'étape $i-1$. Ces deux composants sont spécifiés respectivement avec leurs automates respectifs, A_i et A_{i-1}.

— Le BDD de l'architecture partielle du système est composée des deux blocs C_i et C_{i-1}. Le bloc C_i possède un port d'entrée typé avec un bloc interface décrivant les actions d'entrée spécifiées dans l'automate A_i et un port de sortie typé par un bloc interface décrivant les actions de sortie spécifiées dans A_i. Le bloc C_{i-1} est également décrit avec deux ports typés avec des blocs interfaces décrivant les actions spécifiés dans l'automate A_{i-1}.

— Le IBD est composé des parties (*parts*) des blocs C_i et C_{i-1}, qui communiquent entre elles via deux ports d'entrée/sortie, typés par des blocs interfaces décrivant les actions partagées entre A_i et A_{i-1}. Ces parties communiquent également avec l'extérieur via des ports d'entrée/sortie typés avec des blocs interfaces décrivant les actions non partagées entre A_i et A_{i-1}.

Les éléments traités dans cette section sont illustrés dans la section suivante.

4.9/ ILLUSTRATION SUR L'ÉTUDE DE CAS

Dans cette section, nous appliquons notre approche sur l'étude de cas présentée dans la section 4.4.

Nous commençons par analyser le diagramme des exigences SysML, décrit dans la figure 4.3, pour extraire les exigences atomiques : $R1.1.1$, $R1.1.2$, $R1.1.3$ et $R1.2$. Ensuite, nous exprimons ces exigences avec des propriétés LTL, pour les vérifier sur les protocoles des composants décrits par leurs diagrammes de séquence.

Pour la première exigence $R1.1.1$, nous sélectionnons le composant *Sensors*, son diagramme de séquence est illustré dans la figure 4.4. Ce composant reçoit des informations de plusieurs capteurs (accéléromètres, capteurs de choc, ...) tout autour de la voiture à chaque appel du service `get_sensor_values` et les envoie via un service `sensor_values` . Ces services sont décrits respectivement par l'action d'entrée { `get_sensor_values` } et l'action de sortie { `sensor_values` }, qui sont liées à l'exigence $R1.1.1$. Pour vérifier si le composant *Sensors* satisfait l'exigence $R1.1.1$, nous traduisons son diagramme de séquence associé en Promela. Ensuite, nous formalisons l'exigence comme une propriété LTL : " *toujours, après la réception de Sensors d'un appel de* `get_sensor_values`*, il envoie un message* `sensor_values` *à l'environnement*". Ainsi, la formule LTL spécifiant l'exigence $R1.1.1$ est :

```
□((sensors && receive && msg_get_sensor_values)
        → ◊(sensors && send && msg_sensor_values))
```

La prochaine exigence à analyser est $R1.1.2$, qui est associée aux actions d'entrée `sensor_values` } et aux actions de sortie { `act_sb, act_ab` }. Elle est donc connectée à l'exigence $R1.1.2$. Nous proposons de la vérifier sur le composant *ACU* qui offre et requière les services représentés par les actions d'entrée/sortie associées à $R1.1.2$. Son diagramme de séquence est décrit dans la figure 4.6.Pour vérifier si ce composant satisfait l'exigence $R1.1.2$, nous implémentons son diagramme de séquence en Promela et exprimons l'exigence sous la forme d'une propriété LTL : "*toujours après avoir reçu un message avec* `sensor_value`*, l'ACU enverra un message, pour verrouiller la ceinture de sécurité (*`act_sb`*), ou activer l'airbag (*`act_ab` *), ou attendre un autre appel (*`reset`*)*", la formule LTL exprimant cette exigence est :

```
□((acu && receive && msg_sensor_values)
      → ◊ (acu && send && (msg_reset || msg_act_sb || msg_act_ab)))
```

Ces propriétés sont vérifiées à l'aide du Model-Checker SPIN, qui ne génère aucune erreur pour les deux modèles. Les composants *ACU* et *Sensors* satisfont donc les exigences.

Ensuite, pour relier les composants respectant les exigences $R1.1.1$ et $R1.1.2$, nous

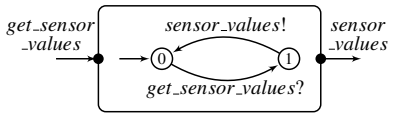


FIGURE 4.8 – L'automate d'interface de Sensors

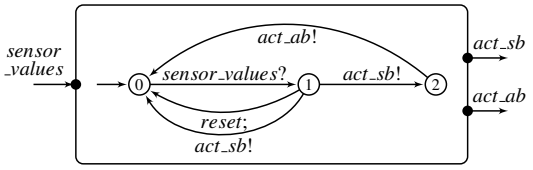


FIGURE 4.9 – L'automate d'interface de ACU

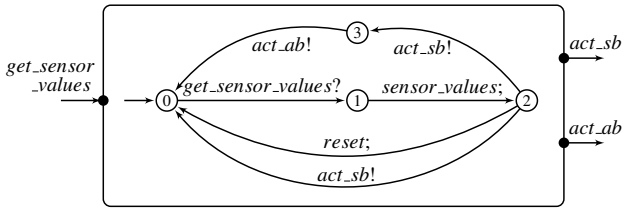


FIGURE 4.10 – L'automate d'interface de la composition des composants Sensors et ACU

vérifions leur compatibilité grâce à leurs automates d'interface. Ces automates sont générés à partir des diagrammes de séquence en suivant l'approche décrite dans [CH11b]. Ils sont représentés dans les figures 4.8 et 4.9. Pour vérifier leur compatibilité, nous calculons leur composition. L'automate composite obtenu est illustré dans la figure 4.10. Cet automate n'est pas vide, donc les composants *Sensors* et *ACU* sont compatibles.

En examinant les transitions dans l'automate composite, nous constatons que l'ensemble des actions d'entrée/sortie, liées aux exigences, est toujours présent, par conséquent les exigences vérifiées sont préservées par la composition des deux automates d'interface modélisant les protocoles des composants *Sensors* et *ACU.* Donc nous pouvons définir une architecture partielle du système, en présentant un BDD d'un bloc composite, composé des blocs *Sensors* et *ACU*. Ce diagramme est présenté dans la figure 4.11. Les interactions entre les blocs composés sont ensuite décrites par un IBD (voir la figure 4.12).

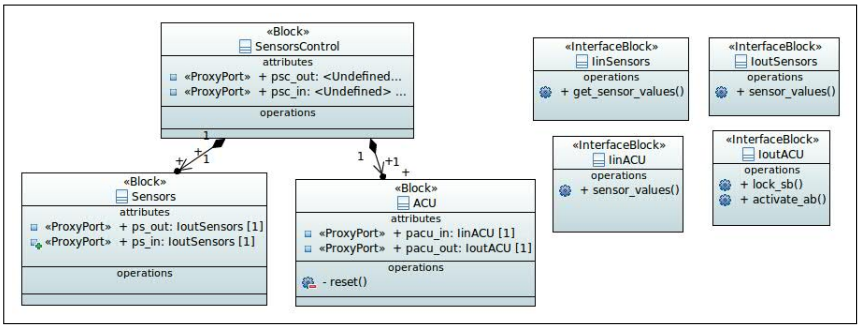


FIGURE 4.11 – Le BDD du bloc *SensorsControl*

Ensuite, nous continuons l'application de notre approche en traitant, de la même manière, les exigences $R1.1.3$, avec l'action d'entrée qui lui est associée $\{$`act_ab` $\}$ et $R1.2$, avec l'action d'entrée associée $\{$`act_sb` $\}$.

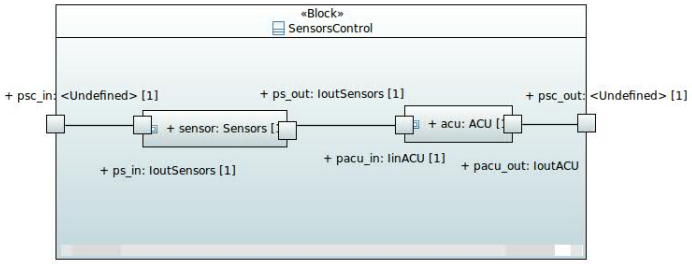


FIGURE 4.12 – Le IBD du bloc *SensorsControl*

4.10/ CONCLUSION ET PERSPECTIVES

Dans ce chapitre, nous avons proposé une approche permettant de spécifier, d'une manière incrémentale, l'architecture d'un SBC directement à partir de ses exigences fonctionnelles modélisées avec SysML. Pour cela, nous exploitons le diagramme des exigences SysML pour en extraire les exigences atomiques, qui sont ensuite vérifiées par Model-Checking sur des composants réutilisables. Ces derniers sont ensuite intégrés à une architecture partielle du système, après avoir vérifié leur compatibilité. Cette architecture est complétée au fur à mesure du traitement de l'ensemble des exigences, jusqu'à l'obtention de l'architecture finale du système global.

Dans ce travail, nous nous sommes restreints aux exigences locales aux composants, autrement dit chaque exigence est satisfaite par un composant. Il serait donc intéressant d'étendre notre approche pour prendre en considération les exigences globales, en lien avec un ensemble de composants.

4.11/ BILAN

Les résultats de ce chapitre concernent les éléments suivants :

1. **Projet** :
   — Le projet Tassili SYSVAP (spécifier en vue de Vérifier et d'évaluer les performances des systèmes complexes), coopération avec l'USTHB Alger (Algérie), 2013-2016.
2. **Encadrement de thèse de doctorat** :
   — La thèse de Oscar Carillo [Roz15].
3. **Encadrement de stage de Master 2** :
   — Stage M2 recherche de Moshe Abtan, 2018.
   Titre : Étude de l'impact de l'évolution des exigences SysML dans le développement des systèmes à base de composants.
4. **Publications** : [CCM14a, CCM13b].

III

Assemblage des composants basé sur leurs contrats comportementaux

5

Conception à base de composants orientés objet

5.1/ Introduction

Nous nous intéressons dans ce chapitre, à la conception rigoureuse des systèmes critiques à base de composants dans le but de garantir leur fiabilité. Notre principale motivation est de fournir des solutions innovantes au développement des systèmes corrects par la conception. Pour cela, nous proposons une approche formelle pour la conception des systèmes à base des composants orientés objet utilisant leurs contrats comportementaux. Ainsi, nous présentons un formalisme, appelé contrat comportemental, qui décrit pour chaque composant, à la fois son protocole d'interaction spécifié avec un automate d'interface, et la sémantique de ses opérations, dans le contexte de composants orientés objet. Les contrats comportementaux nous permettent de vérifier l'interopérabilité des composants aux niveaux signature, protocole, et enfin sémantique, contrairement à l'approche des automates d'interface [dAH01] qui se restreint aux niveaux signature et protocole. En effet la considération de la sémantique des opérations lors de l'assemblage des composants est souvent indispensable pour garantir leur interopérabilité. Outre l'assemblage des composants, nous traitons également le raffinement et l'adaptation des composants.

Par ailleurs, ce travail est en partie le résultat d'une collaboration avec deux chercheurs/ingénieurs expérimentés dans le domaine de la conception des systèmes informatiques pour l'industrie ferroviaire, bénéficiant ainsi de leurs retours critiques. L'élaboration de notre approche tient compte des recommandations de la norme européenne EN 50128 : 2011 [cen11] des applications ferroviaires, recommandant l'utilisation des approches par composants et des méthodes formelles, de la conception à la génération de code, et la répartition de la complexité de la vérification tout au long du cycle de développement. Ainsi, tout au long du chapitre, nous justifions la pertinence de notre approche pour la conception d'un système ferroviaire en présentant une étude de cas sur les fonctions de protection des trains dans les systèmes de contrôle modernes CBTC (*Communications-Based Train Control*) [cbt] des chemins de fer.

Le reste du chapitre est organisé comme suit. Dans la section 5.2, sont présentés les travaux proches de notre approche. Dans la section 5.4, nous introduisons notre étude de cas pour être utilisée progressivement dans les autres sections pour valider les différents concepts formels. Dans les sections 5.3 et 5.5, nous passons à l'étude des contrats comportementaux, et notre approche de la compatibilité et de la composition des compo-

sants. La section 5.6 est consacrée à l'étude du raffinement des contrats comportementaux. Notre approche d'adaptation des composants est présentée dans la section 5.7. La conclusion est le bilan sont présentés respectivement dans les sections 5.9 et 5.10.

5.2/ ÉTAT DE L'ART ET CONTRIBUTIONS

Dans la littérature, il existe de nombreux travaux proposant des formalismes pour la modélisation et l'analyse des contrats des composants (et services), dans le but de vérifier principalement leur l'assemblage et leur substituabilité. Dans cette section nous nous focalisons sur ceux proposant l'extension du formalisme des automates d'interface. Par exemple, dans [dAHS02], les auteurs proposent les automates d'interface temporisés, permettant de considérer les contraintes temporelles, pour l'assemblage des composants des systèmes temps réel. Dans [DLL+10], une extension des automates d'interface temporisés est proposée pour prendre en charge le raffinement, la composition logique et structurelle, et le quotient des composants. D'autres travaux [RBB+09, LNW07], proposent d'enrichir le formalisme des automates d'interface avec les spécifications modales, basées sur la logique mu-calcul modale, pour le développement des SBC. Ils proposent le formalisme des interfaces modales qui est plus expressif que les automates d'interface, permettant ainsi de définir d'autres opérateurs de composition comme la conjonction, et aussi d'exprimer des propriétés de vivacité. Un travail concernant les contrats des composants, qui s'inspire des formalismes des automates d'interfaces et des interfaces modales, est présenté dans [CFN05]. Les auteurs proposent une approche, basée sur les types de session, permettant de spécifier les interfaces des composants avec un langage de type d'interfaces comportementales, doté d'un ensemble de règles de compatibilité de sous typages. Ces travaux liés aux automates d'interface, n'ont pas traité la sémantique des actions comme c'est le cas dans notre approche. De plus l'adaptabilité des composants n'est pas également considérée.

Le travail présenté dans ce chapitre traite la problématique du développement de systèmes critiques à base de composants, corrects par la conception. Plusieurs travaux ont été proposés dans ce contexte. Nous commençons par la Méthode B [Abr96a] et son impact industriel. Cette méthode formelle est parmi les rares utilisée par de nombreux acteurs industriels majeurs tels que la RATP, Alstom, Siemens, etc [Lec09]. La conception d'un système avec B repose sur des raffinements successifs (conception incrémentale) de sa spécification abstraite jusqu'à obtenir son implémentation. Notre approche présente certains points de similitudes avec B concernant la conception incrémentale de systèmes, par le raffinement, et la préservation des propriétés entre le niveau abstrait et raffiné.

Toujours dans le contexte de la construction de systèmes corrects par la conception, nous citons le framework BIP [BBB+11], développé à Verimag. Il permet la conception des SBC avec des interactions complexes. Dans BIP, les SBC sont construits en superposant trois couches de modélisation : comportement, interaction, et priorité. Ainsi, l'architecture du système est représentée par les couches interaction et priorité, et son comportement est modélisé par les systèmes de transition qui spécifient les comportements des composants atomiques. Ce qui distingue notre proposition de BIP est notre exploitation de l'approche optimiste des automates d'interface dans un contexte orienté objet pour la conception de systèmes.

Dans le domaine ferroviaire, les auteurs dans [ZTL+11], proposent une description for-

melle pour les protocoles de communications sûres, dans les systèmes de contrôle de trains, en utilisant des automates d'interface et des diagrammes de séquence UML [uml]. Des propriétés de sûreté ont été vérifiées sur l'étude de cas proposée avec le Model-Checker SPIN [Hol97]. Les résultats montrent à la fois l'efficacité potentielle et l'utilité pratique de leur approche. Dans [PQ09], les auteurs proposent une approche pour vérifier un ensemble de propriétés (sûreté, vivacité,...) concernant la sûreté de fonctionnement du protocole de coopération du système européen de contrôle des trains (ETCS) [Eur10], en utilisant le prouveur de théorème déductif KeYmaera [PQ08]. Dans [BMC+12], les auteurs proposent une approche pour la validation et la vérification d'un système ferroviaire en exploitant SysML et la méthode B. Leur proposition repose sur la définition d'une transformation entre les deux langages en préservant leur sémantique. Contrairement à notre approche, les aspects liés à la conception et à la vérification des SBC ne sont pas considérés dans ces derniers travaux.

Contributions : La première contribution de ce travail est de montrer comment la conception à base de composants orientée objet est rigoureuse grâce aux contrats comportementaux fusionnant les automates d'interface avec la sémantique des méthodes. L'approche optimiste de la composition des automates d'interface est par conséquent adaptée pour prendre en considération les caractéristiques des interactions entre les composants orientés objet. La seconde contribution concerne la définition de la relation de raffinement des composants utilisant des contrats comportementaux, destinée à assurer leur implémentabilité indépendante. Et enfin, la troisième contribution, est la proposition d'une approche d'adaptation des composants décrits par leurs contrats comportementaux.

5.3/ CONTRATS COMPORTEMENTAUX

Dans cette section, nous présentons le formalisme des contrats comportementaux permettant de décrire à la fois les protocoles des composants orientées objet, spécifiés avec les automates d'interfaces, et la sémantique de leurs méthodes. Avant la définition d'un contrat comportemental, nous présentons la spécification des protocoles des composants orientés objet avec les automates d'interface.

5.3.1/ FORMALISATION DES PROTOCOLES DES COMPOSANTS ORIENTÉS OBJET

Nous proposons d'utiliser et d'enrichir le formalisme des automates d'interface [dAH01], défini dans la section 2.3 du chapitre 2, pour modéliser les protocoles de communication des composants orientés objet, sous forme d'ordonnancements temporels de leurs actions d'entrée, de sortie, et internes. Ainsi, dans la conception des composants orientés objet :

- Les actions d'entrée peuvent représenter des méthodes publiques fournies, l'attribution des valeurs de retour, et la capture des exceptions.
- Les actions de sortie peuvent représenter des appels de méthodes, des événements de retour, et le lancement des exceptions.
- Les actions internes peuvent représenter des appels de méthodes privées, l'attribution de leurs valeurs de retour, et la capture de leurs exceptions levées.

Définition 5.1 (Automate d'interface pour un composant orienté objet)**.** *Un automate d'interface pour modéliser le protocole d'un composant orienté objet est représenté par le tuple* $A = (S_A, \iota_A, \Sigma_A^{\mathrm{I}}, \Sigma_A^{\mathrm{O}}, \Sigma_A^{\mathrm{H}}, \delta_A)$*, tel que :*

- S_A *est l'ensemble fini des états ;*
- $\iota_A \in S_A$ *est l'état initial ;*
- Σ_A^{I}, Σ_A^{O}, *et* Σ_A^{H} *sont respectivement les ensembles des actions d'entrée, de sortie, et internes, tels que :*
  - $\Sigma_A^{\mathrm{I}} \cup \Sigma_A^{\mathrm{O}} \cup \Sigma_A^{\mathrm{H}} = \Sigma_A$,
  - $\Sigma_A^{\mathrm{I}} = \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Ir}} \cup \Sigma_A^{\mathrm{Ie}}$, *les ensembles* Σ_A^{Im}, Σ_A^{Ir}, *et* Σ_A^{Ie}, *sont respectivement des actions des méthodes publiques fournies, l'attribution des valeurs de retour, et la capture des exceptions.*
  - $\Sigma_A^{\mathrm{O}} = \Sigma_A^{\mathrm{Om}} \cup \Sigma_A^{\mathrm{Or}} \cup \Sigma_A^{\mathrm{Oe}}$, *les ensembles* Σ_A^{Om}, Σ_A^{Or}, *et* Σ_A^{Oe}, *sont respectivement des appels des méthodes publiques de l'environnement, le renvoi des valeurs de retour, et le lancement des exceptions.*
  - $\Sigma_A^{\mathrm{H}} = \Sigma_A^{\mathrm{Hm}} \cup \Sigma_A^{\mathrm{Hr}} \cup \Sigma_A^{\mathrm{He}}$, *les ensembles* Σ_A^{Hm}, Σ_A^{Hr}, *et* Σ_A^{He}, *sont des appels des méthodes privées, l'attribution de leurs valeurs de retour, ou la capture de leurs exceptions*
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *est l'ensemble des transitions.*

L'ensemble Σ_A^{m} des *actions des méthodes* de A est défini par $\Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Om}} \cup \Sigma_A^{\mathrm{Hm}}$. De la même manière les ensembles Σ_A^{r} des actions de retour Σ_A^{e} pour les actions des exceptions, sont définies respectivement par, $\Sigma_A^{\mathrm{Ir}} \cup \Sigma_A^{\mathrm{Or}} \cup \Sigma_A^{\mathrm{Hr}}$, et $\Sigma_A^{\mathrm{Ie}} \cup \Sigma_A^{\mathrm{Oe}} \cup \Sigma_A^{\mathrm{He}}$.

Pour définir la signature des actions de méthodes dans un automate d'interface, nous considérons un ensemble de variables V, et nous définissons par $\mathbb{T}[v]$ le type de $v \in V$, et par $\mathbb{T}[\![V]\!] = \bigcup_{v \in V} \mathbb{T}[v]$ le type de V.

Définition 5.2 (Signature d'une action)**.** *Soit* A *un automate d'interface, pour toute action* $a \in \Sigma_A^{\mathrm{m}}$ *est associée une signature notée* $a(i_1{:}\mathbb{T}[i_1], ..., i_k{:}\mathbb{T}[i_k]) \rightarrow o{:}\mathbb{T}[o] \sharp\, e$ *telle que*

- *L'ensemble des paramètres d'entrée de* a *est* $\Psi_A^{\mathrm{i}}(a) = \{i_1, ..., i_k\}$.
- *L'ensemble des paramètres de retour* $\Psi_A^{\mathrm{o}}(a)$ *de* a *est le singleton* $\{o\}$*. Nous notons par* $\mathsf{R}_A(a) = o$ l'action retour *de* a*, et* $\mathsf{E}_A(a) = e$ l'action exception *de* a*.*
- *L'ensemble des attributs utilisés dans* a *est défini par* $\Lambda_A(a)$ *si* $a \in \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Hm}}$*. Si* $\mathsf{R}_A(a)$ *et* $\mathsf{E}_A(a)$ *sont définis,* Σ_A^{r} *et* Σ_A^{e} *sont associés respectivement à* $\{\mathsf{R}_A(a) \mid a \in \Sigma_A^{\mathrm{m}}\}$ *et* $\{\mathsf{E}_A(a) \mid a \in \Sigma_A^{\mathrm{m}}\}$.

Pour une transition $(s, a, t) \in \delta_A$, nous définissons $Succ_A(s, a) = t$. Un *chemin d'exécution* σ de A est une séquence alternée finie $s_0[a_0]....s_k[a_k]s_{k+1}..[a_{n-1}]s_n$ d'états et d'actions où $(s_k, a_k, s_{k+1}) \in \delta_A$ pour tout $k \in \mathbb{N}_{<n}$. Nous notons par $\Sigma_A\langle\sigma\rangle$, l'ensemble $\{a_k \in \Sigma_A \mid k \in \mathbb{N}_{<n}\}$ et, par $\Theta_A(s)$, l'ensemble des exécutions atteignant $s \in S_A$ à partir de ι_A. Un état $s \in S_A$ est *atteignable* dans A si $\Theta_A(s) \neq \emptyset$.

Exemple 5.1. *Nous présentons un exemple d'un automate d'interface (voir la figure 5.1) décrivant le comportement d'un composant qui offre principalement une méthode* p*. Au niveau de son implémentation, il y a l'appel d'une méthode externe* c*, suivi soit, de la réception d'une valeur de retour* r_c*, l'appel d'une méthode privée (action interne)* h *et le calcul de* rp*, ou la capture d'une exception et le lancement d'une nouvelle. Dans l'automate d'interface les flèches en double représentent les actions de méthodes, les simples c'est pour les actions de retour, et celles en pointillés pour les exceptions.*

```
method p (ip1:int,ip2:int)
returns rp:int; throws ep
 begin
     int ap=0;
        try
        rc = c(ic1,ic2);
        h(ih1,ih2);
        //calcul de  rp
        rp= 2+ 5lv;
        return rp;
        catch ec -> raise ep
 end
```

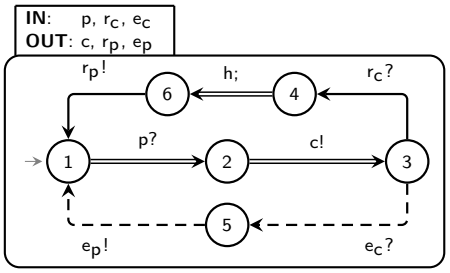


FIGURE 5.1 – Illustration d'un automate d'interface

AUTOMATE D'INTERFACE BIEN FORMÉ

Les chemins d'exécutions des automates d'interface doivent respecter, vis à vis des actions de méthodes, les règles d'implémentation orientées objet. Par exemple, Un appel d'une méthode non void, fait par un composant nécessitant l'attribution de sa valeur de retour, doit être spécifié au moins par une séquence débutant et se terminant respectivement par une action de sortie de méthode et une action d'entrée de retour.

Pour l'illustration nous considérons la méthode c dans l'exemple 5.1, et le chemin d'exécution suivant qui est bien formé : $1[p?]2[c!]3[r_c?]4$.

Actions Autonomes. Toutes les actions d'un composant sont *autonomes* (donc s'activent forcement), sauf les actions d'entrée de méthodes et les actions d'entrée liées aux exceptions. En effet c'est l'environnement qui décide d'activer ou non ces actions. L'ensemble Σ_A^{aut} des actions autonomes est $\Sigma_A \setminus (\Sigma_A^{\text{Im}} \cup \Sigma_A^{\text{Ie}})$.

Nous définissons par $\Sigma_A^*(s)$ où $* \in \{\text{I}, \text{O}, \text{H}, \text{Im}, \text{Om}, \text{Hm}, \text{Ir}, \text{Or}, \text{Hr}, \text{Ie}, \text{Oe}, \text{He}, \text{m}, \text{r}, \text{e}, \text{aut}\}$ l'ensemble des actions dans Σ_A^* activables de $s \in S_A$. L'ensemble $\Sigma_A(s)$ définit toutes les actions activables de s.

Le chemin d'exécution $\sigma = s_0[a_0]...[a_{n-1}]s_n$ est dit autonome dans A si $\Sigma_A\langle\sigma\rangle \subseteq \Sigma_A^{\text{aut}}$. Il est dit *sans exception* si $\Sigma_A\langle\sigma\rangle \subseteq \Sigma_A \setminus \Sigma_A^{\text{e}}$.

Un état $s' \in S_A$ est atteignable d'une manière autonome (respectivement sans exception) de $s \in S_A$ dans A si il existe un chemin d'exécution autonome (respectivement sans exception) entre s et s'.

Définition 5.3. *Un automate d'interface A est bien formé si pour tout $s \in S_A$, et $a \in \Sigma_A^{\text{m}}(s)$ telle que $\text{R}_A(a) \in \Sigma_A^{\text{r}}$, il existe au moins un état $t \in S_A$, tel que $\text{R}_A(a) \in \Sigma_A^{\text{r}}(t)$, et t atteignable d'une manière autonome sans exceptions de $Succ_A(s, a)$.*

5.3.2/ SÉMANTIQUE DES MÉTHODES

La sémantique d'une méthode fournie (action d'entrée) consiste en : (i) une précondition représentant les hypothèses environnementales sur les paramètres d'entrée, (ii) une

spécification abstraite sur le calcul du paramètre de retour exprimée à l'aide des paramètres d'entrée et des attributs, (iii) une postcondition sur le paramètre de retour en fonction des paramètres d'entrée et des attributs, et (iv) une postcondition supplémentaire décrivant les conditions d'exception sur les paramètres et les attributs. Une sémantique d'appel de méthode (action de sortie) n'est définie que par une précondition sur les paramètres d'entrée et une postcondition sur les paramètres d'entrée et de retour.

Étant donné un ensemble de variables V, une *condition* sur v est un sous-type de $\mathbb{T}[v]$. Nous notons par $Q[w1, ..., wn]$ (ou $Q[\![W]\!]$) la *projection* de Q sur les variables dans $W = \{w_1, ..., w_n\} \subseteq V$.

Définition 5.4 (Sémantique des actions)**.** *Soit un automate d'interface A,* une sémantique d'entrée $I_a = (P_a, S_a, Q_a, E_a)$ *d'une action* $a \in \Sigma_A^{\mathrm{Im}}$ *est définie par : une précondition* $P_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a)]\!]$*, une spécification* $S_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$*, une postcondition de terminaison* $Q_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$*, et une postcondition d'exception* $E_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$. Une sémantique de sortie $O_b = (P_b, Q_b)$ *d'une action* $b \in \Sigma_A^{\mathrm{Om}}$ *est définie par : une précondition* $P_b \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(b)]\!]$ *et une postcondition* $Q_b \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(b) \cup \Psi_A^{\mathrm{o}}(b)]\!]$*. Ces conditions sont respectivement notées par* $I_a.P$*,* $I_a.S$ *,* $I_a.Q$*,* $I_a.E$*,* $O_b.P$*, et* $O_b.Q$*.*

Nous définissons un contrat comportemental comme suit.

Définition 5.5 (Contrat comportemental)**.** *Un* contrat comportemental B_i *d'un composant est un tuple* $(\mathcal{A}_i, \mathcal{I}_i, \mathcal{O}_i)$ *tel que* $\mathcal{A}_i$ *est un automate d'interface,* $\mathcal{I}_i$ *est une fonction qui associe pour chaque* $a \in \Sigma_A^{\mathrm{Im}}$ *une sémantique d'entrée* I_a*, et* $\mathcal{O}_i$ *est celle qui associe pour chaque* $a \in \Sigma_A^{\mathrm{Om}}$ *une sémantique de sortie* O_a*. Nous notons par,* $B_i.\mathcal{A}$*, l'automate d'interface de* B_i*,* $B_i.\mathcal{I}$*, la fonction* $\mathcal{I}_i$ *de* B_i*, et* $B_i.\mathcal{O}$*, la fonction* $\mathcal{O}_i$ *de* B_i*.*

5.4/ ÉTUDE DE CAS D'UN SYSTÈME FERROVIAIRE

Nous présentons une étude de cas simplifiée sur les fonctions de protection des trains dans les systèmes CBTC (voir la figure 5.2). Un système CBTC est un contrôleur automatique des trains qui détermine continuellement les positions précises des trains. Il est composé des dispositifs embarqués dans les trains et installés dans les voies permettant la prise en charge des fonctions pour la protection automatique d'un train (ATP), ainsi que celles pour sa conduite automatique (ATO) et sa supervision (ATS). Les fonctions ATP assurent les exigences critiques pour la sécurité (vitesse, contrôle, freinage, etc.). Les fonctions ATO couvrent les fonctions non liées à la sécurité. Les fonctions ATS couvrent la gestion du trafic [cbt]. Les fonctions ATO et ATS n'ont pas d'impact significatif sur la sécurité et ne sont pas considérées dans cette étude de cas.

Les positions d'un train et sa vitesse sont calculées en continu, et sont communiquées via des équipements sans fil au bord de la voie. Ainsi, *une zone de circulation protégée* est établie pour chaque train jusqu'au prochain obstacle le plus proche. Le train est donc en mesure d'adapter sa vitesse et ses courbes de freinage afin de ne pas dépasser la limite de cette zone, appelée *point de danger* [SLMP13].

Chaque train est équipé d'un composant OBD (*On-Board Device*), qui permet le calcul de deux emplacements fictifs : les postions de la tête et de la queue du train (TEL et HEL). Le fragment de piste entre ces deux positions couvre tout le train. Habituellement, ce choix est pris pour des raisons de sécurité pour garder une distance de sécurité entre les trains

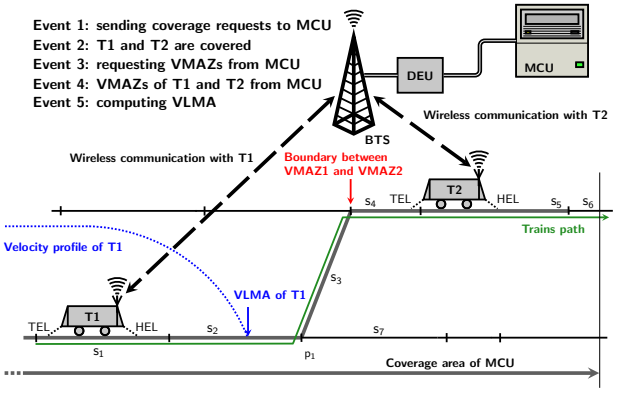


FIGURE 5.2 – Les fonctions simplifiées de la protection des trains.

en cas de dysfonctionnement du système. Les emplacements sont des coordonnées sur *un chemin des trains* composé de *segments* et placé dans une direction donnée selon des positions des *commutateurs ferroviaires*. Un segment est identifié par un numéro, une longueur et une coordonnée de début. Dans la figure 5.2, le commutateur p_1 est positionné sur le segment s_3, et le chemin des trains est la séquence s_1, s_2, s_3, s_4, s_5, s_6, etc.

Selon la figure 5.2, les OBD de T1 et T2 lancent le processus de protection en demandant s'ils sont visibles à une *Unité de contrôle du mouvement* (MCU). Plusieurs MCU couvrent l'ensemble de la ligne, avec des sections de couverture se chevauchant permettant un transfert d'informations en toute sécurité entre elles. Pour la simplicité, une seule est représentée dans notre étude de cas. Les emplacements de train sont envoyés par radio à la plus proche *station de transmission de base* (BTS). Cette dernière convertit les signaux radio en données numériques et les transmet à l'*unité d'échange de données* (DEU), qui les transfère à son tour au composant MCU (événement 1). MCU détermine si la zone entre TEL et HEL des trains est complètement ou partiellement incluse dans sa zone de couverture, et répond aux requêtes de T1 et T2. Dans la figure 5.2, T1 et T2 sont tous les deux visibles pour MCU (événement 2).

Ensuite, chaque train demande à son MCU de couverture, la *zone autorisée du mouvement vital* (VMAZ) : la zone (séquence de segments) dans laquelle le train peut circuler en toute sécurité (événement 3). Dans la figure 5.2, MCU envoie à T1 une VMAZ limitée par le début de s_1 (contenant ses TEL et HEL) et la fin de s_3, et envoie à T2 une VMAZ limitée par le début de s_4 (contenant son TEL) et la fin de s_5, dernier segment couvert par MCU (événement 4). Le composant MCU garantit que les VMAZ de deux trains successifs ne se chevauchent jamais pour éviter les collisions.

Enfin, le train calcule le point de danger, à savoir la *limite vitale pour l'autorisation du mouvement* (VLMA), au sein de VMAZ. Pour localiser la VLMA, OBD prend une marge de sécurité fixe avant la limite de sa VMAZ. La vitesse du train est progressivement réduite pour atteindre zéro lorsque HEL atteint la VLMA (événement 5).

5.4.1/ CONCEPTION DE L'ÉTUDE DE CAS FERROVIAIRE

La figure 5.3 représente l'architecture, modélisée avec UML, de l'équipement de protection ATP décrit dans la section précédente. Quatre classes de composants sont

considérés : OnBoardDevice, DataExchangeUnit, MovementControlUnit, et SubRoute-Builder instanciés respectivement par les composants OBD, DEU, MCU, et SRB. Les trois derniers implémentent respectivement les interfaces DataExchange, MovementControl, et RouteBuilder. Les protocoles des composants OBD, DEU, MCU, sont spécifiés par leurs automates d'interfaces respectifs dans la figure 5.4.

Par exemple, le composant DEU implémente la méthode publique (+) $covReq$ (demande de couverture), dont les arguments sont : tel et ts (la coordonnée de TEL et l'identifiant du segment contenant TEL), hel et hs (la coordonnée de HEL et l'identifiant du segment contenant HEL), et t (l'identifiant du train). Selon l'automate d'interface A_d de DEU (figure 5.4(b)), la méthode $covReq$ transfère la demande de couverture à MCU en invoquant la méthode $iscovered$. MCU répond à OBD, via DEU, en retournant 2 (resp. 1) si elle couvre complètement (resp. partiellement) le train (signal $covered$), ou en levant l'exception $uncouvered$ dans le cas contraire.

Par la suite, si le train est couvert par MCU, OBD demande sa VMAZ ($vmazReq$), DEU transfère la demande en appelant $computeVmaz$ implémentée par MCU, à son tour MCU appelle la méthode $chain$ de SRB pour effectuer le chaînage sur les segments afin de calculer les limites VMAZ dans la séquence des segments de $start$ à end (arguments de $chain$). Si MCU ne couvre que HEL, alors $start$ prend la valeur de l'identifiant du premier segment dans le chemin des trains entièrement couvert par MCU, sinon il prend la valeur de ts. L'argument end prend la valeur de l'identifiant du segment contenant l'obstacle suivant. Selon A_m dans la figure 5.4(c), si le chaînage est interrompu par un commutateur non contrôlé, MCU gère l'exception $uncontSW$, qui devrait être levée par $chain$, et à son tour lève l'exception $default$.

La description complète des protocoles des composants est présentée dans notre papier [MACM15b].

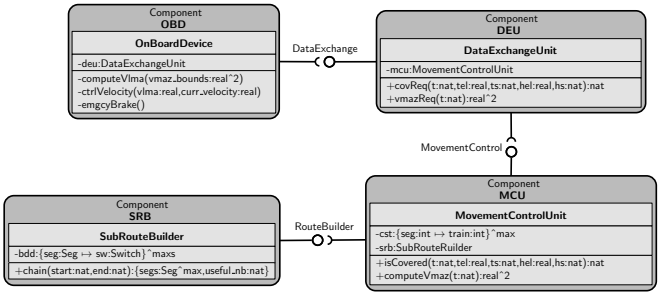
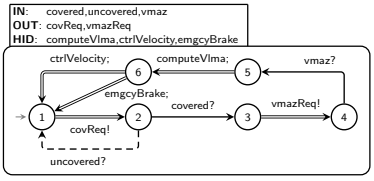
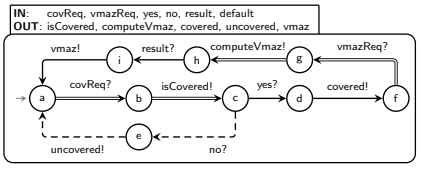


FIGURE 5.3 – Modélisation UML de l'architecture du système.

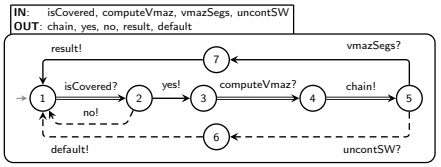
Considérons trois contrats comportementaux B_o, B_d, et B_m, associés respectivement au composants OBD, DEU, et MCU tels que $B_o.\mathcal{A}$ est A_o, $B_d.\mathcal{A}$ est A_d, et $B_m.\mathcal{A}$ est A_m. La table 5.1 montre un exemple de la sémantique de la méthode $covReq$ dans B_o et B_d avec la signature $covReq\ (t, tel, ts, hel, hs) \rightarrow covered \sharp uncovered$ (les types des paramètres sont donnés dans la figure 5.3). Il faut noter que la spécification $B_d.\mathcal{I}(covReq).S$ n'est pas définie ($\perp[t, tel, hel, ts, hs, covered]$) : au niveau de B_d, il n'y a pas de paramètre ou d'attribut ($\Lambda_{A_d}(covReq) = \emptyset$) décrivant comment le paramètre de retour $covered$ est calculé.



(a) A_o (OBD)



(b) A_d (DEU)



(c) A_m (MCU)

FIGURE 5.4 – Automates d'interface des composants OBD, DEU, and MCU : actions de méthodes (transitions doubles) ; action de retour (transitions simples) ; actions des exception (transitions en pointillés).

Sémantique de sortie $B_o.\mathcal{O}$	Sémantique d'entrée $B_d.\mathcal{I}$
$B_o.\mathcal{O}(covReq).P \equiv t \in \{0,...,30\} \wedge ts, hs \in \{0,...,500\} \wedge$ $tel, hel \in [0,5000] \wedge tel < hel$	$B_d.\mathcal{I}(covReq).P \equiv t \in \{0,...,30\} \wedge ts, hs \in \{0,...,500\} \wedge$ $tel, hel \in [0,5000]$
	$B_d.\mathcal{I}(covReq).S \equiv \perp[t, tel, hel, ts, hs, covered]$
$B_o.\mathcal{O}(covReq).Q \equiv covered \in \{0,1,2\}$	$B_d.\mathcal{I}(covReq).Q \equiv covered \in \{1,2\}$
	$B_d.\mathcal{I}(covReq).E \equiv covered = 0$

TABLE 5.1 – Sémantique de l'action de méthode $covReq$.

5.5/ COMPATIBILITÉ DES COMPOSANTS SPÉCIFIÉS AVEC LEURS CONTRATS COMPORTEMENTAUX

Dans cette section, nous présentons notre démarche permettant de décider de la compatibilité de deux composants spécifiés avec leur contrats comportementaux. Une fois la compatibilité est vérifiée, leur assemblage est possible.

La composition de deux contrats comportementaux peut entraîner des situations de blocage causées par des incompatibilités aux niveaux sémantique ou protocole. Au niveau sémantique la synchronisation des actions partagées de méthodes d'entrée/sortie avec des sémantiques incompatibles, conduit à des états de blocage. Au niveau du protocole la composition de deux automates d'interface peut contenir des états de blocage dans lesquels un composant demande une entrée (appelle une méthode d'un autre composant) non acceptée par l'autre. Par exemple, un composant appelle une méthode levant des exceptions sans pouvoir les gérer.

5.5.1/ Synchronisation des automates d'interface et compatibilité sémantique

Dans cette section nous présentons quelques définitions formelles nécessaires à la définition de la composition de deux composants décrits avec leurs contrats comportementaux. C'est cette dernière qui nous permet de décider de leur compatibilité.

Définition 5.6 (Composabilité des contrats comportementaux)**.** *Soient deux contrats comportementaux B_1 et B_2 tels que $B_1.\mathcal{A} = A_1$ et $B_2.\mathcal{A} = A_2$, B_1 et B_2 sont* composables *si A_1 et A_2 sont composables, et chaque action $a \in Shared(A_1, A_2) \cap \Sigma^{\mathrm{m}}_{A_1}$ a la même signature dans A_1 et A_2.*

Dans la définition suivante nous présentons les conditions pour la compatibilité sémantique entre les actions partagées d'entrée/sortie de A_1 et A_2.

Définition 5.7 (Compatibilité sémantique des actions partagées)**.** *Soit $a \in Shared(A_1, A_2) \cap \Sigma^{\mathrm{m}}_{A_1}$, pour tout $(i, o) \in \Psi^{\mathrm{i}}_{A_1}(a) \times \Psi^{\mathrm{o}}_{A_1}(a)$, a dans B_1 est* compatible sémantiquement *avec a dans B_2, notée $SemComp_a(B_1, B_2) = vrai$ (faux dans le cas contraire), si l'une des conditions suivantes est vraie :*

— $B_1.\mathcal{O}(a).P[i] \Rightarrow B_2.\mathcal{I}(a).P[i] \wedge$
$B_1.\mathcal{O}(a).Q[i, o] \Leftarrow B_2.\mathcal{I}(a).Q[i, o]$, *si* $a \in \Sigma^{\mathrm{Om}}_{A_1}$ *;*

— $B_1.\mathcal{I}(a).P[i] \Leftarrow B_2.\mathcal{O}(a).P[i] \wedge$
$B_1.\mathcal{I}(a).Q[i, o] \Rightarrow B_2.\mathcal{O}(a).Q[i, o]$, *si* $a \in \Sigma^{\mathrm{Im}}_{A_1}$.

Ces conditions sont définies sur les paramètres d'entrée/sortie des méthodes. Le sens des implications entre les pré et post conditions des méthodes dépend de leur appartenance à un composant appelant (méthode *output*) ou appelé (méthode *input*).

Exemple 5.2. *Dans notre étude de cas (section 5.4.1), $B_o.\mathcal{A}$ et $B_d.\mathcal{A}$ sont composables. L'ensemble $Shared(A_o, A_d)$ est défini par $\{covReq, vmazReq, covered, uncovered, vmaz\}$. D'après la table 5.1, $SemComp_a(B_o, B_d)$ est vraie pour $a = covReq$.*

Définition 5.8 (Contrat comportemental synchrone)**.** *Soient B_1 et B_2 deux contrats comportementaux composables, nous définissons par $B_1|B_2$, le contrat comportemental* synchrone *de B_1 de B_2 tel que $(B_1|B_2).\mathcal{A}$ est $A_1 \otimes A_2$ restreint à l'ensemble des états atteignables, $(B_1|B_2).\mathcal{I}$ est défini par $B_i.\mathcal{I}(a)$ pour tout $a \in \Sigma^{\mathrm{Im}}_{A_i} \setminus Shared(A_1, A_2)$, et $(B_1|B_2).\mathcal{O}$ est défini par $B_i.\mathcal{O}(a)$ pour tout $a \in \Sigma^{\mathrm{Om}}_{A_i} \setminus Shared(A_1, A_2)$ tels que $i \in \{1, 2\}$. Pour simplifier nous notons $(B_1|B_2).\mathcal{A}$ par A_{12}.*

Les états de blocage dans A_{12} représentent les blocages possibles durant la communication entre les composants spécifiés par B_1 et B_2 aux niveaux protocole et sémantique. C'est les états $s_1 s_2$ tels que :

— il existe au moins $a \in Shared(A_1, A_2)$ activable de s_1 et non de s_2 ou inversement, ou

— a est une action de méthode activable de s_1 et aussi de s_2, par contre la condition $SemComp_a(B_1, B_2)$ n'est pas valide.

Définition 5.9. *L'ensemble des* états de blocage *$Dead(A_1, A_2)$ dans A_{12} est $\{s_1 s_2 \in S_{A_{12}} \mid (\exists\, a \in Shared(A_1, A_2).\, D_1(s_1 s_2) \vee D_2(s_1 s_2))\}$ tel que :*

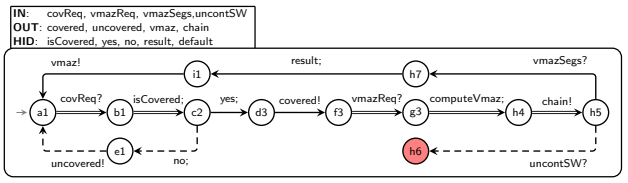


FIGURE 5.5 – Automate d'interfacon $(B_d|B_m).\mathcal{A}$.

— $D_1(s_1 s_2) \equiv (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \vee$
$(a \in \Sigma^{\mathrm{Om}}_{A_1}(s_1) \wedge a \in \Sigma^{\mathrm{Im}}_{A_2}(s_2) \wedge \neg SemComp_a(B_1, B_2))$;

— $D_2(s_1 s_2) \equiv (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \vee$
$(a \in \Sigma^{\mathrm{Om}}_{A_2}(s_2) \wedge a \in \Sigma^{\mathrm{Im}}_{A_1}(s_1) \wedge \neg SemComp_a(B_1, B_2))$.

Exemple 5.3. *Les automates d'interface A_d et A_m sont composables (voir la figure 5.4). Considérons deux contrats comportementaux composables B_d et B_m tels que $B_d.\mathcal{A} = A_d$ et $B_m.\mathcal{A} = A_m$. Supposons que les actions $isCovered$ et $computeVmaz$ sont sémantiquement compatibles entre B_d et B_m, donc l'état $h6$ est le seul état de blocage dans $(B_d|B_m).\mathcal{A}$: car l'action d'exception $default \in \Sigma^{\mathrm{e}}_{A_m}(6) \cap Shared(A_d, A_m)$ n'est pas activable depuis l'état h dans A_d (voir la figure 5.5).*

5.5.2/ L'APPROCHE OPTIMISTE POUR L'ASSEMBLAGE DE COMPOSANTS

Dans notre approche pour l'assemblage de composants, nous adoptons la conception ascendante incrémentale, ce qui signifie que le système est construit incrémentalement en commençant par l'assemblage de deux composants élémentaires en vérifiant leur compatibilité, ensuite le composite obtenu sera assemblé avec un troisième composant, et ainsi de suite jusqu'a l'obtention du système en entier. Ce choix de conception est inspiré de l'approche optimiste de la composition des automates d'interface, qui est en adéquation avec la conception incrémentale des SBC.

Compatibilité des contrats comportementaux : Dans l'automate produit A_{12}, tous les états $s_1 s_2$ depuis lesquels des états de blocage sont atteignables d'une manière autonome (en activant uniquement des transitions avec des actions de sortie ou internes), sont considérés incompatibles et doivent être supprimés dans A_{12}. Aucun environnement ne peut empêcher l'atteignabilité des blocages à partir de ces états. Un état $s_1 s_2 \in S_{A_{12}}$ est *compatible* dans A_{12} si il n'y a pas d'état $d_1 d_2 \in Dead(A_1, A_2)$ atteignable d'une manière autonome depuis $s_1 s_2$. Nous notons par $Cmp(A_1, A_2)$, l'ensemble des états compatibles dans A_{12}, et atteignable après la suppression des états incompatibles dans A_{12}. **Les contrats comportementaux B_1 et B_2 sont *compatibles* ssi ils sont composables et $\iota_{A_{12}} \in Cmp(A_1, A_2)$ (leur composition n'est pas vide)**. L'automate d'interface de la composition de deux contrats comportementaux est restreint aux états compatibles de leur produit synchrone

Définition 5.10 (La composition des contrats comportementaux)**.** *La* composition $B_1 \| B_2$ *de B_1 et B_2 est un contrat comportemental tel que :* $(B_1 \| B_2).\mathcal{I} = (B_1|B_2).\mathcal{I}$, $(B_1 \| B_2).\mathcal{O} = (B_1|B_2).\mathcal{O}$, *et* $(B_1 \| B_2).\mathcal{A}$ *est un automate d'interface tel que* $S_{(B_1 \| B_2).\mathcal{A}}$ *est restreint à* $Cmp(A_1, A_2)$, $\iota_{(B_1 \| B_2).\mathcal{A}} = \iota_{A_{12}}$, $\Sigma^*_{(B_1 \| B_2).\mathcal{A}} = \Sigma^*_{A_{12}}$ *pour* $* \in \{\mathrm{I}, \mathrm{O}, \mathrm{H}\}$, *et* $\delta_{(B_1 \| B_2).\mathcal{A}} = \{(s, a, t) \in \delta_{A_{12}} \mid s, t \in Cmp(A_1, A_2)\}$.

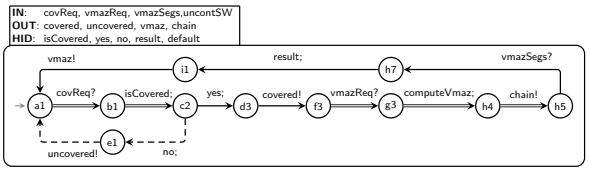


FIGURE 5.6 – Automate d'interface $(B_d \| B_m).\mathcal{A}$.

Exemple 5.4. *L'automate d'interface $(B_d \| B_m).\mathcal{A}$ est la restriction de $(B_d | B_m).\mathcal{A}$ à l'ensemble $S_{(B_d|B_m).\mathcal{A}} \setminus \{h6\}$ des états compatibles (présenté dans figure 5.6).*

Les preuves des théorèmes suivants, dans le reste du chapitre, sont détaillées dans [MACM15a]. Le théorème suivant indique la préservation de la propriété bien formé des automates d'interface par la composition des contrats comportementaux.

Théorème 5.1. *Soient B_1 et B_2 deux contrats comportementaux avec leurs automates d'interface A_1 et A_2. Si A_1 et A_2 sont bien formés et B_1 est compatible avec B_2, alors $(B_1 \| B_2).\mathcal{A}$ est bien formé.*

Le théorème suivant est au cœur de la conception incrémentale de systèmes à base de composants orientés objet. C'est une généralisation directe de l'associativité de la composition des automates d'interface [dAH01] aux contrats comportementaux.

Théorème 5.2. *L'opération de composition $\|$ entre des contrats comportementaux est commutative et associative.*

L'algorithme pour la vérification de la compatibilité entre deux contrats comportementaux est similaire à celui décrit dans [dAH01] pour les automates d'interface, en considérant en plus la couche sémantique des actions.

Implémentation : L'outil implémentant partiellement l'approche proposée a été développé dans le cadre de la thèse de Sebti Mouelhi [Mou11]. Il permet de construire la composition des automates d'interface enrichis avec la sémantique des actions, vérifiant ainsi leur compatibilité. La vérification de la compatibilité sémantique est effectuée en utilisant le solveur automatique de satisfiabilité CVC3 [BT07].

5.6/ RAFFINEMENT

Dans cette section nous présentons nos travaux concernant le raffinement des composants décrits par leurs contrats comportementaux.

Le raffinement introduit plus de détails dans une spécification abstraite d'un composant pour avoir une version plus concrète. Il garantit une substituabilité sûre d'une version abstraite d'un composant par sa version raffinée. Nous proposons une approche de raffinement pour les contrats comportementaux aux niveaux protocole et sémantique, adaptée au contexte orienté objet. Nous commençons par l'introduction du raffinement au niveau des automates d'interface spécifiant les protocoles des composants orientés objet.

5.6.1/ SIMULATION D'EXPANSION

La relation de raffinement originelle des automates d'interface est *contravariante* [dAH01] sur les actions de sortie. Autrement dit, une version abstraite d'un automate d'interface A contient au moins les mêmes actions de sortie et au plus les mêmes actions d'entrée, que son raffinement A'. La relation de raffinement est basée sur une relation de simulation alternée [AHKV98] : chaque action de sortie de A' est simulée par au moins la même action dans A, et chaque action d'entrée de A est simulée exactement par la même action dans A'. Au niveau du protocole dans la conception de composant orientés objet, notre relation de raffinement doit garantir qu'une spécification raffinée d'un composant peut :

— fournir plus de méthodes que la spécification abstraite, et

— contenir plus de détails des méthodes fournies comparées à celles du niveau abstrait, sous forme d'appels de méthodes de sortie ou internes, encapsulés dans les corps de ces méthodes.

Ces exigences nous conduisent à considérer la relation de raffinement comme *covariante* sur les actions d'entrée et de sortie, autrement dit, elle préserve le sous-typage pour les deux types d'actions. Donc A' raffine A si A accepte (resp. évoque) moins d'actions d'entrée (resp. de sortie) que A'. Nous proposons alors notre relation de simulation dite d'expansion : chaque entrée, chaque sortie, ou action local de A est simulé dans A' par la même action,suivi ou précédé, par d'autres actions. Pour formaliser cette relation, nous définissons l'ensemble de *fermeture* $Clos_A(s, \Sigma)$ de $s \in S_A$ avec les actions dans $\Sigma \subseteq \Sigma_A$ par le plus grand ensemble $S \subseteq S_A$ tel que $s \in S$ et si $t \in S$, $v = Succ_A(t, a)$, et $a \in \Sigma$, alors $v \in S$. Autrement dit, $Clos_A(s, \Sigma)$ contient des états atteignables à partir de l'état s en activant des actions dans Σ.

Définition 5.11 (Simulation d'expansion)**.** *Soient deux automates d'interface A et A', une relation binaire $\gtrsim \subseteq S_A \times S_{A'}$ est une* simulation d'expansion *de A à A' si pour tout $ss' \in S_A \times S_{A'}$ et $a \in \Sigma_A(s)$ tels que $s \gtrsim s'$ et $t = Succ_A(s, a)$, les conditions suivantes sont vraies :*

(1) si $a \in \Sigma_A^{\mathrm{Om}}(s) \cup \Sigma_A^{\mathrm{Ir}}(s) \cup \Sigma_A^{\mathrm{Ie}}(s)$, alors $a \in \Sigma_{A'}(s')$ et $t \gtrsim t'$ pour $t' = Succ_{A'}(s', a)$;

(2) si $a \in \Sigma_A^{\mathrm{Im}}(s) \cup \Sigma_A^{\mathrm{Hm}}(s)$, alors $a \in \Sigma_{A'}(s')$, et il existe un sous ensemble $\Sigma \subseteq ((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$ et un état $t' \in Clos_{A'}(Succ_A(s', a), \Sigma)$ tel que $t \gtrsim t'$;

(3) si $a \in \Sigma_A^{\mathrm{Or}}(s) \cup \Sigma_A^{\mathrm{Hr}}(s)$, alors il y a $v' \in Clos_{A'}(s', \Sigma)$ tel que $\Sigma = ((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$, $a \in \Sigma_{A'}(v')$, et $t \gtrsim t'$ pour $t' = Succ_{A'}(v', a)$;

(4) si $a \in \Sigma_A^{\mathrm{Oe}}(s) \cup \Sigma_A^{\mathrm{He}}(s)$, alors il y a $v' \in Clos_{A'}(s', \Sigma)$ tel que $\Sigma = ((\Sigma_{A'}^{\mathrm{aut}} \setminus (\Sigma_{A'}^{\mathrm{Oe}} \cup \Sigma_{A'}^{\mathrm{Or}})) \cup \Sigma_{A'}^{\mathrm{Ie}}) \setminus \Sigma_A$, $a \in \Sigma_{A'}(v')$, et $t \gtrsim t'$ pour $t' = Succ_{A'}(v', a)$.

Notre relation de simulation d'expansion indique où les détails de raffinement sont ajoutés dans la spécification abstraite d'un automate d'interface. La condition (1) de la définition 5.11 indique que chaque transition étiquetée par une action de sortie de méthode, ou une action d'entée de retour ou d'exception doit être reliée avec une transition étiquetée par la même action dans A'. Autrement dit, les appels de méthodes destinés à l'environnement, la réception de leurs valeurs de retour, et la levée et la capture de leurs exceptions, ne sont pas modifiés ou détaillés au niveau raffiné.

La condition (2) stipule que chaque transition étiquetée par une action d'entrée ou interne de méthode dans A est reliée dans A' à une transition étiquetée par la même action suivie de zéro ou plusieurs transitions étiquetées par un sous-ensemble de nouvelles actions

de non-exceptions autonomes dans $((\Sigma^{\text{aut}}_{A'} \setminus \Sigma^{\text{Or}}_{A'}) \setminus \Sigma^{\text{e}}_{A'}) \setminus \Sigma_A$. Autrement dit, une méthode publique fournie dans l'abstraction d'un composant peut être raffinée en ajoutant à son corps de nouveaux appels de méthodes privées ou publiques. En outre, étant donné que l'offre de méthodes privées n'est pas spécifiée par des actions dans les automates d'interface (voir la section 5.3), notre relation de simulation permet d'ajouter des détails de raffinement concernant les méthodes privées après leurs appels.

La condition (3) stipule que chaque transition étiquetée par une action de sortie ou interne de retour a dans A est reliée dans A' avec zéro ou plusieurs transitions étiquetées par de nouvelles actions de non-exception autonomes dans $((\Sigma^{\text{aut}}_{A'} \setminus \Sigma^{\text{Or}}_{A'}) \setminus \Sigma^{\text{e}}_{A'}) \setminus \Sigma_A$ suivi d'une transition étiquetée par a. L'événement de retour d'une méthode privée ou publique dans l'abstraction est calculé sur la base des valeurs de retour des nouveaux appels de méthodes privées ou publiques ajoutées comme détails de raffinement.

La condition (4) stipule que chaque transition étiquetée par une action de sortie ou interne d'exception a dans A est reliée dans A' par zéro ou plusieurs transitions étiquetées soit par de nouvelles actions d'exception autonomes et internes dans $(\Sigma^{\text{aut}}_{A'} \setminus (\Sigma^{\text{Oe}}_{A'} \cup \Sigma^{\text{Or}}_{A'})) \setminus \Sigma_A$, ou par de nouvelles actions d'exception d'entrée dans $\Sigma^{\text{Ie}}_{A'} \setminus \Sigma_A$, suivie d'une transition étiquetée par a. Les événements d'exception d'une méthode fournie privée ou publique dans l'abstraction est la propagation des événements d'exception de capture de nouveaux appels privés ou publiques de méthodes ajoutées comme détails dans le raffinement.

Nous définissons la relation de raffinement comme suit.

Définition 5.12. *A' raffine A ($A \geq A'$) ssi (1) $\Sigma^{\text{I}}_A \subseteq \Sigma^{\text{I}}_{A'}$, $\Sigma^{\text{O}}_A \subseteq \Sigma^{\text{O}}_{A'}$, $\Sigma^{\text{H}}_A \subseteq \Sigma^{\text{H}}_{A'}$, et (2) il existe une simulation d'expansion $\gtrsim$ de A à A' telle que $\iota_A \gtrsim \iota_{A'}$.*

Une conséquence triviale de la condition (1) de la définition 5.12 est la covariance de A à A' sur les actions de méthodes, de retour, d'exception : $\Sigma^{\text{m}}_A \subseteq \Sigma^{\text{m}}_{A'}$, $\Sigma^{\text{r}}_A \subseteq \Sigma^{\text{r}}_{A'}$, et $\Sigma^{\text{e}}_A \subseteq \Sigma^{\text{e}}_{A'}$. La condition (2) exige l'existence d'une simulation d'expansion de A à A' reliant leurs états initiaux ι_A et $\iota_{A'}$ et propagée récursivement aux états successeurs.

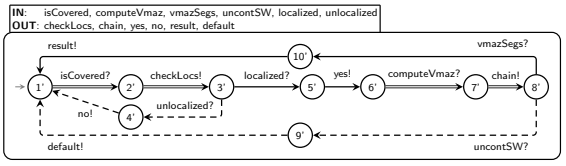


FIGURE 5.7 – L' automate d'interface raffiné de MCU

Exemple 5.5. *Dans la figure 5.7, est présenté l'automate d'interface raffiné du composant MCU. Il faut noter que l'appel de la nouvelle méthode $checkLocs!$ est encapsulé dans les chemins d'exécutions décrivant le corps de la méthode $iscovered$ fournie par MCU. Les conditions de raffinement sont respectées par A'_m et A_m (l'exemple détaillé est présenté dans [MACM15b]).*

5.6.2/ SUBSTITUABILITÉ SÉMANTIQUE

La substituabilité sémantique des actions de méthodes entre une version abstraite et une version concrète d'un contrat comportemental d'un composant est basée sur *les*

principes du sous-typage des comportements introduits dans [Ame91, LW94] : dans la spécification raffinée, une méthode fournie doit avoir une précondition plus faible, une postcondition de terminaison plus forte, et n'introduit pas d'exceptions en fournissant une condition d'exception plus forte que celle du niveau abstrait. Inversement, un appel de méthode doit avoir, dans le raffinement, une précondition plus forte et une postcondition plus faible que l'abstraction.

Considérons deux contrats comportementaux B et B', nous désignons $B.\mathcal{A}$ par A et $B'.\mathcal{A}$ par A'.

Définition 5.13. *Soit une action* $a \in \Sigma_A^{\mathrm{Im}}$, $B'.\mathcal{I}(a) = (P'_a, S'_a, Q'_a, E'_a)$ peut substituer à $B.\mathcal{I}(a) = (P_a, S_a, Q_a, E_a)$, *donc* $SemSub_a(B, B')$ *est valide, si pour tout* $(i, f, o) \in \Psi_A^{\mathrm{i}}(a) \times \Lambda_A(a) \times \Psi_A^{\mathrm{o}}(a)$, *les conditions suivantes sont satisfaites :*

(1) $P_a[i] \Rightarrow P'_a[i]$ *;* $R[i, f, o] \Leftarrow R'[i, f, o]$ *pour* $R \in \{Q_a, E_a\}$ *;*

(2) $P_a[i] \wedge S'_a[i, f, o] \Rightarrow S_a[i, f, o]$.

Soit une action $b \in \Sigma_A^{\mathrm{Om}}$, $B'.\mathcal{O}(b) = (P'_b, Q'_b)$ peux substituer à $B.\mathcal{O}(b) = (P_b, Q_b)$, *donc* $SemSub_b(B, B')$ *est valide, si pour tout* $(i, o) \in \Psi_A^{\mathrm{i}}(b) \times \Psi_A^{\mathrm{o}}(b)$, *les conditions suivantes sont satisfaites :*

(3) $P_b[i] \Leftarrow P'_b[i]$ *;* $Q_b[i, o] \Rightarrow Q'_b[i, o]$.

Enfin, nous pouvons définir le raffinement des contrats comportementaux en fonction du raffinement des automates d'interface et la substituabilité sémantique des actions de méthodes.

Définition 5.14. B' raffine B *(*$B \sqsupseteq B'$*) si* $A \succeq A'$ *et pour tout* $a \in \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Om}}$, $SemSub_a(B, B')$ *est valide.*

5.6.3/ Propriétés du raffinement

Dans cette sous-section, nous présentons les propriétés et les conditions sous lesquelles notre approche de raffinement permet une implémentation indépendante des composants en utilisant leurs contrats comportementaux. Nous rappelons que ces résultats sont prouvés dans [MACM15a].

Réflexivité et transitivité

Théorème 5.3. *La relation de raffinement* $\sqsupseteq$ *entre deux contrats comportementaux est un préordre, donc elle est réflexive et transitive.*

Le théorème précédent stipule qu'un contrat comportemental peut être raffiné progressivement en plusieurs étapes tout en restant cohérent avec sa spécification abstraite.

Implémentabilité indépendante des composants

Le raffinement devrait permettre l'implémentabilité indépendante des composants : les contrats comportementaux compatibles peuvent être raffinés séparément, tout en maintenant leur compatibilité.

Notre approche de raffinement garantit la cohérence entre deux contrats comportementaux B et B' où $B \sqsupseteq B'$ si ils sont considérés comme «isolés» de leur environnement. Cependant, il est très générique et n'empêche pas l'introduction de comportements mal conçus dans leurs automates d'interface. Comme le raffinement peut évoquer de nouvelles actions de sorties, le concepteur devrait le définir "correctement" afin de préserver la compatibilité avec l'environnement. Par exemple, selon la définition 5.3 et les conditions (2) et (3) de la définition 5.11, la relation de simulation d'expansion proposée conserve la propriété de bien-formé dans le raffinement, uniquement pour les événements d'actions de méthodes issues du niveau abstrait. Par contre, il ne garantit pas que les nouveaux événements d'actions de méthodes sont nécessairement suivis de leurs événements de retour.

Donc, pour garantir l'implémentabilité indépendante des composants raffinés, nous définissons les conditions dans lesquelles le raffinement d'un contrat comportemental est considéré comme sûr par rapport à son environnement. Pour formaliser ces exigences, considérons trois contrats comportementaux B_1, B'_1 et B_2 tels que B_1 et B_2 sont compatibles, $B_1 \sqsupseteq B'_1$, et B'_1 est composable avec B_2. Soient $B_1.\mathcal{A} = A_1$, $B'_1.\mathcal{A} = A'_1$, $B_2.\mathcal{A} = A_2$, $(B_1|B_2).\mathcal{A} = A_{12}$, et $(B'_1|B_2).\mathcal{A} = A'_{12}$. Nous définissons par $EnabledRiseDead(A_1, A_2) = \{a \in (\Sigma^{\mathrm{Im}}_{A_{12}} \cup \Sigma^{\mathrm{Ie}}_{A_{12}}) \cap \Sigma_{A_{12}}\langle\sigma\rangle \mid \sigma \in \Theta_{A_{12}}(d_1d_2), d_1d_2 \in Dead(A_1, A_2)\}$: l'ensemble des actions non-autonomes activables par des exécutions $\sigma \in \Theta_{A_{12}}(d_1d_2)$ pour tout $d_1d_2 \in Dead(A_1, A_2)$. Comme B_1 et B_2 sont compatibles alors, $EnabledRiseDead(A_1, A_2) \neq \emptyset$ si $Dead(A_1, A_2) \neq \emptyset$.

Soit $\gtrsim$ une simulation d'expansion de A_1 à A'_1 telle que $\iota_{A_1} \gtrsim \iota_{A'_1}$, B'_1 est un *raffinement sûr* de B_1 par rapport à B_2, noté $B_1 \sqsupseteq^s_{B_2} B'_1$, si les conditions suivantes sont satisfaites sur A_1, A'_1, et A_2 :

(1) pour tout état de blocage $d'_1d_2 \in Dead(A'_1, A_2)$, il existe un état de blocage $d_1d_2 \in Dead(A_1, A_2)$ tel que $d_1 \gtrsim d'_1$ ou $d'_1 \in Clos_{A'_1}(c'_1, \Sigma_{A'_1} \setminus \Sigma_{A_1})$ and $d_1 \gtrsim c'_1$, et

(2) $Shared(A'_1, A_2) \cap EnabledRiseDead(A_1, A_2) = \emptyset$.

La condition (1) indique que A'_{12} n'introduit pas de nouveaux blocages par rapport à A_{12} en garantissant que tous les états dans $Dead(A'_1, A_2)$ sont simulés par des états dans $Dead(A_1, A_2)$. La condition (2) stipule que A'_1 ne partage pas les actions non-autonomes dans $EnabledRiseDead(A_1, A_2)$ avec A_2, car leur activation par l'environnement de A_{12}, conduit inévitablement à des états de blocage.

Théorème 5.4. *Si $B_1 \sqsupseteq^s_{B_2} B'_1$, alors B'_1 est compatible avec B_2 et $B_1 \| B_2 \sqsupseteq B'_1 \| B_2$.*

Étant donné un quatrième contrat comportemental B'_2 tels que B'_2 est composable avec B'_1 et $B_2 \sqsupseteq B'_2$, la propriété d'implémentabilité indépendante des contrats comportementaux est établie par le corollaire suivant, ce qui est évidemment déductible des théorèmes 5.3 et 5.4.

Corollaire 5.1. *Si $B_1 \sqsupseteq^s_{B_2} B'_1$ et $B_2 \sqsupseteq^s_{B'_1} B'_2$, alors B'_1 est compatible avec B'_2 et $B_1 \| B_2 \sqsupseteq B'_1 \| B'_2$.*

5.7/ ADAPTATION DES COMPOSANTS

Dans cette section, nous présentons nos contributions [CMM12b, CMM10b] concernant l'adaptation des composants spécifiés avec des contrats comportementaux. Nous proposons donc une approche d'adaptation qui prend en considération les niveaux signature,

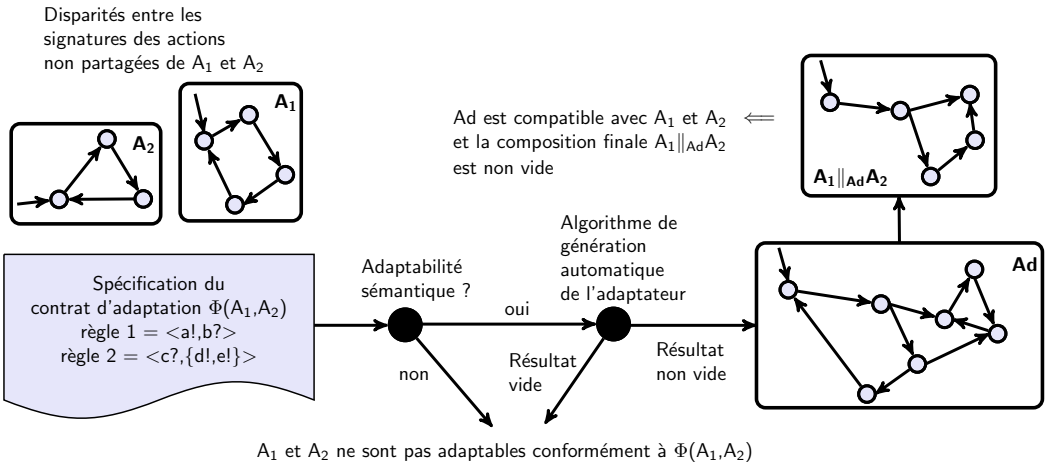


FIGURE 5.8 – Processus d'adaptation de deux automates d'interface composables A_1 et A_2.

sémantique, et protocole des composants. Alors que, la plupart des travaux existants se focalisent sur la résolution des problèmes de l'adaptation uniquement aux niveaux signature et protocole (voir le chapitre 3). Et contrairement à notre travail présenté dans le chapitre 3, le langage SysML n'est pas pris en considération, et nous traitons uniquement les inadéquations entre deux composants en interaction, sans considération des composants parents, comme c'est le cas dans l'approche présentée dans le chapitre 3. L'originalité de ce travail est la proposition d'une solution d'adaptation pour l'approche optimiste de composition exploitant les contrats comportementaux.

5.7.1/ APERÇU DE L'APPROCHE

Note approche est destinée à la résolution des inadéquations de comportements entre deux composants exploitant l'approche optimiste d'assemblage de composants spécifiés par des contrats comportementaux. Elle est incrémentale comme celle présentée dans [BBC05], qui permet la construction des systèmes par assemblage incrémental des composants deux par deux. Deux composants sont assemblés et un adaptateur est généré automatiquement en cas d'inadéquation conformément à un contrat d'adaptation (*mapping*). Ce contrat est défini par le concepteur en se basant sur les rôles des actions inadéquates décrites par leurs signatures et leurs sémantiques dans les contrats comportementaux des deux composants. Contrairement à d'autres approches d'adaptation [CPS06c, CPS08b], les inadéquations entre les actions non partagées ne peuvent pas être détectées en calculant le produit synchrone de deux automates d'interface composables. En effet, les actions inadéquates qui devraient être synchronisées par l'adaptation sont nommées différemment, ce qui conduit à les rendre absentes dans l'ensemble des actions partagées. D'où l'intérêt de notre approche pour apporter une solution d'adaptation applicable dans ce contexte.

Notre objectif est donc, de générer automatiquement un adaptateur pour deux composants spécifiés par leurs contrats comportementaux B_1 et B_2 tels que $B_1.\mathcal{A} = A_1$, $B_2.\mathcal{A} = A_2$. Ainsi nous proposons de générer un adaptateur Ad pour A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$. Notre approche est illustrée dans la figure 5.8, et ses

étapes décrites ci-dessous :

1. Si A_1 et A_2 sont composables, vérifier l'adaptabilité sémantique entre les actions reliées par le contrat d'adaptation. Si ces actions sont sémantiquement adaptables, alors on passe à la deuxième étape. Sinon, A_1 et A_2 ne peuvent pas être adaptés conformément à $\Phi(A_1, A_2)$ (détails dans la section 5.7.4) ;

2. Appliquer l'algorithme de génération automatique d'adaptateur (détails dans la section 5.8) sur A_1 et A_2 et le contrat d'adaptation $\Phi(A_1, A_2)$. Dans le cas où A_1 et A_2 sont adaptables, l'algorithme génère un adaptateur non vide Ad pour A_1 et A_2 conformément à $\Phi(A_1, A_2)$. Dans ce cas, l'automate Ad permet de consommer les sorties de l'un des deux automates avant de générer les sorties pour leurs entrées correspondantes dans l'autre automate. L'automate d'interface du composite final est défini par le produit $A_1 \circledast_{Ad} A_2$. Dans le cas où l'algorithme retourne un automate vide, les deux automates d'interface A_1 et A_2 ne sont pas adaptables conformément à $\Phi(A_1, A_2)$.

5.7.2/ ÉTAT DE L'ART ET CONTRIBUTIONS

Cette section présente les principaux travaux sur l'adaptation proches de notre travail, en complément des travaux présentés dans le chapitre 3.

La plupart des propositions d'adaptation portent sur la résolution des inadéquations comportementales entre les descriptions abstraites de composants logiciels [MPS12, TI08, BBC05, VW01, CPS06c, CPS08b]. Par exemple, dans [TI08], les auteurs présentent une approche exploitant les systèmes de transition, modélisant les protocoles des composants, pour générer un composant adaptateur. Dans [VW01], les auteurs ont développé l'outil PaCoSuite pour modifier visuellement les composants et générer des adaptateurs, en utilisant les signatures des méthodes et les protocoles. Les approches proposées dans [CPS06c, CPS08b] sont basées sur une procédure non incrémentale de composition et d'adaptation. Plusieurs composants peuvent être adaptés en même temps de manière non incrémentale en supposant que l'environnement de chaque composant est préfixé auparavant.

Dans [MLS06], les auteurs ont proposé un modèle d'adaptateurs utilisant la méthode B permettant de définir l'interopérabilité entre les composants. Dans [SEG08], les auteurs proposent un état de l'art sur les approches d'adaptation semi-automatiques et automatiques des composants et services. Dans [DBMN08], les auteurs discutent la notion de la compatibilité au niveau protocole des services Web et proposent une approche d'adaptation pour les services incompatibles. Leur proposition traite des notions comme la substituabilité et la contrôlabilité. Dans [Pad09], l'auteur propose un langage formel pour décrire les contrats des services Web. Le travail fournit une sémantique pour définir les règles d'adaptation entre les services en terme de relation d'équivalence. Dans [MPM08], les auteurs proposent une approche d'adaptation entre les services Web qui intègre les trois niveaux d'interopérabilité des composants : signature, sémantique,et protocole. Le niveau sémantique est décrit par une notation SAWSDL [1] complémentée par une notation BPEL.

1. http://www.w3.org/TR/sawsdl/

Contributions : L'apport de notre approche comparée aux travaux cités, est la proposition d'une solution d'adaptation, aux niveaux signature, sémantique, et protocole, pour les composants spécifiés avec des contrats comportementaux et exploitant l'approche optimiste des automates d'interface.

5.7.3/ CONTRAT D'ADAPTATION

Soient C_1 et C_2 deux composants spécifiés par leurs contrats comportementaux respectifs B_1 et B_2 avec $B_1.\mathcal{A} = A_1$ et $B_2.\mathcal{A} = A_2$. Ces deux composants présentent des comportements inadéquats et sont reliés par un contrat d'adaptation. Le contrat d'adaptation de leurs deux automates d'interface composables A_1 et A_2 est la mise en œuvre d'une spécification minimale des exigences de l'adaptation de A_1 et A_2. Il décrit de manière explicite les interactions entre les deux automates d'interface A_1 et A_2 grâce à un ensemble de *règles*. Ces règles établissent les relations entre les actions non partagées de A_1 et A_2 qui sont équivalentes du point de vue du concepteur.

Définition 5.15 (*Contrat d'adaptation*)**.** *Soient deux automates d'interface composables A_1 et A_2, une* règle d'adaptation *α est un tuple $\langle L_1, L_2\rangle \in (2^{\Sigma^{O}_{A_1}} \times 2^{\Sigma^{I}_{A_2}}) \cup (2^{\Sigma^{I}_{A_1}} \times 2^{\Sigma^{O}_{A_2}})$ tel que $(L_1 \cup L_2) \cap Shared(A_1, A_2) = \emptyset$, $|L_1|$[2] ≥ 1 et $|L_2| = 1$, ou $|L_1| = 1$ et $|L_2| \geq 1$. Un* contrat d'adaptation *$\Phi(A_1, A_2)$ de A_1 et A_2 est un ensemble non vide de règles α_i pour $1 \leq i \leq |\Phi(A_1, A_2)|$.*

Nous dénotons, par *ActIncomp*$(\Phi(A_1, A_2))$, l'ensemble $\{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} \mid (\exists\, \alpha \in \Phi(A_1, A_2) \mid a \in \Pi_1(\alpha) \vee a \in \Pi_2(\alpha))\}$[3], tel que $\Sigma^{ext}_{A_i} = \Sigma^{I}_{A_i} \cup \Sigma^{O}_{A_i}$. D'après la définition précédente, une règle d'adaptation autorise des correspondances « une-pour-une », « une-pour-plusieurs » et « plusieurs-pour-une » entre les actions. L'adaptation fait correspondre une ou plusieurs actions d'un automate avec une seule action de l'autre.

5.7.4/ ADAPTABILITÉ SÉMANTIQUE

L'adaptabilité sémantique entre les actions reliées par un contrat d'adaptation de deux automates d'interface composables doit être vérifiée avant la génération de l'automate adaptateur. Ces actions doivent être cohérentes au niveau sémantique et dans le cas où l'adaptabilité sémantique n'est pas vérifiée, la génération d'un adaptateur entre les composants reliés par le contrat d'adaptation est impossible.

5.7.4.1/ COHÉRENCE AU NIVEAU DES PARAMÈTRES DES ACTIONS

Soient deux contrats comportementaux B_1 et B_2, leurs automates d'interface respectifs A_1 et A_2, et un contrat d'adaptation $\Phi(A_1, A_2)$. Nous rappelons que $\Psi^{i}_{A}(a)$ représente l'ensemble des paramètres d'entrée d'une action *a*, et $\Psi^{o}_{A}(a)$ celui de ses paramètres de sortie. Pour effectuer la vérification d'adaptabilité sémantique entre A_1 et A_2, il est nécessaire de vérifier les conditions suivantes pour chaque règle $\alpha = \langle L_1, L_2\rangle \in \Phi(A_1, A_2)$:

1. $\sum_{a \in L_1} |\Psi^{i}_{A_1}(a)| = \sum_{b \in L_2} |\Psi^{i}_{A_2}(b)|$ et $\sum_{a \in L_1} |\Psi^{o}_{A_1}(a)| = \sum_{b \in L_2} |\Psi^{o}_{A_2}(b)|$;

2. $|E|$ est la cardinalité d'un ensemble *E*.
3. Soit un tuple $\langle a, b\rangle$, $\Pi_1(\langle a, b\rangle) = a$ et $\Pi_2(\langle a, b\rangle) = b$.

2. Si $|L_1| = 1$ et $|L_2| \geq 1$, $L_1 = \{a\}$, $L_2 = \{b_1, ..., b_{|L_2|}\}$ et $\Psi^{o}_{A_1}(a) = \{o_a\}$, alors il existe exactement une action $b_k \in L_2$ ($1 \leq k \leq |L_2|$) telle que $\Psi^{o}_{A_2}(b_k) = \{o_{b_k}\}$, $\Psi^{o}_{A_2}(b_l) = \emptyset$ pour $1 \leq l \leq |L_2|$ et $l \neq k$ et les deux paramètres de sortie o_a et o_{b_k} satisfont la condition de sous-typage des paramètres :

— si $L_1 \subseteq \Sigma^{O}_{A_1}$, alors $D_{o_a} \subseteq D_{o_{b_k}}$,

— sinon $D_{o_a} \supseteq D_{o_{b_k}}$.

Le tuple θ_α représente le couple (a,b_k). Si $\Psi^{o}_{A_1}(a) = \emptyset$, ($a$,$b_k$) n'est pas défini.

3. La condition est analogue à la précédente avec $|L_1| \geq 1$, $|L_2| = 1$, $L_1 = \{a_1, ..., a_{|L_1|}\}$ et $L_2 = \{b\}$.

4. Dans le cas où $\bigcup_{a \in L_1} \Psi^{i}_{A_1}(a)$ et $\bigcup_{b \in L_2} \Psi^{i}_{A_2}(b)$ ne sont pas vides, il existe une fonction non vide $\varphi^{i}_{\alpha} : \bigcup_{a \in L_1} \Psi^{i}_{A_1}(a) \rightarrow \bigcup_{b \in L_2} \Psi^{i}_{A_2}(b)$ qui associe chaque paramètre d'entrée p d'une action dans L_1 avec un paramètre d'entrée q d'une action dans L_2. La fonction φ^{i}_{α} doit satisfaire la condition de sous-typage des paramètres :

— si $L_1 \subseteq \Sigma^{O}_{A_1}$, alors $D_p \subseteq D_{\varphi^{i}_{\alpha}(p)}$ où $p \in \bigcup_{a \in L_1} \Psi^{i}_{A_1}(a)$,

— sinon $D_{\varphi^{i}_{\alpha}(p)} \subseteq D_p$ où $p \in \bigcup_{a \in L_1} \Psi^{i}_{A_1}(a)$.

L'intérêt de ces conditions est d'établir une cohérence entre les paramètres des actions dans L_1 et L_2 afin de pouvoir vérifier la compatibilité entre les pré et post-conditions des actions dans l'étape de la vérification de l'adaptabilité sémantique.

La première condition établit que le nombre des paramètres d'entrée (resp. les paramètres de sortie) des actions dans L_1 est égal au nombre des paramètres d'entrée (resp. les paramètres de sortie) des actions dans L_2. Les conditions 2 et 3 établissent la relation entre les paramètres de sortie des actions dans L_1 et les paramètres de sortie de l'action b_k appartenant à L_2. Nous assumons que les autres actions dans $L_2 \setminus \{b_k\}$ n'ont pas de paramètres de sortie. La quatrième condition établit les relations entre les paramètres d'entrée des actions dans L_1 et L_2 grâce à la fonction φ^{i}_{α}.

5.7.4.2/ DÉFINITIONS

Avant de définir l'adaptabilité sémantique entre les actions reliées par le contrat d'adaptation de deux automates d'interface, il est indispensable que leurs noms de paramètres soient les mêmes dans leurs préconditions et postconditions. Donc il est nécessaire de renommer les paramètres d'entrée et de sortie dans les sémantiques des actions dans chaque règle $\alpha \in \Phi(A_1, A_2)$.

Nous notons par ($B_1.\mathcal{X}(a).P_\alpha$, $B_1.\mathcal{X}(a).Q_\alpha$) et ($B_2.\mathcal{X}(b).P_\alpha$, $B_2.\mathcal{X}(b).Q_\alpha$), tels que $\mathcal{X} = O$ pour les actions de sortie et prend la valeur $\mathcal{I}$ pour les actions d'entrée, les pré et post-conditions de a dans $\Pi_1(\alpha)$ et celles de b dans $\Pi_2(\alpha)$, après le renommage.

Définition 5.16. *Soient deux contrats comportementaux B_1 et B_2, leurs deux automates d'interface composables A_1 et A_2, et un contrat d'adaptation $\Phi(A_1, A_2)$, tels que les conditions 1,2,3 et 4 établies dans la section 5.7.4.1 sont satisfaites. L'*adaptabilité sémantique *SemAdap$_\alpha(A_1, A_2)$ d'une règle α dans $\Phi(A_1, A_2)$ est satisfaite si les conditions suivantes sont valides :*

1. *si* $\Pi_1(\alpha) \subseteq \Sigma^{O}_{A_1}$, *alors*

$$\bigwedge_{a\in\Pi_1(\alpha)} B_1.\mathcal{O}(a).P_\alpha \Rightarrow \bigwedge_{b\in\Pi_2(\alpha)} B_2.\mathcal{I}(b).P_\alpha$$
$$\wedge$$
$$\bigwedge_{a\in\Pi_1(\alpha)} B_1.\mathcal{O}(a).Q_\alpha \Leftarrow \bigwedge_{b\in\Pi_2(\alpha)} B_2.\mathcal{I}(b).Q_\alpha$$

2. *si* $\Pi_1(\alpha) \subseteq \Sigma^{I}_{A_1}$, *alors la condition est analogue à la première en inversant les implications.*

Définition 5.17 (*Adaptabilité sémantique*)**.** A_1 *et* A_2 *sont* sémantiquement adaptables *conformément au contrat d'adaptation* $\Phi(A_1, A_2)$ *si SemAdap*$_\alpha(A_1, A_2) \equiv$ *vrai pour toute règle* $\alpha \in \Phi(A_1, A_2)$.

5.7.5/ Spécification de l'adaptateur

Après l'étape de vérification de l'adaptabilité sémantique entre deux contrats comportementaux, nous présentons dans cette section la spécification du contrat comportemental de l'adaptateur de deux composants décrits par leurs contrats comportementaux. Il est spécifié par son contrat comportemental B_{Ad}, qui est défini par son automate d'interface Ad et la sémantique de ses actions. Ce contrat est calculé à partir des contrats comportementaux B_1 (avec son automate A_1) et B_2 (avec son automate A_2) spécifiant respectivement les deux composants à adapter. Donc l'automate d'interface adaptateur Ad des deux automates d'interface A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$, est un automate d'interface qui leur permet d'interagir malgré les inadéquations entre leurs actions non partagées. L'adaptateur doit être composable avec A_1 et A_2 et doit respecter un ensemble de contraintes imposées par le contrat d'adaptation.

Définition 5.18 (*Adaptateur*)**.** *Soient deux contrats comportementaux* B_1 *et* B_2*, leurs automates d'interface* A_1 *et* A_2*, composables, et un contrat d'adaptation* $\Phi(A_1, A_2)$*. L'automate* adaptateur *de* A_1 *et* A_2 *conformément à* $\Phi(A_1, A_2)$ *est l'automate d'interface non vide* Ad *faisant partie du contrat comportemental* $B_{Ad} = (Ad, \mathcal{I}, \mathcal{O})$, $Ad = \langle S_{Ad}, \iota_{Ad}, \Sigma^{I}_{Ad}, \Sigma^{O}_{Ad}, \Sigma^{H}_{Ad}, \delta_{Ad} \rangle$ *tel que* S_{Ad} *est l'ensemble des états,* ι_{Ad} *est son état initial, et :*

1. $\Sigma^{I}_{Ad} = \{a \mid a \in$ *ActIncomp*$(\Phi(A_1, A_2)) \cap (\Sigma^{O}_{A_1} \cup \Sigma^{O}_{A_2})\}$;
2. $\Sigma^{O}_{Ad} = \{a \mid a \in$ *ActIncomp*$(\Phi(A_1, A_2)) \cap (\Sigma^{I}_{A_1} \cup \Sigma^{I}_{A_2})\}$;
3. $\Sigma^{H}_{Ad} = \{\epsilon_a \mid a \in (\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus$ *ActIncomp*$(\Phi(A_1, A_2))\}$. *Pour chaque action* a *qui n'appartient pas ActIncomp*$(\Phi(A_1, A_2))$*, l'action interne* ϵ_a *symbolise l'événement non opérationnel associé à* a ;
4. $Shared(Ad, A_1) = \bigcup_{\alpha\in\Phi(A_1,A_2)} \Pi_1(\alpha)$;
5. $Shared(Ad, A_2) = \bigcup_{\alpha\in\Phi(A_1,A_2)} \Pi_2(\alpha)$;
6. *pour toute* $a \in \Sigma^{I}_{Ad}$, $B_{Ad}.\mathcal{I}(a).P = B_1.\mathcal{O}(a).P$ *et* $B_{Ad}.\mathcal{I}(a).Q = B_1.\mathcal{O}(a).Q$ *si* $a \in \Sigma^{O}_{A_1}$*, sinon,* $B_{Ad}.\mathcal{I}(a).P = B_2.\mathcal{O}(a).P$ *et* $B_{Ad}.\mathcal{I}(a).Q = B_2.\mathcal{O}(a).Q$;
7. *pour toute* $a \in \Sigma^{O}_{Ad}$, $B_{Ad}.\mathcal{O}(a).P = B_1.\mathcal{I}(a).P$ *et* $B_{Ad}.\mathcal{O}(a).Q = B_1.\mathcal{I}(a).Q$ *si* $a \in \Sigma^{I}_{A_1}$*, sinon,* $B_{Ad}.\mathcal{O}(a).P = B_2.\mathcal{I}(a).P$ *et* $B_{Ad}.\mathcal{O}(a).Q = B_2.\mathcal{I}(a).Q$

Les conditions 1 et 2 de la définition exigent que pour toute action $a \in$ *ActIncomp*$(\Phi(A_1, A_2))$, si a appartient à $\Sigma^{O}_{A_1}$ ou $\Sigma^{O}_{A_2}$ alors a appartient à Σ^{I}_{Ad}, sinon si a appartient à $\Sigma^{I}_{A_1}$ ou

$\Sigma^{\mathrm{I}}_{A_2}$ alors a appartient à Σ^{O}_{Ad}. La condition 3 exige qu'une action interne non opérationnelle ϵ_a est associée pour chaque action a dans l'ensemble $(\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus$ *ActIncomp*$(\Phi(A_1, A_2))$. L'algorithme de génération automatique de l'adaptateur remplace chaque transition atteignable étiquetée par une action a de l'ensemble $(\Sigma_{A_1} \cup \Sigma_{A_2}) \setminus$ *ActIncomp*$(\Phi(A_1, A_2))$ dans A_1 ou A_2 par une transition étiquetée par une action ϵ_a. Une action ϵ_a synchronise avec l'action a correspondante dans le produit final entre les deux automates et l'adaptateur.

Les conditions 6 et 7 exigent que la sémantique d'une action externe a dans $\Sigma^{\mathrm{ext}}_{Ad}$ est la même que celle de sa correspondante dans l'un des deux automates.

5.8/ GÉNÉRATION AUTOMATIQUE DE L'ADAPTATEUR

Dans cette section, nous donnons aperçu de notre algorithme, présenté dans [CMM12b], permettant de construire automatiquement un adaptateur pour deux automates d'interface composables A_1 et A_2 conformément à un contrat d'adaptation $\Phi(A_1, A_2)$. L'algorithme parcourt en parallèle A_1 et A_2 et construit au fur et à mesure l'ensemble des états et des transitions de l'adaptateur. En effet, à chaque fois qu'il traverse des transitions étiquetées par des actions de sortie appartenant à *ActIncomp*$(\Phi(A_1, A_2))$, il génère une ou plusieurs transitions étiquetées par leurs actions d'entrée correspondantes. L'algorithme parcourt en profondeur A_1 et A_2 en explorant alternativement leurs états et leurs transitions. Il met à jour au fur et à mesure l'ensemble *S* des états et l'ensemble *T* des transitions de l'adaptateur. L'adaptateur généré est conforme à la spécification donnée dans la définition 5.18.

L'algorithme permet également de détecter s'il y a des chemins qui conduirait vers des états de blocage dans le produit de A_1 et A_2, sans calculer ce dernier. L'existence de ces chemins rend leur composition impossible. Dans ce cas, l'algorithme génère un adaptateur vide (adaptation impossible). Dans le cas contraire, il génère un adaptateur qui respecte les règles d'adaptation et assure la compatibilité de A_1 et A_2 après l'adaptation.

Exemple 5.6. *À titre d'exemple, nous allons considérer un système composé de deux composants logiciels : un composant client qui communique avec un serveur d'applications distant. Les automates d'interface A_C et A_S des deux composants sont décrits dans la figure 5.9.*

Si l'authentification s'achève avec succès, le client envoie une requête req *! qui a pour paramètres son identifiant de connexion. Ensuite il envoie, en argument de l'action* arg *!, le nom d'un média à ouvrir. Le serveur traite les deux actions en exécutant l'action* ouvrir *? qui vérifie en invoquant la base de données, si le média demandé existe et si le client est autorisé à visualiser son contenu. Si le média existe dans la base de données et si le client est autorisé à l'ouvrir, le serveur renvoie le média (action de sortie* media *!) comme valeur de retour. L'action* media *! correspond à l'action d'entrée* service *? du client. Dans le cas où le média n'est pas trouvé, un message d'erreur (action* mdNonTrouve *!) est envoyé au client. Pour se déconnecter, le client envoie une requête* deconnexion *! au serveur. L'action* su *? représente l'opération fournie par le serveur qui permet d'établir la connexion d'un super-utilisateur pour ouvrir une session. L'identifiant de l'administrateur est envoyé en argument de* su*. Le mot de passe est reçu ensuite par le serveur. Les deux automates A_C et A_S sont composables. L'ensemble des actions partagées $Shared(A_C, A_S)$ égal à* {deconnexion}*.*

*Le contrat d'adaptation $\Phi(A_C, A_S)$ de A_C et A_S est défini par l'ensemble $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ tel que : $\alpha_1 = \langle\{$*login*$\}, \{$*identifiant,mdp*$\}\rangle$; $\alpha_2 = \langle\{$*ok*$\},$*

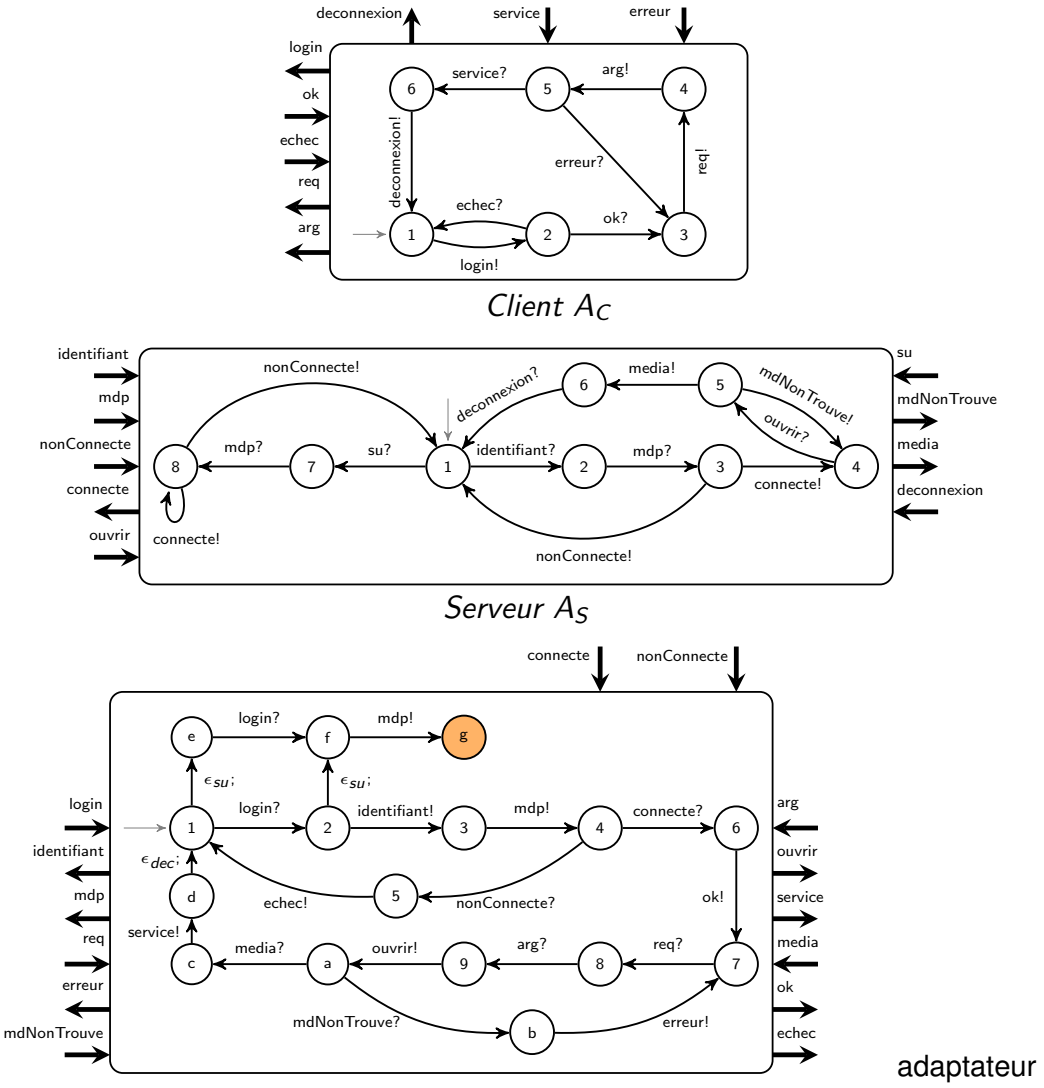


FIGURE 5.9 – Illustration de l'approche d'adaptation

{*connecte*}⟩ ; α_3 = ⟨{*echec*}, {*nonConnecte*}⟩ ; α_4 = ⟨{*req*,*arg*},{*ouvrir*}⟩ ; α_5 = ⟨{*erreur*},{*mdNonTrouve*}⟩ ; α_6 = ⟨{*service*},{*media*}⟩.

*Dans ce qui suit nous donnons un exemple des signatures et des sémantiques des actions (*login, identifiant, mdp*) reliées par la règle* α_1*. Les signatures :* login*(*uid,mdp*)→(*usr*),* identifiant*(*id*)→(),* mdp*(*pass*)→(*u*). La sémantique de* login *:* $B_C.\mathcal{O}(login).P \equiv$ uid $> 0 \wedge 8 \leq$ mdp.length*()* ≤ 10 *;* $B_C.\mathcal{O}(login).O \equiv$ usr.getId*()>0 . La sémantique de* identifiant *:* $B_S.\mathcal{I}(identifiant).P \equiv$ id ≥ 1 *;* $B_S.\mathcal{I}(identifiant).Q \equiv$ vrai*. Et celle de* mdp *:* $B_S.\mathcal{I}(mdp).P \equiv$ *6* $\leq$ pass.length*()* $\leq$ *10 ;* $B_S.\mathcal{I}(mdp).Q \equiv$ u.getId*()* ≥ 1*. L'adaptabilité sémantique est facilement vérifiable entre les actions dans la règle* α_1*. Et nous l'avons également vérifiée sur l'ensemble des règles [CMM12b].*

Après la vérification de l'adaptabilité sémantique, nous avons généré, en appliquant notre algorithme, l'automate d'interface dans la figure 5.9 qui représente l'adaptateur de A_C *et* A_S*, conformément au contrat d'adaptation* $\Phi(A_C, A_S)$*. Il est défini en fonction de l'ordonnancement des actions des deux automates d'interface* A_C *et* A_S*, et*

également en fonction du contrat d'adaptation $\Phi(A_C, A_S)$*. Donc les chemins de l'adaptateur sont calculés de telle sorte que les inadéquations entre les actions des deux automates soient résolus. Par exemple, le chemin (1,login ?,2, identifiant !,3,mdp !,4), permet des résoudre l'inadéquation entre l'action* $login$ *dans* A_C *et les actions correspondantes* $identifiant$ *et* mdp *dans* A_S*. Par ailleurs, la transition (d,*ϵ_{dec}*;,1) remplace les transitions, (6,*deconnexion *!, 1) de* A_C *et (6,*deconnexion *?,1) de* A_S*, dans l'adaptateur. Les transitions (1,*ϵ_{su}*;,e) et (2,*ϵ_{su}*;,f) remplacent la transition (1,*su *?,7) de* A_S *dans l'adaptateur.*

5.9/ Conclusion

Dans ce chapitre, nous avons présenté une approche formelle basée sur l'approche optimiste des automates d'interface pour la construction des systèmes à base de composant orientés objet, corrects par la conception. Nous avons introduit les contrats comportementaux combinant les niveaux protocole et sémantique pour la spécification des interfaces des composant orientés objet. Ce qui nous a permis de proposer une méthodologie pour l'assemblage de composants spécifiés avec des contrats comportementaux, dans le contexte orienté objet. Nous avons également défini une relation de raffinement en tenant compte de ce contexte, pour vérifier la substituabilité des composants dans le but de leur réutilisation. Cette relation permet également de vérifier l'implémentabilité indépendante des composants. Et enfin, nous avons également proposé une approche d'adaptation permettant la résolution automatique des inadéquations que pourraient présenter des composants spécifiés avec des contrats comportementaux.

5.10/ Bilan

Les résultats de ce chapitre concernent les éléments suivants :

1. **Projets** :
   - Le projet ANR TACOS, 2007-2010.
   - L'action européenne COST IC1201 - Behavioural Types for Reliable Large-Scale Software Systems (BETTY), 2012-2016.
2. **Encadrement de thèse de doctorat** :
   - La thèse de Sebti Mouelhi [Mou11], 2011.
3. **Publications** : [MACM15b, CMM12b, CMM10e, CMM10b, MCM09].

6

VÉRIFICATION DE L'INTEROPÉRABILITÉ DES COMPOSANTS AVEC LE RAFFINEMENT B

6.1/ INTRODUCTION

Le travail présenté dans ce chapitre décrit notre approche permettant, d'une part, de spécifier les contrats comportementaux des composants avec le langage B [Abr96a], et d'autre part, de vérifier leur assemblage avec le raffinement.

Pour exploiter réellement l'approche de développement à base de composants, il doit être possible d'acquérir des composants développés par des tiers et de les assembler de manière à obtenir le comportement souhaité du système à implémenter. L'assemblage de composants tiers est possible à condition des respecter les exigences suivantes :

— Les composants à assembler doivent être décrits suffisamment, sans pour autant dévoiler leurs codes sources, dans le but de pouvoir décider sur la possibilité de leur assemblage.

— Pour que différents composants puissent interagir, ils doivent s'accorder sur le format des données à échanger entre eux. Par conséquent, chaque interface d'un composant doit être équipée d'un modèle de données décrivant le format des données acceptées et produites par le composant. De plus, il ne suffit pas de donner uniquement la signature des opérations d'interface (l'opération foo, par exemple, prend deux entiers et donne un entier comme résultat), comme cela est courant dans les langages de description d'interface classiques. Il est également nécessaire de décrire l'effet d'une opération d'interface (par exemple, l'opération foo prend deux entiers et donne leur somme en conséquence).

Afin de répondre aux exigences présentées ci-dessus, une spécification d'interface de composant doit contenir les informations suivantes : (i) un modèle de données associé à chaque interface requise et fournie d'un composant (modèle de données d'interface), (ii) les pré et post-conditions pour chaque opération d'interface, de sorte que la conception par contrat [Mey97] devienne possible.

Nous proposons d'utiliser des diagrammes de classes UML [uml] pour exprimer le modèle de données d'interface et la notation formelle B [Abr96a]. Sur la base de ces ingrédients, nous vérifions l'interopérabilité entre deux composants en utilisant une relation de raffine-

ment entre une adaptation de leurs spécifications d'interfaces. Cette vérification repose sur une approche de correspondance de spécifications [ZW95].

Nous avons choisi d'utiliser la méthode B parce que ses concepts sous-jacents de machine et de raffinement correspondent bien aux composants et à leur interopérabilité, et parce que B est dotée d'un support d'outils puissant. Ainsi, nous pouvons exploiter les technologies existantes pour prouver l'interopérabilité des composants.

Le reste du chapitre est organisé comme suit : dans la section 6.2, nous discutons des travaux connexes. Ensuite, nous introduisons la spécification des interfaces de composants dans la section 6.3. La notion d'interopérabilité entre deux composants est définie dans la section 6.4. Le chapitre se termine par quelques remarques finales dans la section 6.5 et un bilan concernant les publications dans 6.6.

6.2/ ÉTAT DE L'ART ET CONTRIBUTIONS

Dans [HSS02], les auteurs ont étudié le rôle des modèles de composants dans la spécification des composants. La spécification d'un modèle de composant permet d'obtenir des spécifications plus concises de composants individuels, car celles-ci peuvent faire référence à la spécification du modèle de composant. La spécification du modèle de composant n'a pas besoin d'être répétée pour chaque composant individuel adhérant au modèle de composant en question. Dans ce travail, nous étudions les ingrédients nécessaires à une spécification de composant pour pouvoir construire un système logiciel à partir de composants. Ces ingrédients sont indépendants des modèles de composants concrets. Plusieurs propositions de spécification de composants ont déjà été faites. Elles ont en commun de ne pas avoir d'équivalent de notre modèle de données d'interface et de ne pas prendre en compte les problèmes d'interopérabilité, mais uniquement la spécification de composants individuels.

Un groupe de travail Allemand «Gesellschaft für Informatik» (GI) a défini une structure de spécification pour les composants métier [ABC+02]. Cette structure comprend sept niveaux, à savoir marketing, tâche, terminologie, qualité, coordination, comportement et interface. Notre structure de spécification couvre les niveaux terminologie, coordination, comportement, et interface, en proposant des moyens concrets de spécifier chacun de ces niveaux. Les autres niveaux de la proposition GI concernent les aspects non fonctionnels des composants.

Beugnard et al. [BJPW99] proposent de définir des contrats pour les composants. Ils distinguent quatre niveaux de contrats : syntaxique, comportemental, synchronisation et qualité de service. Le niveau syntaxique ne spécifie que les signatures des opérations, le niveau de comportement contient des pré et post-conditions, le niveau de synchronisation correspond aux protocoles d'utilisation et le niveau de qualité de service traite des aspects non fonctionnels. Contrairement à notre approche, Beugnard et al. n'ont pas introduit de modèles de données pour leurs interfaces.

L'approche de spécification de composants de Lau et Ornaghi [LO01] est plus proche de la nôtre, car chaque composant possède un contexte qui correspond à notre modèle de données d'interface. Un contexte est une spécification algébrique, composée d'une signature, d'axiomes et de contraintes. En revanche, nous estimons plus approprié d'opter pour une spécification orientée objet du modèle de données d'une interface de com-

posant. Cela permet d'utiliser les concepts orientés objet, comme l'héritage, qui sont fréquemment utilisés dans la pratique.

Cheesman et Daniels [CD01] proposent un processus pour spécifier un SBC. Ce processus commence par une description informelle des exigences et produit une architecture montrant les composants à développer ou à réutiliser, leurs interfaces et leurs dépendances. Pour chaque opération d'interface, une spécification est élaborée, consistant en une précondition, une postcondition, et éventuellement un invariant. Cette approche suit le principe de la conception par contrat [Mey97]. Les spécifications de nos interfaces de composants s'inspirent des travaux de Cheesman et Daniels, qui montrent clairement que, pour chaque interface, un modèle de données est nécessaire. Toutefois, Cheesman et Daniels ne considèrent pas le cas où des composants existants avec éventuellement des modèles de données différents doivent être assemblés et ne définissent donc pas la notion d'interopérabilité.

Bastide et al. [BSP99] utilisent des réseaux de Petri pour spécifier le comportement des objets CORBA, y compris la sémantique et les protocoles des opérations. La différence avec notre approche est que nous prenons en compte les invariants des spécifications des interfaces. Zaremski et Wing [ZW95] proposent une approche intéressante pour comparer deux composants logiciels afin de déterminer si un composant peut être remplacé par un autre. Ils utilisent des spécifications formelles pour modéliser le comportement des composants et le prouveur Larch pour prouver l'adéquation des spécifications des composants. D'autres [Han98, VNnT99] ont également proposé d'enrichir les spécifications des interfaces de composants en fournissant des informations aux niveaux de la signature, de la sémantique et du protocole. Malgré ces améliorations, nous pensons qu'un modèle de données est également nécessaire pour effectuer une vérification formelle de la compatibilité des interfaces.

Dans [LS08], une extension de nos travaux a été proposée en considérant en plus la problématique d'adaptation des composants. Par ailleurs, la méthode B événementiel [Abr10] a été également exploitée pour le développement de systèmes à base de composants/services dans [AA13, AA15, AAAL10a, AAAL10b] (publiés après les travaux présentés dans ce chapitre). Ainsi, dans [AA13, AA15], les auteurs proposent une approche exploitant le raffinement pour modéliser et vérifier la composition des services Web décrits avec BPEL. Dans ces travaux, contrairement à notre approche, la relation de compatibilité entre les interfaces des composants n'est pas considérée. Les auteurs se sont focalisés sur la formalisation et la vérification de la relation de décomposition de BPEL, en exploitant le raffinement B. Dans [AAAL10a, AAAL10b], les auteurs exploitent le B événementiel pour vérifier par preuve l'assemblage des services, ainsi que leurs propriétés, dans le modèle à composants Kmelia [AAA06, AAA08]. Notre proposition comparée à ces derniers travaux, ne concerne pas un modèle de composants spécifique. De plus, dans notre démarche, UML est exploité pour décrire le modèle de données des composants.

Contributions : Notre travail se distingue, globalement, par rapport aux travaux existants, en proposant une démarche permettant, d'une part, de spécifier les interface des composants, en indiquant les ingrédients nécessaires (comme le modèle de données), pour vérifier leurs compatibilité, et d'autre part, d'exploiter le raffinement B pour effectuer cette vérification.

6.3/ SPÉCIFICATION DES INTERFACES DE COMPOSANTS AVEC B

Notre objectif est de proposer un moyen de spécifier les composants sous forme de boîtes noires tout en exhibant uniquement l'information nécessaire à leur déploiement. Par conséquent, les spécifications des interfaces de composants jouent un rôle important, car les interfaces sont les seuls points d'accès à un composant.

6.3.1/ DÉFINITION

Une spécification d'un composant doit contenir toutes les informations nécessaires pour décider si le composant peut être utilisé ou non dans un contexte donné. Elles concernent les données utilisées par le composant ainsi que son comportement visible pour son environnement. Ce comportement est réalisé par des services pouvant être utilisés par d'autres composants ou systèmes logiciels. Ces services sont décrits dans les interfaces fournies. Cependant, dans de nombreux cas, un composant dépend des services offerts par d'autres composants. Dans ce cas, le composant ne peut fonctionner correctement qu'en présence d'autres composants offrant les services requis. Les services requis par un composant sont décrits dans les interfaces requises.

Une spécification d'interface comprend les parties suivantes :

— La spécification de son modèle de données d'interface qui spécifie (i) les types utilisés dans l'interface, (ii) un état des données nécessaire pour exprimer les effets des opérations, et (iii) des invariants sur cet état des données. Dans ce qui suit, nous utilisons des diagrammes de classes UML pour exprimer le modèle de données pour des raisons de lisibilité. Ce diagramme de classe est ensuite automatiquement transformé en une spécification B [MS99]. D'autres langages, tels que Object-Z [Smi99], conviennent également pour spécifier le modèle de données d'interface (voir [HSS02]).

— Un ensemble de spécifications des opérations. Une spécification d'opération se compose de sa signature (c'est-à-dire des types de ses paramètres d'entrée et de sortie), de sa précondition dans laquelle l'opération peut être invoquée et de sa postcondition exprimant l'effet de l'opération. Les pré et post-conditions font référence au modèle de données d'interface.

Pour chaque interface de composant, une machine B est définie. Elle contient les spécifications du modèle de données d'interface et des opérations.

6.3.2/ ÉTUDE DE CAS

Nous illustrons notre approche en considérant un système de réservation d'hôtel, une variante de l'étude de cas utilisée par Cheesman et Daniels [CD01]. L'architecture du système de réservation global utilisant des composants est décrite dans la figure 6.1 en utilisant la notation UML 2 [uml]. Il dispose de deux interfaces fournies, *IMakeReservation* et *ITakeReservation*, et de deux interfaces requises *IHotelHandling* et *ICustomerHandling*. Un des composants utilisé est *HotelMgr* avec son interface fournie *IHotelMgt*.

Dans ce qui suit, nous examinerons plus en détail les interfaces *IHotelHandling* et *IHotelMgt* afin de prouver que le composant *HotelMgr*, doté de son interface *IHotelMgt*, répond aux besoins de l'interface *IHotelHandling*.

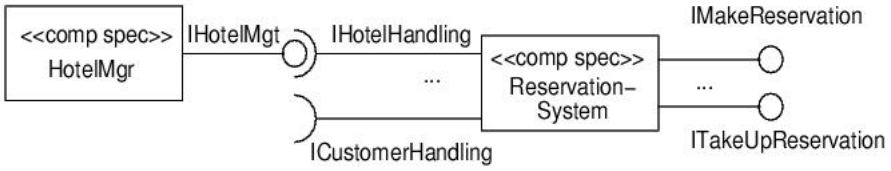


FIGURE 6.1 – Architecture à composants du système de réservation d'hôtel

6.3.2.1/ SPÉCIFICATION DE L'INTERFACE IHOTELHANDLING

La figure 6.2 montre le modèle de données d'interface, exprimé sous forme de diagramme de classes. La spécification B correspondante est obtenue par les règles de transformation systématiques appliquées au diagramme de classes UML de la manière suivante. Comme dans B toutes les variables doivent avoir des noms différents, nous utilisons la convention de dénomination selon laquelle tous les noms de variables sont préfixés par une abréviation du nom de la classe à laquelle ils appartiennent. Par exemple, l'attribut *hotel* de la classe *ReservationDetails* devient la variable *RD_hotel* dans la machine B *IHotelHandling*.

Classes. Comme on peut le voir sur la figure 6.3, les classes du modèle de données d'interface et les types de leurs attributs sont représentés sous la forme d'ensembles. Les attributs sont définis comme des variables qui sont des fonctions. Les ensembles d'objets existants dans le système, tels que *cust*, *res*, *hotels* et *rooms*, sont également définis en tant que variables. Par exemple, *cust* est déclarée comme sous-ensemble de l'ensemble Client.

Associations entre classes. Elles sont spécifiées sous forme de variables dont le type est une fonction ou une relation (en fonction des multiplicités de l'association) entre les ensembles modélisant les classes associées. La figure 6.4 montre la spécification B des associations entre les classes : *Reservation* et *Customer*, *Reservation* et *Hotel*, *Reservation* et *Room*.

Contraintes d'intégrité. Elles sont spécifiées en tant que prédicats dans la clause INVARIANT de la machine B. Par exemple, la contrainte qui exprime le fait qu'une réservation est réclamée si et seulement si une chambre lui est attribuée est exprimée par :

$$\forall(re).((re \in res)) \leftarrow ((Res_claimed(re) = TRUE) \iff (re \in dom(assoc_allocation)))$$

Opérations de classe. Les opérations *R_availabale* et *R_stayPrice* de la classe *Room* sont spécifiées en tant que variables dont les types sont des fonctions, exprimées dans la clause INVARIANT de la machine B comme suit :

$R_availabale \in Room \times DateRange \rightarrow BOOL$
$R_stayPrice \in Room \times DateRange \rightarrow Currency$

Opérations. Elles sont spécifiées dans la clause OPERATIONS de la machine B. La figure 6.5 donne deux exemples d'opérations : *getHotelDetails* donne comme résultat

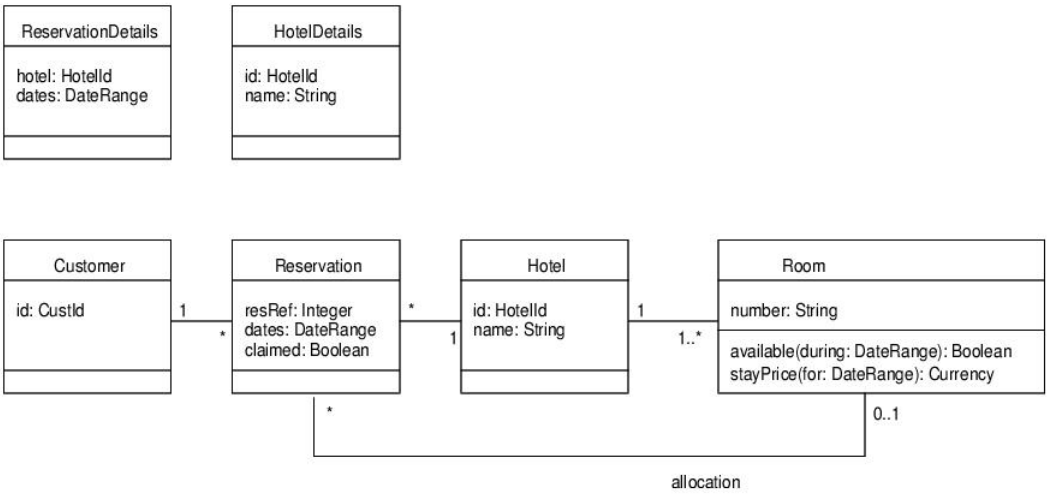


FIGURE 6.2 – Le modèle de donnée d'interface de *IHotelHandling*

MACHINE $IHotelHandling$

SETS $ReservationDetails; HotelId; DateRange; HotelDetails;$

$Customer; CustID; Reservation; Hotel; Room; Currency$

VARIABLES $RD_hotel, RD_dates, HD_id, HD_name, C_id, cust, RES_resRef, RES_dates,$

$RES_claimed, RES_number, Hotel_id, Hotel_name, hotels, res, R_number,$

$R_available, R_stayPrice, rooms$

INVARIANT

$/ * classReservationDetails * /$

$RD_hotel \in ReservationDetails \rightarrow HotelId \ \wedge \ RD_dates \in ReservationDetails \rightarrow DateRange \ \wedge$

$/ * classHotelDetails * /$

$HD_id \in HotelDetails \rightarrow HotelId \ \wedge \ HD_name \in HotelDetails \rightarrow STRING \ \wedge$

$/ * classReservation * /$

$RES_resRef \in Reservation \rightarrow INTEGER \ \wedge \ RES_dates \in Reservation \rightarrow DateRange \ \wedge$

$RES_claimed \in Reservation \rightarrow BOOL \ \wedge \ RES_number \in INTEGER \ \wedge$

$/ * classHotel * /$

$Hotel_id \in Hotel \rightarrow HotelId \ \wedge \ Hotel_name \in Hotel \rightarrow STRING$

$/ * state \quad of \quad the \quad system * /$

$cust <: Customer \ \wedge \ hotels <: Hotel \ \wedge \ res <: Reservation \ \wedge \ rooms <: Room \ \ldots$

FIGURE 6.3 – La spécification B des classes dans *IHotelHandling*

une collection de détails d'hôtel, où le nom de l'hôtel doit correspondre à son paramètre d'entrée *match* ; *makeReservation* crée une réservation en fonction de certains clients et de certains détails de la réservation.

$$
\begin{array}{l}
\textbf{VARIABLES} \ \ldots \\
\qquad assoc_ResCust, assoc_ResHot, assoc_Allocation \\
\textbf{INVARIANT} \ \ldots \\
\qquad assoc_ResCust \in Reservation \rightarrow Customer \ \wedge \ assoc_ResHot \in Reservation \rightarrow Hotel \ \wedge \\
\qquad assoc_Allocation \in Reservation \nrightarrow Room
\end{array}
$$

FIGURE 6.4 – La spécification B des associations entre les classes

$$
\begin{array}{l}
\textbf{OPERATIONS} \ \ldots \\
hotdets \leftarrow getHotelDetails(match) \ \hat{=} \\
\textbf{PRE} \ match \in STRING \\
\textbf{THEN} \ hotdets := \{hdx | hdx \in HotelDetails \ \wedge \ \exists ho.((ho \in Hotel) \ \wedge \ (HD_id(ho) = HD_id(hdx)) \ \wedge \\
\qquad (HD_name(ho) = HD_name(hdx)) \ \wedge \ (matches(match, HD_name(hdx)) = TRUE))\} \\
\textbf{END}; \\
resref \leftarrow makeReservation(pres, cus) \ \hat{=} \\
\textbf{PRE} \ pres \in ReservationDetails \ \wedge \ cus \in CustID \wedge \ \exists ho.(ho \in hotels \ \wedge \ HD_id(ho) = RD_hotel(pres)) \\
\textbf{THEN} \ ANY \ ho \ WHERE \ (ho \in hotels \ \wedge \ H_id(ho) = RD_hotel(pres)) \ THEN \\
\qquad ANY \ nres \ WHERE \ nres \in Reservation \ \wedge \ nres \notin res \ \wedge \\
\qquad nres \notin dom(assoc_Allocation) \ \wedge \ nes \notin dom(assoc_ResHot) \ THEN \\
\qquad res := res \cup nres \ || \ assoc_ResHot(nres) := ho \ || \ C_id(assoc_ResCust(nres)) := cus \\
\qquad || \ resref := RES_number + 1 \ || \ RES_resRef(nres) := RES_number + 1 \\
\qquad || \ RES_dates(nres) := RD_dates(pres) \ || \ RES_claimed(nres) := FALSE \quad END \ \textbf{END}
\end{array}
$$

FIGURE 6.5 – La spécification B des opérations

6.3.2.2/ SPÉCIFICATION DE L'INTERFACE IHOTELMGT

Nous supposons qu'un composant *HotelMgr* est disponible et peut gérer des hôtels avec différents types de chambres. La figure 6.6 montre le modèle de données d'interface pour l'interface fournie *IHotelMgt*. Les différences entre les interfaces *IHotelHandling* et *IHotelMgt* sont dues à la nouvelle classe *RoomType* dans *IHotelMgt* permettant de prendre en compte différents types de chambres. Toutes les classes présentes dans le modèle de données d'interface de *IHotelHandling* sont également présentes dans le modèle de données d'interface de *IHotelMgt*. Cependant, les classes *ReservationDetails* et *HotelDetails* ont maintenant un attribut supplémentaire lié à *RoomType*. Dans ce qui suit, nous ne montrons qu'une partie de la spécification B de l'interface *IHotelMgt* qui exprime les modifications par rapport à *IHotelHandling*.

La classe *RoomType* et ses associations. La figure 6.7 présente la spécification de la classe *RoomType*, son attribut et les associations entre les classes *Room* et *Roomtype*, *Reservation* et *RoomType*.

Les invariants. Les opérations doivent respecter deux propriétés invariantes importantes :

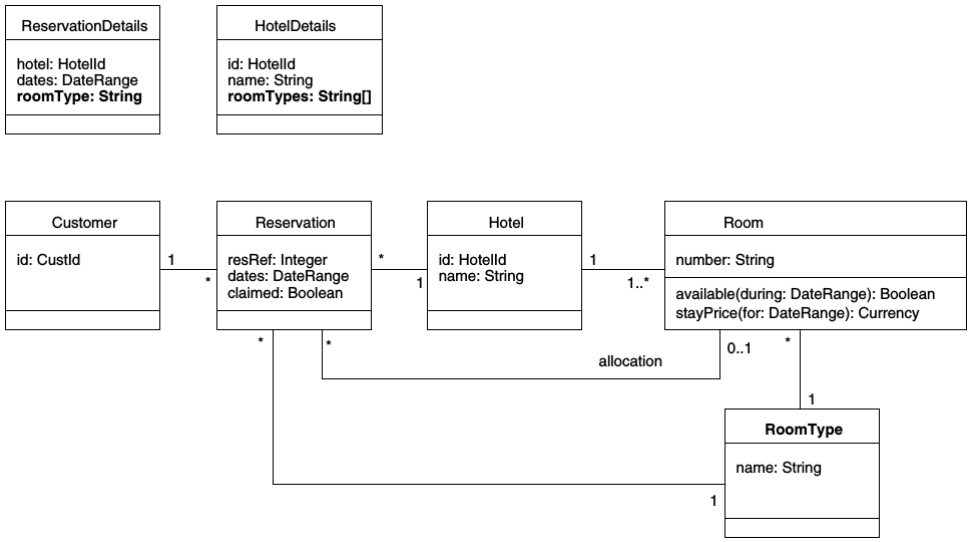


FIGURE 6.6 – Le modèle de données d'interface de *IHotelMgt*

— pour chaque objet de la classe *ReservationDetails* associé à un objet de la classe *Hotel*, la valeur de sa variable *RD_roomType* est la valeur du nom d'attribut d'un objet de type *RoomType* associé à une chambre appartenant à l'hôtel :

$\forall(pres, ho).(pres \in ReservationDetails \wedge ho \in hotels \wedge HD_id(ho) = RD_hotel(pres) \Rightarrow RD_rooType(pres) \in \{rtn | rtn \in STRING \wedge \exists(sty, ro).((rty \in RoomType) \wedge ro \in Room \wedge ro \in assoc_ResHot^{-}1[\{ho\}] \wedge asso_RoomRt(ro) = rty \wedge RT_name(rty) = rtn)\}))$

— pour chaque objet de la classe *HotelDetails* associé à un hôtel, la valeur de sa variable *RD_roomTypes* est l'ensemble des noms d'attributs des objets de la classe *RoomType* associés à une chambre appartenant à l'hôtel :

$\forall(hdx, ho).(hdx \in HotelDetails \wedge ho \in hotels \wedge HD_id(ho) = HD_id(hdx) \Rightarrow RD_roomTypes(hdx) = \{rtn || rtn \in STRING \wedge \exists(rty, ro).((rty \in RoomType) \wedge ro \in Room \wedge ro \in assoc_ResHot^{-1}[\{ho\}] \wedge assoc_RoomRt(ro) = rty \wedge RT_name(rty) = rtn)\})$

Opérations. L'interface *IHotelMgt* offre des opérations portant les mêmes noms que dans l'interface *IHotelHandling*. Cependant, la spécification de l'opération *makeReservation* est différente de la spécification de la même opération dans *IHotelHandling*, en raison de la classe *RoomType* (voir la figure 6.8).

6.4/ VÉRIFICATION DE L'INTEROPÉRABILITÉ AVEC LE RAFFINEMENT B

Dans ce travail, nous ne traitons que la vérification de l'interopérabilité des composants aux niveaux signature et sémantique. L'interopérabilité au niveau du protocole est traitée

$$
\begin{array}{l}
\textbf{MACHINE}\ IHotelMgt \\
\textbf{SETS}\ \ldots \\
\quad RoomType \\
\textbf{VARIABLES}\ \ldots \\
\quad RT_name, RD_roomType, HD_roomTypes \\
\textbf{INVARIANT}\ \ldots \\
\quad /*\ classRoomType\ */ \\
\quad RT_name \in RoomType \rightarrow STRING\ \wedge\ RD_roomType \in ReservationDetails \rightarrow STRING\ \wedge \\
\quad HD_roomTypes \in HotelDetails \rightarrow POW(STRING) \\
\quad /*\ associations\ */ \\
\quad assoc_RoomRt \in Room \rightarrow RoomType\ \wedge\ assoc_ResRt \in Reservation \rightarrow RoomType\ \wedge\ \ldots
\end{array}
$$

FIGURE 6.7 – La spécification B de la classe *RoomType* dans *IHotelMgt*

$$
\begin{array}{l}
\textbf{OPERATIONS}\ \ldots \\
resref \leftarrow makeReservation(pres, cus) \triangleq \\
\textbf{PRE}\ pres \in ReservationDetails\ \wedge\ cus \in CustID\ \wedge\ \exists ho.(ho \in hotels\ \wedge\ H_id(ho) = RD_hotel(pres)) \\
\textbf{THEN}\ ANY\ ho\ WHERE\ (ho \in hotels \wedge HD_id(ho) = RD_hotel(pres))\ THEN \\
\quad \textbf{ANY romt, ro WHERE romt} \in \textbf{RoomType} \wedge \textbf{RT_name(romt)= RD_roomType(pres)} \wedge \\
\quad \textbf{ro} \in \textbf{rooms} \wedge \textbf{ro} \in \textbf{assoc_RHot}\ ^{-1}[\{\textbf{ho}\}] \wedge \textbf{assoc_RoomRt(ro)} = \textbf{romt} \\
\quad THEN\ ANY\ nres\ WHERE\ nres \in Reservation\ \ldots (see\ figure\ 5) \\
\quad \|\ \textbf{assoc_ResRt(nres):= romt}\ END\ END \\
\textbf{END}
\end{array}
$$

FIGURE 6.8 – L'opération *makeReservation* dans *IHotelMgt*

dans [SC05]. Les composants étant décrits par leurs interfaces, la vérification de leur interopérabilité doit être effectuée au niveau de celles-ci en vérifiant leur compatibilité.

6.4.1/ DÉFINITIONS

Nous donnons d'abord une description intuitive de la compatibilité des l'interfaces de composants dans les sections 6.4.1.1 et 6.4.1.2. Nous montrons ensuite comment cette notion intuitive peut être reliée avec le raffinement en B (section 6.4.1.3).

Une interface fournie PI d'un composant C' peut jouer le rôle d'une interface requise RI d'un composant C si leurs modèles de données d'interface et leurs opérations sont compatibles.

6.4.1.1/ COMPATIBILITÉ DES MODÈLES DE DONNÉES D'INTERFACE

L'idée de base de la compatibilité entre les modèles de données d'interface est que le modèle de données d'interface (IDM) de RI n'est pas plus restrictif que celui de PI. Dans ce cas uniquement, l'IDM de PI peut être utilisé à la place de l'IDM de RI. Ainsi, chaque type ou classe de RI doit avoir une contrepartie dans PI, mais pas nécessairement l'inverse. Cela signifie que PI peut contenir des données qui ne sont pas nécessaires à l'implémentation de RI. Les cas suivants doivent être distingués :

— Les types de base sont compatibles s'ils portent le même nom ou s'il existe une règle explicite stipulant que les deux types sont compatibles.

— Pour les classes, les conditions suivantes doivent être remplies :

(i) Pour chaque classe $class_r$ de l'IDM de RI, il existe une classe $class_p$ dans l'IDM de PI telle que :

— Pour chaque attribut de $class_r$, il existe un attribut de $class_p$ ayant un type compatible.

— Pour chaque opération op_r de $class_r$, il existe une opération op_p de $class_p$, telle que pour chaque type de la signature de op_r, il existe un type compatible dans la signature de op_p [1].

— Il existe une fonction injective $tr : class_r \rightarrow class_p$, qui transforme un objet de $class_r$ en un objet de $class_p$. Il doit être possible de transformer des objets RI en objets PI pour pouvoir les utiliser comme paramètres d'entrée d'opérations PI. La transformation inverse est nécessaire pour transformer les paramètres de sortie des opérations PI en objets RI. Pour les types de données de PI qui n'ont pas d'équivalent dans RI, aucune fonction de transformation n'est nécessaire, car ces données peuvent être ignorées par RI.

(ii) Chaque association dans l'IDM de RI a une contrepartie dans l'IDM de PI, dont les contraintes de cardinalité ne sont pas plus fortes que dans l'IDM de PI.

(iii) L'invariant inv_p de l'IDM de PI implique l'invariant transformé $tr(inv_r)$ de l'IDM de RI. Cette condition garantit que les états autorisés par l'IDM de RI sont également autorisés par l'IDM de PI. Cependant, pour montrer l'implication souhaitée, il est nécessaire que les deux conditions fassent référence au même modèle de données. Par conséquent, les données apparaissant dans l'invariant inv_r (qui appartient à l'IDM de RI) doivent être remplacées par leurs équivalents dans l'IDM de PI, comme défini par le paramètre de la fonction tr.

6.4.1.2/ COMPATIBILITÉ DES OPÉRATIONS

Pour chaque opération op_r de l'interface RI, il doit exister une opération op_p de l'interface PI telle que :

(i) Leurs signatures sont compatibles, c'est-à-dire que, pour chaque type de signature de op_r, il existe un type compatible dans la signature de op_p.

1. Ceci est une version simple de la correspondance de signature. Zaremski et Wing [ZW95] donnent différentes variantes de correspondance des signatures dans un contexte algébrique.

REFINEMENT $New_IHotelMgt$

REFINES $IHotelHandling$

SETS $RoomType$

VARIABLES $RD_hotelRef, RD_datesRef, HD_idRef, HD_nameRef, C_idRef, custRef,$
$RES_resRefRef, RES_datesRef, RES_claimedRef, RES_numberRef, Hotel_idRef,$
$Hotel_nameRef, hotelsRef, resRef, R_numberRef, R_availableRef, R_stayPriceRef, RT_name,$
$RD_roomType, HD_roomTypes, assoc_RoomRt, assoc_ResRt$

INVARIANT

$/ * renaming\ variables * /$
$RD_hotelRef = RD_hotel \ \wedge \ RD_datesRef = RD_dates \ \wedge HD_idRef = HD_id \ \wedge$
$HD_nameRef = HD_name \ \wedge \ C_idRef = C_id \ \wedge \ custRef = cust \ \wedge$
$RES_resRefRef = RES_resRef \ \wedge \ RES_datesRef = RES_dates \ \wedge$
$RES_claimedRef = RES_claimed \ \wedge \ RES_numberRef = RES_number \ \wedge$
$Hotel_idRef = Hotel_id \ \wedge \ Hotel_nameRef = Hotel_name \ \wedge$
$hotelsRef = hotels \ \wedge \ resRef = res \ \wedge \ R_numberRef = R_number \ \wedge$
$R_availableRef = R_available \ \wedge \ R_stayPriceRef = R_stayPrice \ \wedge$
$/ * type\ of\ the\ attributes\ related\ to\ RoomType * /$
$RT_name \in RoomType \rightarrow STRING \ \wedge \ RD_roomType \in ReservationDetails \rightarrow STRING \ \wedge$
$HDroomTypes \in HotelDetails \rightarrow POW(STRING) \ \wedge \ assoc_RoomRt \in Room \rightarrow RoomType \ \wedge$
$assoc_ResRt \in Reservation \rightarrow RoomType \ \wedge \ \ldots$

FIGURE 6.9 – Spécification B de *New_IHotelMgt*

(ii) La précondition transformée de op_r, $tr(pre(op_r))$ implique la précondition de op_p. En ce qui concerne la relation d'implication sur les invariants d'IDM requis pour la compatibilité des IDM, nous devons transformer les données apparaissant dans la précondition de op_r.

(iii) La postcondition de op_p, $post(op_p)$ implique la postcondition transformée de opr, $tr(post(op_r))$.

Cette définition de la compatibilité des opérations correspond à la notion de correspondance de "plug-in-matching" telle qu'elle est définie par Zaremski et Wing [ZW97]

6.4.1.3/ VÉRIFICATION DE LA COMPATIBILITÉ DES INTERFACES AVEC LE RAFFINEMENT B

Dans cette section, nous montrons qu'il est possible d'utiliser le raffinement B pour prouver que deux composants sont compatibles aux niveaux signature et sémantique. Nous donnons d'abord les conditions du raffinement dans B [Abr96a], puis nous montrons comment la compatibilité des interfaces des composants peut être définie avec le raffinement B.

Le raffinement B. Soit M et N deux spécifications B. Dans ce qui suit, nous donnons les principales conditions qui doivent être vérifiées entre M et N pour montrer que N raffine M. La spécification M est plus abstraite que N, mais il peut aussi s'agir d'un raffinement d'une autre spécification.

(i) Les variables d'état d'une machine de raffinement doivent être différentes de celles de la machine abstraite.

(ii) La spécification abstraite M a une initialisation T_m qui satisfait son invariant I_m. Une spécification raffinée N a une initialisation T_n et un invariant de collage J_n. Donc, si N raffine M, alors T_n doit satisfaire J_n d'une manière cohérente (non contradictoire) avec T_m. Formellement, T_n est un raffinement de T_m, si et seulement si $\neg[T_m]\neg J_n$ est vraie pour tout état atteignable à partir de T_n.

(iii) Toute opération définie dans M doit être définie dans N, c'est-à-dire que toutes les opérations abstraites doivent être raffinées.

(iv) Si une opération OP_n définie dans N raffine une opération OP_m dans M, OP_m et OP_n doivent avoir la même signature.

(v) Soit $OP_m = PRE\ P_m\ THEN\ S_m\ END$ et $OP_n = PRE\ P_n\ THEN\ S_n END$ deux opérations respectivement dans M et N. Soit I_m l'invariant défini dans M et J_n l'invariant de collage défini dans N. Lorsque l'opération OP_n est un raffinement de l'opération OP_m, les conditions suivantes sont remplies :

— $I_m \wedge J_n \wedge P_m \Rightarrow [S_n]\neg[S_m]\neg J_n$, si les opérations n'ont pas de sorties.

— Si out_m et out_n sont les sorties respectives de OP_m et OP_n, la condition suivante doit être remplie : $I_m \wedge J_n \wedge P_m \Rightarrow [S_n[out_n/out_m]]\neg[S_m]\neg(J_n \wedge out_m = out_n)$.

— $I_m \wedge J_n \wedge P_m \Rightarrow P_n$

Ces dernières conditions expriment le fait que lorsque l'opération OP_m est raffinée par OP_n, alors pour toute exécution raffinée de S_n sur un état dans lequel $I_m \wedge J_n \wedge P_m$ est maintenue, il existe une exécution abstraite de S_m (S_m et S_n sont des substitutions généralisées). Nous pouvons conclure que pour tout OP_m raffinée par OP_n, la précondition de OP_m implique la précondition de OP_n (car le raffinement affaiblit les préconditions) et les états dans lesquels la postcondition OP_n est vérifiées sont liés aux états dans lesquels la postcondition de OP_m est vérifiée.

Raffinement des interfaces et compatibilité. Considérons maintenant le cas où M est une spécification B d'une interface requise RI et N est une spécification B d'une interface fournie PI. Les conditions de raffinement de M avec N concernant l'initialisation [2] et les opérations impliquent les conditions de la compatibilité entre les interfaces requises et fournies, c'est-à-dire que le raffinement B est suffisant pour la compatibilité d'interface. Cependant, la condition de raffinement concernant la disjonction des variables d'état (condition (i)) ne peut pas être garantie (et n'est pas nécessaire pour la compatibilité des interfaces de composants). Par conséquent, pour utiliser le raffinement B pour prouver la compatibilité entre RI et PI, il est nécessaire de transformer la spécification B de PI afin de satisfaire la condition de raffinement (i). Cette transformation est effectuée comme suit :

2. Une initialisation raisonnable doit être choisie lors de la représentation des interfaces de composants en tant que machines B, par exemple en utilisant des ensembles vides.

— la spécification B de PI est transformée en une spécification New_PI qui est une spécification raffinée de RI,

— New_PI ne contient pas les ensembles déjà définis dans RI et PI,

— les variables définies à la fois dans RI et dans PI sont renommées dans New_PI,

— l'invariant de New_PI est constitué de l'invariant de PI, où la renommage de variables a été effectué, et d'un invariant de collage qui relie les noms de variables nouvellement introduites à leurs équivalents dans RI. Après avoir effectué ces étapes, nous pouvons vérifier que RI est compatible avec PI en prouvant que New_PI raffine RI.

6.4.2/ ÉTUDE DE CAS

Dans cette section nous montrons comment prouver que l'interface requise *IHotelHandling* est compatible avec l'interface fournie *IHotelMgt* à l'aide du raffinement B. La figure 6.9 présente une partie de la spécification B *New_IHotelMgt* obtenue en transformant *IHotelMgt* conformément aux étapes décrites ci-dessus. Les principales modifications concernent le changement des noms des variables également définies dans *IHotelHandling*, la définition de l'invariant de collage, et la définition des ensembles et des propriétés d'invariant également définis dans *IHotelHandling*. Nous utilisons l'outil Atelier B [STE] pour vérifier que *New_IHotelMgt* raffine *IHotelHandling*. Les résultats de la vérification sont les suivants.

— L'atelier B a généré 197 obligations de preuve évidentes et 22 obligations de preuve pour la spécification B *IHotelHandling*. Toutes ces obligations de preuve ont été prouvées automatiquement.

— L'atelier B a généré 243 obligations de preuve évidentes et 13 obligations de preuve pour la spécification B *New_IHotelMgt*. 12 obligations de preuve ont été prouvées automatiquement et une a été facilement prouvée manuellement.

D'après ces résultats, nous concluons que *New_IHotelMgt* raffine *IHotelHandling*. Par conséquent, l'interface requise *IHotelHandling* est compatible avec l'interface fournie *IHotelMgt* aux niveaux signature et sémantique [3].

6.5/ CONCLUSION

Nous avons présenté une approche pour spécifier les interfaces des composants indépendantes des modèles de composants spécifiques. Sur la base de cette spécification, nous avons défini une notion de compatibilité entre les interfaces de composants qui permet de vérifier si deux composants peuvent ou non interagir via les interfaces données. Nous avons montré qu'il est possible d'utiliser le raffinement B pour prouver la compatibilité de deux composants aux niveaux signature et sémantique.

Contrairement aux travaux précédents, notre spécification contient un modèle de données associé avec chaque interface de composant. Sans un tel modèle de données

3. Pour transformer les objets des classes *ReservationDetails* et *HotelDetails* (fonction tr), nous utilisons un type de chambre par défaut appelé «Standard».

d'interface explicite, il ne serait pas possible de vérifier l'interopérabilité des composants sans connaître les détails de leur implémentation.

6.6/ BILAN

Les résultats de ce chapitre concernent les éléments suivants :

1. **Encadrement de stage de Master 2** : stage de Charles Guillemmot, concernant la modélisation avec la méthode B des systèmes à base de composants, 2008.
2. **Publications** : [CHS06, SC05].

7

VÉRIFICATION DES PROTOCOLES DE COORDINATION REO AVEC SPIN

7.1/ INTRODUCTION

Ce chapitre est dédié la présentation de notre travail sur la vérification des protocoles de coordination dans les SBC, spécifiés avec le langage Reo [Arb04]. Ce travail est le résultat de ma collaboration avec l'équipe du professeur Farhad Arbab du groupe méthodes formelles du centre de recherche CWI, suite à mes séjours scientifiques en 2017, 2018 et 2019.

Dans ce travail, nous nous plaçons dans le contexte de la conception d'un SBC, constitué d' un ensemble de composants hétérogènes, dont les interactions sont coordonnées par un protocole exogène. Ce dernier représente une sorte de contrat comportemental entre les différentes entités en interaction. Une difficulté majeure dans le développement de tels systèmes concerne la spécification et la vérification d'un tel protocole. En effet, le comportement correct de chaque composant individuel, faisant partie du SBC, est insuffisant pour garantir la fiabilité du système composé, car son comportement dépend du protocole qui coordonne l'ensemble de ses composants. Ainsi, La spécification précise et la fiabilité des protocoles de coordination jouent donc un rôle crucial dans la construction des SBC, en général, et des systèmes complexes concurrents et distribués, en particulier.

Notre intérêt porte sur le développement de protocoles fiables, spécifiés avec un langage de coordination où leur conception, leur vérification des propriétés, et leur implémentation, reposent sur le même modèle exact. Dans ce contexte, pour construire des protocoles complexes de manière compositionnelle, nous utilisons Reo, un puissant langage de coordination basé sur des canaux avec une sémantique bien définie. Le modèle Reo d'un système concurrent se compose d'un ensemble de composants qui interagissent les uns avec les autres via un ensemble de constructions de protocole, spécifiées en externe (en dehors des composants), appelées connecteurs, qui imposent des contraintes de synchronisation lors des échanges de données entre ces composants. Les connecteurs primitifs et complexes sont spécifiés dans une syntaxe graphique ou textuelle [DA18]. Et de nombreux formalismes sémantiques existent pour la spécification formelle de leurs comportements [JA12]. Nous proposons donc d'utiliser Reo pour garantir une spécification précise des protocoles complexes. Notre objectif principal est de proposer une approche formelle pour s'assurer de leur fiabilité. Pour cela, nous proposons d'exploiter le Model-Checker SPIN [Hol03] pour vérifier les propriétés de sûreté et de vivacité des protocoles spécifiés. Notre choix de SPIN est motivé par le fait qu'il est

l'un des Model-Checker les plus puissants (particulièrement pour faire face au problème d'explosion combinatoire de l'espace d'état lors de la vérification) et les plus utilisés, dans les communautés industrielles et universitaires, pour détecter les défauts logiciels dans les conceptions de systèmes concurrents. Dans SPIN, une spécification formelle est construite à l'aide de Promela, qui est un langage impératif qui ressemble au langage C. Nous proposons donc, de générer automatiquement les propriétés LTL à vérifier et le code Promela à partir des spécifications Reo pour vérifier à l'aide de SPIN le protocole d'interaction spécifié.

Dans le reste du chapitre, nous présentons dans la section 7.2 les travaux connexes. Le langage Reo et sa sémantique formelle sont présentés la section 7.3. Notre proposition de formaliser les contraintes Reo sous forme de commandes gardées est présentée dans la section 7.4. Dans la section 7.5, nous présentons le passage de Reo à Promela et la vérification des propriétés avec SPIN. Une conclusion et un bilan sont présentés respectivement dans les sections 7.6 et 7.7.

7.2/ ÉTAT DE L'ART ET CONTRIBUTIONS

Dans la littérature, la vérification formelle des spécifications Reo à l'aide d'un certain nombre d'outils, excepté SPIN, est déjà traitée. Par exemple l'outil Vereofy [BBKK09a, BBK+10, BBKK09b], développé à l'Université de Dresde, permet d'analyser et de vérifier les connecteurs Reo. Il possède deux langages d'entrée : le langage de script Reo (RSL), utilisé pour spécifier le protocole de coordination et un langage sous forme de commandes gardées appelé *Constraint Automata Reactive Module Language (CARML)*, une version textuelle des automates à contraintes, utilisé pour spécifier le comportement des composants. Vereofy permet la vérification des propriétés temporelles exprimées en logiques de type LTL et CTL. Contrairement à Vereofy, dans notre travail, nous utilisons un seul langage, Promela, pour spécifier à la fois le comportement des composants et les protocoles de coordination. La sémantique de Reo avec les automates à contraintes a également été prise en compte dans [BSAR06] pour définir et vérifier la bisimulation et l'équivalence du langage entre les connecteurs Reo. Dans [Arb04, ABdBR07], les auteurs ont considéré les contraintes de temps et ont proposé une version temporisée des automates à contraintes pour vérifier par Model-Checking des propriétés CTL temporisées. Dans [Kem10, Kem12], les auteurs utilisent des automates à contraintes temporelles et présentent une approche basée sur SAT pour la vérification par Model-Checking borné de connecteurs de composants en temps réel. Une autre approche de vérification basée sur des solveurs SAT a été proposée dans [KSA+08], exploitant Alloy. Ce travail permet d'analyser les circuits Reo et de vérifier leurs propriétés exprimées sous forme de prédicats dans le langage de modélisation léger d'Alloy, basé sur une logique relationnelle de premier ordre. Dans [KKdV10, KKdV12], les auteurs ont proposé un cadre pour la vérification des circuits Reo à l'aide du jeu d'outils mCRL2 (développé au TU d'Eindhoven). Leur outil génère automatiquement des spécifications mCRL2 à partir de modèles graphiques Reo. La traduction de Reo en mCRL2 utilise la sémantique des automates à contraintes de Reo.

Contributions : En complément de cet ensemble d'outils existants, dans notre travail, nous avons choisi d'utiliser le Model-Checker SPIN, car (i) c'est l'un des outil d'analyse

et de vérification les plus avancés pour vérifier des propriétés LTL des systèmes concurrents, (ii) son langage Promela ressemble à notre spécification formelle, sous forme de commandes gardées, des Circuits Reo, et (iii) SPIN emploie un certain nombre de techniques pour faire face efficacement au problème de l'explosion de l'espace d'état. Nous n'avons pas connaissance d'un travail complet sur la vérification des circuits Reo utilisant SPIN. Pour cela, nous avons proposé un cadre formel pour la traduction des modèles Reo dans Promela. Cette traduction prend en charge conjointement le protocole de coordination décrit avec Reo, ainsi que ses propriétés à vérifier, pour générer leur implémentation en Promela. Notre proposition est implémentée comme un nouveau module dans le compilateur Reo, développé dans le groupe méthodes formelles du laboratoire CWI.

7.3/ REO

Reo est un langage de coordination basé sur les canaux, utilisé pour la construction compositionnelle des protocoles de coordination dans les SBC, en composants des connecteurs de base définis dans Reo. Ces derniers se composent de ports, de canaux et de nœuds. Ils implémentent des contraintes exogènes d'interaction entre les composants. Leur composition permet la conception de protocoles complexes pour définir les interactions entre un ensemble de composants. Dans cette section, nous montrons quelques connecteurs de base et présentons ensuite une syntaxe logique pour la formalisation de leurs contraintes exogènes.

7.3.1/ CONNECTEURS SYNCHRONES ET ASYNCHRONES

Le comportement d'un connecteur Reo est décrit par les séquences possibles de données circulant de manière synchrone à travers ses ports lorsque il interagit avec son environnement. Nous le décrivons formellement avec une formule de la logique de premier ordre dans la section suivante. Par ailleurs, chaque connecteur Reo est caractérisé par une *contrainte* interne, qui impose une restriction sur la séquence des données observables à ses ports. La figure 7.1 montre un certain nombre de connecteurs de base (appelés aussi primitive Reo), tous ayant des contraintes différentes. Nous expliquons intuitivement le comportement de ces connecteurs dans les paragraphes suivants.

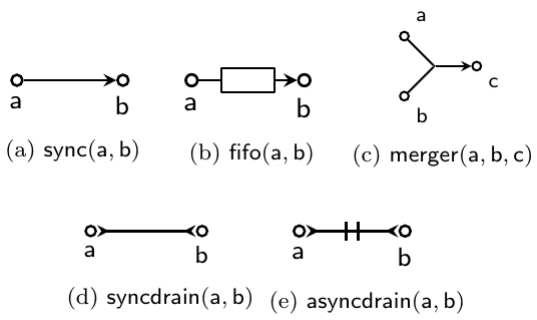


FIGURE 7.1 – Syntaxe graphique pour les primitives Reo.

Le connecteur *sync* dans la figure 7.1(a) représente un transfert synchrone de données

du port *a* vers le port *b*. Et *syncdrain* dans la figure 7.1 (d) modélise une activation synchrone des ports *a* et *b* sans tenir compte des données de ces ports. Ces deux connecteurs représentent des comportements synchrones.

Le connecteur *fifo* dans la figure 7.1(b) possède une mémoire interne qui contient l'élément de données obtenu à partir de son port d'entrée avant de le distribuer via son port de sortie. Les opérations de réception et de transmission de données sont asynchrones : le connecteur doit d'abord prendre un élément de données via son port d'entrée, avant de le libérer ultérieurement via son port de sortie. Le connecteur *asyncdrain* de la figure 7.1(e) nécessite que les ports *a* et *b* se déclenchent de manière asynchrone : soit une donnée est observée au port *a* ou une donnée est observée au port *b*, mais pas aux deux. Ces deux connecteurs représentent des comportements asynchrones.

Enfin, le connecteur *merger* permet de prendre au plus une donnée dans l'un de ses ports d'entrée pour la libérer d'une manière synchrone via son port de sortie. Notez que, comme l'*asyncdrain*, le *merger* impose une exclusion à ses ports d'entrée (un seul des ports se déclenche), et le déclenchement de ses ports d'entrée et de sortie est synchrone.

7.3.2/ Expression logique du comportement d'un connecteur Reo

Comportement d'un connecteur : Nous exprimons formellement le comportement d'un connecteur comme un prédicat reliant le flux de données à travers ses ports. Notez que nous étudions d'abord le comportement d'un connecteur dans son environnement idéal, c'est-à-dire que toutes les séquences d'entrée et de sortie sont possibles. Le comportement résultant est décrit par un ensemble de tuples de flux de données représentant le flux synchrone de données via les ports du connecteur. Ces données appartiennent à un domaine que nous appelons D. L'élément $*$ est ajouté à ce domaine, il désigne l'absence de données dans un port ou une mémoire. Nous utilisons D_* pour désigner le domaine de données résultant. Les ports et les mémoires sont définis comme des variables et prennent des valeurs dans le domaine D_*.

Exemple 7.1. *Le connecteur fifo a deux ports a et b, et une mémoire que nous notons par m. Étant donné $D_* = \{*, 1\}$, une exécution du protocole correspondant au connecteur fifo est représentée par la table 7.1.*

a	m	b
$*$	$*$	$*$
1	$*$	$*$
$*$	1	$*$
$*$	1	1
...	...	...

TABLE 7.1 – Exemple d'un comportement de *fifo* avec le domaine $D = \{1, *\}$

Expression logique : Nous notons par P, l'ensemble des variables associées aux ports, et par M l'ensemble des variables désignant les mémoires. Une mémoire stocke toujours le dernier élément de données issue d'un port. Pour chaque variable mémoire

$m \in M$, il existe une variable mémoire $m' \in M'$ qui définit l'état de la mémoire à l'état suivant du connecteur.

Les connecteurs définis par l'utilisateur pourraient être caractérisés par un prédicat ou une fonction. Les ensembles Q et F désignent respectivement l'ensemble des symboles de prédicat n-aire et des symboles de fonction n-aire. Un terme t, est soit une variable $p \in P$, $m \in M$ ou $m' \in M'$, une fonction n-aire $f \in F$, soit une constante $d \in C$. Donc une formule ϕ, formalisant le comportement d'un connecteur est construite par induction par :

$$\phi \ ::= t_1 = t_2 \mid B(t_1, ..., t_n) \mid \phi_1 \wedge \phi_2 \mid \neg\phi,$$

telle que $B \in Q$ est un symbole de prédicat.

Étant donné un port $p \in P$, la proposition de déclencher ou non p est formalisée comme une égalité entre p et les éléments de D_*. Nous considérons que p se déclenche si $p \neq *$. D'un autre côté, p ne se déclenche pas si $p = *$. En prenant $a, b \in P$, nous pouvons maintenant exprimer que a et b se déclenchent de manière synchrone avec la même donnée, avec la formule $a = b$.

Exemple 7.2. *Nous donnons deux exemples de connecteurs sync et fifo. Les formules suivantes expriment les contraintes concernant ces connecteurs.*

$$\text{sync}(a, b) = a = b \tag{7.1}$$

$$\begin{aligned}\text{fifo}(a, b) = {} & (m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee \\ & (m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee \\ & (m' = m \wedge a = * \wedge b = *)\end{aligned} \tag{7.2}$$

La contrainte pour le connecteur fifo a trois clauses. La première correspond au remplissage de la mémoire avec la donnée observée au port a ; la seconde vide le mémoire via le port b ; et la dernière correspond au cas où aucun port ne se déclenche, auquel cas la valeur dans la mémoire doit rester inchangée.

Comme indiqué précédemment, les protocoles peuvent être construits en composant des primitives (connecteurs de base). Dans le cas d'un connecteur composite (appelé également circuit), la contrainte résultante est définie comme la conjonction des contraintes des connecteurs sous-jacents.

Propriétés temporelles des connecteurs : Nous proposons de spécifier les propriétés temporelles sur les comportements des connecteurs à l'aide des formules LTL, respectant la syntaxe suivante :

$$\varphi ::= \ v = d \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

où v est un port ou une variable de mémoire dans $P \cup M$ et d est un élément du domaine de données D_*.

Nous introduisons plusieurs propriétés d'intérêt sur le circuit Reo. Étant donné p, m et m' respectivement le port, la mémoire et la variable de mémoire suivante, nous notons par $\mathbf{p}$ et $\mathbf{m}$, les propositions atomiques $p \neq *$ et $m' \neq m$ qui représentent, qu'une donnée est observée au niveau de p et m change de valeur. Dans les deux cas, on dit que p ou m se déclenche. Par ailleurs, dans le cas ou il n'y a pas de donnée observée au niveau d'un port p, on dit que p est silencieux et on le note $\neg\mathbf{p}$.

Exemple 7.3. *Comme exemple de formule LTL pour le connecteur synch, nous avons* $\Box(\mathbf{a} \iff \mathbf{b})$ *telle que* a *et* b *sont des variable des ports. Cette propriété indique la synchronisation de déclenchement entre les ports* a *et* b*. Et pour fifo, nous avons* $\Box(\mathbf{a} \implies \mathrm{X}(\neg\mathbf{a}\mathrm{U}\mathbf{b}))$ *qui indique que toujours quand* a *se déclenche, dans l'état suivant* a *devient silencieux jusqu'au déclenchement du port* b*.*

7.4/ Formalisation des contraintes Reo sous forme de commandes gardées

Le langage des contraintes présenté précédemment n'est pas adapté pour définir la traduction des contraintes Reo vers un programme Promela. Nous devons restreindre l'expressivité des contraintes afin de pouvoir compiler le protocole résultant dans un programme. Dans cette section, nous proposons de formaliser les contraintes logiques exprimant le comportement des connecteurs sous forme de *commandes gardées*. C'est une étape intermédiaire avant la traduction des contraintes Reo vers Promela, le langage cible. En effet, un programme Promela est composé d'un ensemble de processus définis avec *proctype*, où les instructions sont appelées commandes gardées, empruntées aux langages de commande gardés de E.W. Dijkstra [Dij75]. Une commande gardée est une déclaration de la forme $G \implies S$, où G est une proposition, appelée garde, et S une instruction, appelée commande.

Partitionnement des variables : Pour formaliser les contraintes des connecteurs sous forme de commandes gardées, nous avons besoin de partitionner l'ensemble des variables dans une contrainte ϕ, en variables d'entrée et de sortie. L'objectif derrière cette répartition est de préciser le rôle des variables associées aux ports et mémoires concernant l'entrée ou la sortie des données. Pour cela, nous considérons $V = P \cup M \cup M'$ l'ensemble des variables libres impliquées dans la contrainte ϕ. Soient I et O deux ensembles tels que $V = I \cup O$. Une variable $v \in V$ est une entrée si $v \in I \setminus O$ et une sortie si $v \in O \setminus I$. Pour la partie suivante, nous considérons que chaque contrainte ϕ d'un connecteur Reo est fournie avec une désignation de ses variables d'entrée et de sortie. Nous qualifions cette contrainte comme étant *dirigée*.

Commandes gardées : On note v^{in} une variable d'entrée et v^{out} une variable de sortie. Un terme d'entrée t^{in} fait référence à un élément de données $d \in D_*$, une fonction n-aires $f(t_1^{in}, ..., t_n^{in})$ dont les arguments sont des termes d'entrée $t_1^{in}, ..., t_n^{in}$, ou une variable v^{in}.

Un garde g est une conjonction de littéraux l définis comme suit :

$$l ::= B(t_1^{in}, ..., t_n^{in}) \mid t_1^{in} = t_2^{in} \mid v^{out} = d \mid \neg l$$

Une commande c est une conjonction d'égalités du type $v^{out} = t^{in}$. Une commande gardée gc est une conjonction d'implications $g \implies c$ où g est une garde et c est une commande.

Une formule ϕ est sous la forme de *commande gardée* si

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$$

où g_i sont des formules dans la langage des gardes et c_i sont des formules dans le langage des commandes.

Formule déterministe : Nous considérons qu'une formule sans quantificateur $\phi = \phi_1 \vee ... \vee \phi_n$ sous forme normale disjonctive et exprimée dans le langage des contraintes est *déterministe* si ϕ peut être formalisée avec la grammaire des commandes gardées telle que $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$, où $\wedge$ a priorité sur $\vee$, g_i sont des gardes, c_i sont des commandes et $g_i \wedge g_j \equiv \perp$ pour tous i, j où $i \neq j$.

Les propositions suivantes (les preuves sont détaillées dans [LCA18]), montrent que les formules logiques, décrites dans la section précédente, utilisées pour formaliser les contraintes des connecteurs Reo, peuvent être écrites sous forme de commandes gardées.

Proposition 7.1. *Une formule déterministe $\phi = \bigvee_i (g_i \wedge c_i)$ peut être écrite comme une commande gardée telle que :*

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$$

où i s'étend sur le nombre de disjonctions de ϕ.

Soient deux commandes gradées $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ et $\psi = \bigwedge_i (g'_i \implies c'_i) \wedge (\bigvee_j g'_j)$, nous formalison la composition par $\phi \wedge \psi$ telle que :

$$\phi \wedge \psi = \bigwedge_i (g_i \implies c_i) \bigwedge_i (g'_i \implies c'_i) \wedge (\bigvee_{i,j} g'_i \wedge g_j)$$

Proposition 7.2. *Soient deux formules déterministes ϕ et ψ, leur produit $\phi \wedge \psi$ est déterministe.*

Exemple 7.4. *Nous considérons maintenant un exemple de commandes gardées pour la primitive fifo. Cette dernière a une contrainte décrite dans la l'exemple 7.2, avec $a \in I$et $b \in O$, c'est-à-dire , a est un port d'entrée et b est un port de sortie. La formule d'un fifo est déterministe et avec le résultat de la proposition 7.1, nous pouvons écrire sa contrainte logique comme une commande gardée :*

$$\phi_{fifo} = (g_1 \implies c_1) \wedge (g_2 \implies c_1) \wedge (g_3 \implies c_2) \wedge (g_1 \vee g_2 \vee g_3)$$

telle que $g_1 := (a \neq * \wedge b = * \wedge m = *)$, $g_2 := (a = * \wedge b \neq * \wedge m \neq *)$, $g_3 := (a = * \wedge b = *)$, $c_1 := (m' = a \wedge m = b)$, $c_2 := m' = m$.

La formule pour le connecteur fifo a trois gardes. La première g_1 vérifie si son port source a est actif, auquel cas la commande c_1 remplit la mémoire avec les données observées au port a ; la deuxième garde g_2 vérifie si le port récepteur b du canal est actif, auquel cas sa commande c_1 vide la mémoire via le port b. Et la dernière g_3 vérifie si les deux ports a et b sont inactifs afin de déclencher la commande c_2 où les mémoires sont copiées.

7.5/ Formalisation des connecteurs Reo avec Promela

Dans la section précédente, nous avons détaillé la procédure pour écrire un protocole, décrit par une contrainte logique, sous forme d'une formule exprimée en commandes gardées. Ce formalisme permet de traduire facilement les protocoles décrits vers un programme Promela. Dans cette section, nous définissons cette traduction, en montrant tout d'abord comment spécifier les ports des connecteurs en Promela, de sorte à préserver leur propriété de synchronicité. Ensuite, nous décrivons les règles pour implémenter, en Promela, le protocole défini par un connecteur Reo. Et enfin nous montrons comment formaliser les propriétés LTL d'un protocole avec Promela.

7.5.1/ Formalisation des Ports

Dans Reo, un port est un emplacement où deux composants synchronisent et échangent des données. Dans Promela, nous implémentons un port Reo comme une paire de deux canaux Promela, chacun avec une taille d'une mémoire tampon d'une unité. Nous montrons que cette implémentation simule la synchronisation de l'échange de messages entre les composants dans Reo.

```
typedef port {
chan data = [1] of {Data};
chan trig = [1] of {int}; }
```

Listing 7.1 – Implémentation d'un port Reo en Promela

Comme exprimé dans le listing 7.1, un port a un canal de données et un canal de synchronisation. Le canal data est responsable du flux de données entre les extrémités d'entrée et de sortie d'un port. Le canal Promela trig assure un échange synchrone entre ces deux extrémités.

Deux actions peuvent être effectuées sur un tel port, put et take. La fonction put(q, a) remplit atomiquement le canal data de q avec la donnée a et bloque le canal trig, en attente de synchronisation avec le composant du côté sortie de q. La fonction take(q, a) notifie d'une manière atomique, en remplissant le canal trig, qu'il existe un composant prêt à prendre des données et bloque le canal data jusqu'à ce qu'une donnée puisse être prise dans la variable a.

Propriétés temporelles liées aux ports : Nous décrivons deux propriétés temporelles qui reflètent le comportement synchrone d'un port dans une implémentation Promela asynchrone. Nous considérons qu'un port se déclenche (*fires*) chaque fois qu'une donnée est échangée entre la partie d'entrée et la partie de sortie. Si un port ne se déclenche pas, il est silencieux (*silent*). La propriété de déclenchement se produit chaque fois qu'un port a à la fois une demande d'entrée et une demande de sortie : les deux tampons sont pleins. Dans le cas contraire, le port respecte la propriété silencieux. Les macros liées à ces propriétés sont définies dans le listing 7.2. En outre, nous définissons également des macros liées aux mémoires, utiles pour définir des propriétés temporelles pour les connecteurs dotés de mémoires. Ainsi, nous considérons qu'une mémoire est pleine chaque fois qu'elle contient des données et elle est vide sinon (voir le listing 7.2).

```
#define p_fires (
!(len(p.data) == 0) && !(len(p.trig) == 0) &&
```

```
X((len(p.data) == 0) || (len(p.trig) == 0)))

#define p_silent (! p_fires)


#define m_full  (!(len(m)==0))
#define m_empty  ((len(m)==0))
```

Listing 7.2 – Définition de macros pour les ports et les mémoires

7.5.2/ FORMALISATION DU PROTOCOLE DÉFINI PAR UN CONNECTEUR

Dans Promela, nous définissons le protocole associé à un connecteur Reo comme un *proctype* avec un ensemble d'arguments correspondant à tous les ports de son interface.

```
proctype Protocol(port p1;...){ ...}
```

Formalisation des variables : Pour chaque variable de port $p \in P$, nous définissons une instance globale de la structure de port définie au paragraphe précédent, avec le même nom que la variable. Pour chaque variable mémoire $m \in M$, nous définissons un canal local de taille 1. En outre, les variables de mémoire m et m' sont reliées au même nom dans Promela, car elles se réfèrent au même emplacement de mémoire.

Étant donné une formule ϕ, son ensemble de variables de port P et les variables de mémoire M, la fonction $[\![\cdot]\!]_{\mathcal{I}} : P \cup M \to \mathcal{P}$ relie chaque variable de port $p \in P$ et chaque variable de mémoire $m \in M$ à une définition dans Promela, telle que :

$$[\![p]\!]_{\mathcal{I}} = \textbf{port } p \textbf{ ;}$$
$$[\![m]\!]_{\mathcal{I}} = \textbf{chan } m \textbf{ = [1] of } \{\textbf{Data}\} \textbf{ ;}$$

Formalisation des autres termes logiques : Les symboles constants sont reliés sur leur domaine correspondant. Promela ne prend en charge que quelques types de données, nous supposons que les constantes obtiennent une interprétation dans l'un de ces types de données (entier, booléen, flottant, caractères). Les symboles de fonction sont reliés aux procédures (*inligne*) dans Promela. Pour chaque fonction utilisée dans Reo, nous supposons qu'une procédure avec le même nom est fournie dans Promela. La formalisation des règles du passage de Reo vers Promela sont décrites dans [LCA18].

Commandes gardées : Sur la base du résultat de la proposition 7.1, nous considérons la contrainte exprimant le protocole d'un connecteur Reo sous sa forme de commande gardée. Nous utilisons $\mathcal{GC}$ pour l'ensemble des commandes gardées. Une commande gardée $g \to c \in \mathcal{GC}$ a une interprétation naturelle dans Promela comme une mise à jour conditionnelle. Ainsi, la garde g est traduite comme une condition sur l'état des canaux associés aux ports et aux mémoires, tandis que la commande c est une mise à jour de l'état de ces canaux . Concrètement, les instructions les plus courantes dans les gardes sont `full(p.data)`, `full(p.trig)` et `full(m)`, qui vérifient respectivement si un port `p` a une demande de données entrante, une demande de données sortante ou si la mémoire

`m` est pleine. La négation de ces déclarations est également couramment utilisée par les gardes. Et dans la commande, nous procédons à la mise à jour des ports et des mémoires, ce qui correspond aux instructions du type `take(p, d)`, `put(p, d)`, `m? D` , ou `m! d`, où `d` est une variable locale. Les détails de la traduction se trouvent dans notre papier [LCA18].

La structure générique d'un programme Promela obtenu à partir d'une formule déterministe avec n commandes gardées est présentée dans Listing 7.3.

```
proctype Protocol(port p1;...){
/* p1: input, ... */
/* Memory declaration */
chan m = [1] of {Data}; ...
/* Initial state */
m!0; ...
/* Local variables */
int _m;int _p1 ;...
/* Guarded commands */
do
:: (guard_1) ->command_1
:: ...
:: (guard_n) ->command_n
od }
```

Listing 7.3 – La structure générique d'un protocole

7.5.3/ FORMALISATION DES PROPRIÉTÉS LTL

Nous proposons une traduction des propriétés LTL spécifiées dans Reo vers Promela, de sorte que les propriétés de synchronie, intrinsèques aux échanges de données dans Reo, soient préservées. La table suivante présente quelques propriétés des connecteurs Reo formalisées avec la LTL et Promela.

Propriétés	Formules LTL
p_fires	len(p.data) != 0 && len(p.trig)!=0 && X(len(p.data)==0 || len(p.trig)==0)
p_silent	!(p_fires)
m_full	len(m.data)!=0
m_empty	len(m.data)==0
m_fires	m_full && X(m_empty)) || m_empty && X(m_full)
m_silent	!(m_fires)
p1_before_p2	(p1_fires -> <> (p2_fires))
p1_then_p2	(p1_fires -> X(silent U p2_fires))
sync_p1_p2	[] (p1_then_p2 \/ p2_then_p1)
async_p1_p2	[] (p1_before_p2 \/ p2_before_p1)

TABLE 7.2 – Formalisation en Promela des propriétés LTL des connecteurs Reo

La propriété `p_silent` est définie comme la négation de la propriété de déclenchement et représente tous les états de non-déclenchement du port p. La propriété `silent` indique la

conjonction de toutes les propriétés silencieuses pour tous les ports. Notez que la mise à jour de la mémoire interne est toujours autorisée.

Les deux propriétés `p1_before_p2` et `p1_then_p2` expriment un déclenchement asynchrone pour les ports p_1 et p_2. Cette dernière propriété est plus stricte, car la propriété `silent` requiert qu'entre le déclenchement des ports p_1 et p_2, aucun autre port ne se déclenche.

Nous définissons la propriété synchrone dans Promela, `sync_p1_p2`, comme une relation binaire entre deux ports p_1 et p_2. Les deux ports sont synchrones s'il est toujours vrai que p_1 se déclenche puis p_2 se déclenche (ou l'inverse) et toutes les étapes entre les déclenchements satisfont la propriété `silent`.

Nous avons démontré la pertinence de notre approche dans une étude de cas impliquant la conception et la vérification d'un protocole de coordination d'un SBC ferroviaire dans [LCA18, LCA20].

7.6/ Conclusion et perspectives

Nous avons présenté dans ce chapitre notre de travail de collaboration avec l'équipe du Professeur Arbab du CWI, au sujet du passage des spécification Reo vers Promela pour vérifier des propriétés LTL avec SPIN sur des protocoles de coordination. Pour cela, nous avons défini des conditions suffisantes pour traduire un protocole synchrone spécifié dans Reo dans un programme asynchrone implémenté dans Promela afin de vérifier un certain nombre de propriétés LTL liées aux échanges de données entre les ports des connecteurs. Nous avons défini des conditions pour exprimer les contraintes de connecteurs sous forme de commandes gardées, pour faciliter leur formalisation avec Promela. Certaines formules temporelles sur le comportement des connecteurs Reo sont prises en compte, telles que *firing*, *silent*, *synchronous* ou *asynchronous*. Nous avons formalisé ces propriétés dans le cas d'un connecteur synchrone (Reo) et donné leur implémentation asynchrone dans Promela. Cette approche a été implémentée comme un nouveau module dans le compilateur actuel de Reo.

Comme perspective, à ce travail, nous nous intéressons actuellement à la définition d'une classe de propriétés LTL pouvant être préservées par la composition des connecteurs Reo. L'objectif est de réduire le coût de la vérification lors de la construction de circuits complexes.

7.7/ Bilan

Les résultats de ce chapitre concernent les éléments suivants :

1. **Projet** : le projet CHRYSALIDE (financement pour le soutien aux projets de recherche), université Bourgogne Franche-Comté, 2019
2. **Thèse de doctorat** : participation à l'encadrement de la thèse de Benjamin Lion (CWI), dans la période fin 2017-2018.
3. **Publications** : [LCA18, LCA20]

IV

Conclusion et Perspectives

8

CONCLUSION ET PERSPECTIVES

8.1/ CONCLUSION

Ce document présente une synthèse de mes principales activités de recherche effectuées depuis 2005, l'année de mon recrutement en tant maître de conférences. Dans mes travaux, les différentes contributions répondent à un objectif commun : conception rigoureuse des SBC critiques en exploitant des notations semi-formelles et des approches formelles. Donc, la question du développement des SBC a été étudiée sous plusieurs points de vues, ce qui nous a permis d'obtenir plusieurs résultats :

— Proposition des approches combinant SysML et le formalisme des automates d'interface pour : la modélisation avec SysML, sans ambiguïté, de l'architecture des SBC ; la construction incrémentale des SBC guidée par SysML, en vérifiant des relations de composition et de raffinement entre les composants ; l'adaptation des composants modélisés avec SysML.

— Augmentation de l'expressivité des interfaces des composants en proposant des contrats comportementaux pour les spécifier, soit avec le formalisme de la méthode B, soit avec les automates d'interface enrichis avec la sémantique des opérations, pour vérifier l'interopérabilité, la substituabilité, et l'adaptation des composants.

Par ailleurs, ces dernières années, grâce à des séjours scientifiques effectués au centre de recherche CWI d'Amsterdam, je me suis intéressé aux problématiques de coordination dans les systèmes à composants/services. Cela m'a permis de travailler avec le Professeur F.Arbab du CWI et son équipe, sur le passage du langage Reo vers Promela (SPIN) pour vérifier des propriétés des protocoles REO. Un premier résultat a été obtenu suite à cette collaboration [LCA18]. Dans un autre cadre, j'ai pu aussi m'ouvrir vers d'autres domaines (véhicules communicants et réseaux de capteurs), en collaborant avec un collègue d'une autre équipe (AND) de mon département de recherche, et également avec un collègue de l'université d'Ottawa au Canada. J'ai eu l'occasion, grâce à cette collaboration, d'apporter mes compétences dans le domaine des méthodes formelles pour répondre à des problématiques liées aux nouveaux domaines étudiés, ce qui nous a permis d'avoir des premiers résultats [CBM17a, CBM17b, MBMC19].

8.2/ PERSPECTIVES DE RECHERCHE

En plus des perspectives présentées dans les chapitres précédents, nous proposons ici quelques directions générales de recherche que nous envisageons.

8.2.1/ Conception des CPS, correct-par-construction, en exploitant les langages SysML/Marte/pCCSL et les contrats comportementaux des composants

Une perspective à cours terme et en lien directe avec nos travaux consiste à enrichir les modèles SysML utilisés dans nos travaux pour la conception des SBC, dans le but de modéliser les systèmes cyber-physiques (CPS). L'objectif est de définir et de mettre en œuvre une nouvelle approche de modélisation et de vérification formelle des CPS critiques qui s'appuie, d'une part sur le formalisme SysML comme un langage pivot de modélisation, et d'autre part sur le formalisme des contrats comportementaux (présenté dans le chapitre 5) pour vérifier formellement des propriétés sur les CPS.

Au niveau modélisation, il serait intéressant de s'inspirer du travail présenté dans [HJGD18], qui propose une approche de modélisation basée sur les langages SysML/-MARTE/pCCSL pour capturer les différents aspects d'un CPS tels que : la structure, le comportement, les contraintes temporelles, et les propriétés non fonctionnelles. Une première contribution par rapport à ce travail serait de considérer, en plus, le diagramme des exigences SysML (qui n'est pas pris en compte), en vue de l'exploiter pour la vérification des propriétés. Au niveau formel, nous projetons d'étendre le travail présenté dans le chapitre 5, concernant les contrats comportementaux des composants orientés objet, pour prendre en compte les aspects temporel et distribué des CPS. Il faut ensuite définir un cadre formel pour vérifier l'interopérabilité et le raffinement des composants dans un CPS. Et enfin, il serait intéressant, d'étudier la faisabilité de l'approche, et de montrer la construction correcte des CPS, de la conception à l'implémentation, en proposant une implémentation des contrats des composants avec le langage ADA [ISO12], et son extension SPARK [Ada14], utilisée pour la vérification formelle. En effet, SPARK est une extension du langage ADA, avec des annotations formelles basées sur les contrats (donc plus appropriée pour implémenter les contrats des composants), qui sont considérées lors de la vérification formelle. Sur cette partie, je projette de collaborer avec S. Mouelhi (ancien doctorant), enseignant chercheur à l'ECE Paris, qui a également travaillé dans l'industrie, dans le domaine de la validation des logiciels embarqués dans les transports ferroviaires.

Depuis janvier 2020, je co-encadre une thèse, en collaboration avec une collègue de l'université USTHB d'Alger, qui traite une partie des problématiques énoncées.

8.2.2/ Assemblage et adaptation des composants interagissant d'une manière asynchrone

Dans les travaux précédents, nous nous sommes focalisés sur la vérification de la compatibilité lors de l'assemblage des composants qui interagissent d'une manière synchrone. Actuellement, nous explorons et analysons les problématiques liées à l'assemblage des composants interagissant d'une manière asynchrone. Dans ce cas, les composants interagissent généralement par l'intermédiaire d'un buffer non borné, et les approches de vérification de la compatibilité sont beaucoup plus complexes que dans le cas synchrone, car en général elles conduisent vers un espace d'état infini, et donc sont indécidables. Plusieurs travaux traitant cette problématique proposent de borner la taille du buffer. Néanmoins, ces propositions sont contraignantes, car des problèmes peuvent surgirent lorsque, par exemple, la taille du buffer change durant l'exécution. Ainsi, la contrainte

principale pour ces approches est liée au choix de la taille appropriée du buffer afin de vérifier la compatibilité. Pour contourner ce problème, des approches ont été proposées, par exemple en considérant une classe de systèmes présentant une propriété de synchronicité [OSB13, BB14], ou en limitant le nombre de cycles de communication [BE14]. Ces approches permettent de faire des vérifications avec un buffer non borné, néanmoins avec des restrictions sur les systèmes à considérer ou sur la nature de leurs interactions.

Dans le cadre du projet Tassili SYSVAP (2013-2016), nous avons réalisé un premier travail [DBMC17] en collaboration avec des collègues de l'université USTHB d'Alger, pour proposer une approche formelle afin de répondre aux problématiques décrites précédemment. Notre idée repose sur la définition d'une abstraction finie du produit asynchrone entre les automates d'interface modélisant les comportements infinis des composants, appelé produit asynchrone de couverture. Ce dernier est inspiré du graphe de couverture des réseaux de Petri. Ce faisant, la vérification de la compatibilité est effectuée sur un espace d'état fini, en analysant des chemins cycliques. Cependant, pour compléter cette démarche, il reste à répondre à quelques questions liées à la considération de l'approche optimiste lors de la vérification de la compatibilité et à l'adaptation des composants lorsque leurs interactions présentent des incohérences.

8.2.3/ Analyse et vérification des protocoles de communication dans les véhicules communicants

Dans [CBM17b], nous avons présenté une proposition qui vise à surmonter les inconvénients du protocole MQTT (Message Queuing Telemetry Transport) [Loc10] dans le contexte des véhicules connectés. MQTT est l'un des protocoles d'échange de données les plus prometteurs pour les applications IoT (Internet Of Things). Les inconvénients considérés sont principalement liés à l'énorme volume de données, inutiles, envoyées par les véhicules connectés aux infrastructures automobiles. Pour remédier à cette limitation, nous avons tout d'abord proposé une variante de MQTT, nommée MQTT-CV, qui vise à alléger les infrastructures automobiles en réduisant le volume de données qui leur sont envoyées, en établissant un filtre au niveau du protocole. Ensuite, pour garantir la fiabilité du MQTT-CV, qui est un système critique pour la sécurité des conducteurs, nous l'avons formellement analysé en utilisant le Model-Checker SPIN, pour s'assurer de son bon fonctionnement. De plus, nous avons montré l'efficacité de notre solution par rapport à la consommation de la CPU et la RAM, comparée au protocole MQTT, grâce à notre implémentation du MQTT-CV et aux expérimentations que nous avons menées en exploitant des données réelles. Dans les travaux futurs, nous projetons d'améliorer le MQTT-CV et le rendre capable de gérer des conditions plus complexes que les infrastructures automobiles peuvent exprimer sur les données échangées. De plus, nous prévoyons de considérer ces conditions au niveau de la vérification formelle. Par ailleurs, nous projetons également de considérer les propriétés de qualité de service concernant ce protocole (liées à la délivrance des messages). En effet, MQTT présente trois niveaux de qualité de service, et certains travaux, comme dans [Azi16, RKR19], ont démontré l'existence de certaines ambiguïtés, soit au niveau de la qualité de service numéro 3, susceptibles de compromettre la sécurité du protocole (dans [Azi16]), soit au niveau de l'implémentation des niveaux de la qualité de service qui peut générer des problèmes d'interopérabilité entre les implémentations (dans [RKR19]). Ces ambiguïtés soulèvent de nouvelles questions de recherche que nous envisageons de traiter dans les années à venir.

BIBLIOGRAPHIE

[AA13] Idir Aït-Sadoune and Yamine Aït Ameur. Stepwise development of formal models for web services compositions : Modelling and property verification. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 10 :1–33, 2013.

[AA15] Idir Aït-Sadoune and Yamine Aït Ameur. Formal modelling and verification of transactional web service composition : A refinement and proof approach with event-b. In Bernhard Thalheim, Klaus-Dieter Schewe, Andreas Prinz, and Bruno Buchberger, editors, *Correct Software in Web Applications and Web Services*, Texts and monographs in symbolic computation, pages 1–27. Springer, 2015.

[AAA06] J. Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking component composability. In Welf Löwe and Mario Südholt, editors, *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, volume 4089 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2006.

[AAA08] Pascal André, Gilles Ardourel, and J. Christian Attiogbé. Composing components with shared services in the kmeliamodel. In Cesare Pautasso and Éric Tanter, editors, *Software Composition, 7th International Symposium, SC 2008, Budapest, Hungary, March 29-30, 2008. Proceedings*, volume 4954 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2008.

[AAAL10a] Pascal André, Gilles Ardourel, J. Christian Attiogbé, and Arnaud Lanoix. Using assertions to enhance the correctness of kmelia components and their assemblies. *Electron. Notes Theor. Comput. Sci.*, 263 :5–30, 2010.

[AAAL10b] Pascal André, Gilles Ardourel, J. Christian Attiogbé, and Arnaud Lanoix. Using event-b to verify the kmelia components and their assemblies. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, page 410. Springer, 2010.

[ABC+02] Jörg Ackermann, Frank Brinkop, Stefan Conrad, Peter Fettke, Andreas Frick, Elke Glistau, Holger Jaekel, Otto Kotlar, Peter Loos, Heike Mrech, Erich Ortner, Ulrich Raape, Sven Overhage, Stephan Sahm, Andreas Schmieten, Thorsten Teschke, and Klaus Turowski. Standardized specification of business components. *Memorandum of the working group 5.10.3, Component Oriented Business Application Systems*, 2002.

[ABdBR07] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling*, 6(1) :59–82, Mar 2007.

[Abr96a] J.-R. Abrial. *The B Book*. Cambridge University Press - ISBN 0521-496195, 1996.

[Abr96b] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[acc] Acceleo. http://www.eclipse.org/acceleo.

[Ada14] AdaCore. SPARK 2014 reference manual, 2014.

[Adm98] National Highway Traffic Safety Administration. Federal Motor Vehicle Safety Standards, 1998.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3) :213–249, July 1997.

[AH01] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[AH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, NATO Science Series : Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Procs. of the 9th Int. Conf. on Concurrency Theory*, pages 163–178, London, UK, 1998. Springer-Verlag.

[Ali18] Sajjad Ali. Formal verification of sysml diagram using case studies of real-time system. *ISSE*, 14(4) :245–262, 2018.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.

[Arb04] Farhad Arbab. Reo : A channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3) :329–366, June 2004.

[atl] ATL : Atlas Transformation Language. https://eclipse.org/atl/.

[Azi16] Benjamin Aziz. A formal model and analysis of an iot protocol. *Ad Hoc Networks*, 36 :49 – 57, 2016.

[BB14] Samik Basu and Tevfik Bultan. Automatic verification of interactions in asynchronous systems with unbounded buffers. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 743–754, 2014.

[BBB+11] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3) :41–48, May 2011.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1) :45–54, 2005.

[BBG+06] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, pages 193–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BBK+10] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and verification of systems with exogenous coordination using vereofy. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 97–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BBKK09a] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. Formal verification for components and connectors. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Formal Methods for Components and Objects*, pages 82–101, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BBKK09b] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A uniform framework for modeling and verifying components and connectors. In John Field and Vasco T. Vasconcelos, editors, *Coordination Models and Languages*, pages 247–267, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BCGZ01] Nadia Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro. Coordination models : A guided tour. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents : Models, Technologies, and Applications*, pages 6–24. Springer, 2001.

[BCHM15a] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Compatibility verification of sysml blocks using hierarchical interface automata. In *12th International Symposium on Programming and Systems (ISPS)*, pages 313 – 322, Alger, Algeria, apr 2015. IEEE.

[BCHM15b] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Exploitation de la hiérarchie pour la vérification de la compatibilité entre les blocs sysml. In *CAL 2015, 9ème conférence francophone sur les architectures logicielles*, Hammamet, Tunisia, may 2015.

[BCHM16a] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Exploitation de la hiérarchie pour la véification de la compatibilité des blocs sysml. *Revue des Nouvelles Technologies de l'Information*, RNTI-L(8) :99 – 118, 2016.

[BCHM16b] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. A model-driven approach to adapt sysml blocks. In *Information and Software Technologies - 22nd International Conference, ICIST 2016, Druskininkai, Lithuania, October 13-15, 2016, Proceedings*, pages 255–268, 2016.

[BCHM19] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Sysml model-driven approach to verify blocks compatibility. *IJCAET*, 11(2) :206–231, 2019.

[BCJ05] Françoise Bellegarde, Samir Chouali, and Jacques Julliand. Refinemant verification of fair transition systems can contribute to PLTL model checking. In

3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings, pages 166–175. IEEE Computer Society, 2005.

[BCZ15] Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors. *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*. Springer, 2015.

[BDL05] O. Barais, L. Duchien, and A. . Le Meur. A framework to specify incremental software architecture transformations. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 62–69, Aug 2005.

[BE14] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. *STTT*, 16(2) :127–146, 2014.

[BGMPG99] Gérard Baille, Philippe Garnier, Hervé Mathieu, and Roger Pissard-Gibollet. *The INRIA Rhône-Alpes CyCab*. Technical report, INRIA, 1999. Describes the package natbib.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, pages 38–45, July 1999.

[BMC+12] Erwan Bousse, David Mentré, Benoît Combemale, Benoît Baudry, and Takaya Katsuragi. Aligning SysML with the B Method to provide V&V for systems engineering. In *Procs. of MoDeVVa'12*, pages 11–16, NY, USA, 2012. ACM.

[BMFN01] Jo Barnes, Andrew Morris, Brian Fildes, and S.V. Newstead. Airbag effectiveness in real world crashes. *Road Saf. Res. Policing, Educ. Conf.*, 2001.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On weak modal compatibility, refinement, and the mio workbench. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 175–189, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Bou16] Hamida Bouaziz. *Adaptation of SysML Blocks and Verification of Temporal Properties. (Adptation des Blocs sysML et verification des propriétés temporelles)*. PhD thesis, University of Franche-Comté, France, 2016.

[BP06] Antonio Brogi and Razvan Popescu. Automated generation of BPEL adapters. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, pages 27–39, 2006.

[BR08] Purandar Bhaduri and S. Ramesh. Interface synthesis and protocol conversion. *Formal Asp. Comput.*, 20(2) :205–224, 2008.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2) :75 – 113, 2006. Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03).

[BSP99] R. Bastide, O. Sy, and P. A. Palanque. Formal specification and prototyping of CORBA systems. In *ECOOP '99 : Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 474–494. Springer-Verlag, 1999.

[BT07] Clark Barrett and Cesare Tinelli. Cvc3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 298–302, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CBM17a] Samir Chouali, Azzedine Boukerche, and Ahmed Mostefaoui. Ensuring the reliability of an autonomous vehicle : a formal approach based on component interaction protocols. In *20th International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2017)*, pages 317 – 321, Miami Beach, United States, nov 2017. Association for Computing Machinery (ACM).

[CBM17b] Samir Chouali, Azzedine Boukerche, and Ahmed Mostefaoui. Towards a formal analysis of mqtt protocol in the context of communicating vehicles. In *15th International Symposium on Mobility Management and Wireless Access (MobiWAC 2017)*, pages 129 – 136, Miami, FL, United States, nov 2017. Association for Computing Machinery (ACM).

[cbt] IEEE Std. 1474.1-1999 for Communications-Based Train Control (CBTC) Performance and Functional Requirements (rev. 2004). pages 1–45.

[CCM12a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Formalizing and Verifying Compatibility and Consistency of SysML Blocks. *ACM SIGSOFT Software Engineering Notes*, 37(4) :1–8, 2012.

[CCM12b] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Vérification de la consistance et de la compatibilité entre blocs sysml. In *CAL 2012, 6ème conférence francophone sur les architectures logicielles*, Montpellier, France, Mai 2012.

[CCM13a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Incremental modeling of system architecture satisfying sysml functional requirements. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 79–99, 2013.

[CCM13b] Samir Chouali, Oscar Carrillo, and Hassan Mountassir. Specifying system architecture from sysml requirements and component interfaces. In *Software Architecture - 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings*, pages 348–352, 2013.

[CCM14a] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Incremental modeling of system architecture satisfying SysML functional requirements. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *FACS 2013, 10th Int. Symposium on Formal Aspects of Component Software, Revised Selected Papers*, volume 8348 of *LNCS*, pages 79–99, Nanchang, China, 2014. Springer.

[CCM14b] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. Modélisation incrémentale d'une architecture de système satisfaisant des exigences fonctionnelles. In *CAL 2014, 8ème conférence francophone sur les architectures logicielles*, Paris, France, Juin 2014. Résumé étendu, une page.

[CD01] J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[CDH+10] Samir Chouali, Julien Dormoy, Ahmed Hammad, Jean-Michel Hufflen, Sebti Mouelhi, Olga Kouchnarenko, Hassan Mountassir, and Bruno Tatibouët. Assemblage des composants digne de confiance : de l'ingénierie des besoins aux spécifications formelles. *Génie Logiciel*, 95 :13–18, dec 2010.

[cen11] Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, CENELEC EN 50128. 2001 (rev. 2011).

[CEP00] L.A. Cortes, P. Eles, and Zebo Peng. Verification of embedded systems using a Petri net based representation. In *System Synthesis, 2000. Proceedings. The 13th International Symposium on, Madrid, Spain*, pages 149–155, 2000.

[CFN05] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Assembling components with behavioural contracts. *Annales des Télécommunications*, 60(7-8) :989–1022, 2005.

[CFTV00] C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending corba interfaces with π-calculus for protocol compatibility. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 208–225, Washington, DC, USA, 2000. IEEE Computer Society.

[CH03] K. Czarnecki and s. Helsen. Classification of model transformation approaches. In *OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim, USA, 2003.

[CH11a] Samir Chouali and Ahmed Hammad. Formal Verification of Components Assembly Based on SysML and Interface Automata. *ISSE*, 7(4) :265–274, 2011.

[CH11b] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on sysml and interface automata. *ISSE*, 7(4) :265–274, 2011.

[CHM13] Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Assembling components using sysml with non-functional requirements. *Electr. Notes Theor. Comput. Sci.*, 295 :31–47, 2013.

[Cho06] Samir Chouali. Cooperation between the b method and the automata theory to check the component interoperability. In *FACS'2006, 3rd Int. Workshop on Formal Aspects of Components Software*, pages 211–227, Prague, Czech Republic, sep 2006.

[CHS06] Samir Chouali, Maritta Heisel, and Jeanine Souquières. Proving component interoperability with B refinement. *Electr. Notes Theor. Comput. Sci.*, 160 :157–172, 2006.

[CJMB05] Samir Chouali, Jacques Julliand, Pierre-Alain Masson, and Françoise Bellegarde. Pltl-partitioned model checking for reactive systems under fairness assumptions. *ACM Trans. Embedded Comput. Syst.*, 4(2) :267–301, 2005.

[CLR+09] Zhenbang Chen, Zhiming Liu, Anders P. Ravn, Volker Stolz, and Naijun Zhan. Refinement and verification in component-based model-driven design. *Science of Computer Programming*, 74(4) :168 – 196, 2009. Special Issue on the Grand Challenge.

[CMC+08] Ermeson Carneiro, Paulo Maciel, Gustavo Callou, Eduardo Tavares, and Bruno Nogueira. Mapping SysML State Machine Diagram to Time Petri Net for Analysis and Verification of Embedded Real-Time Systems with Energy Constraints. *Electronics and Micro-electronics, International Conference on Advances in*, 0 :1–6, 2008.

[CMM10a] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation des Protocoles des Composants par les Automates d'Interface. In *AFADL'10, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 253–266, Poitiers, France, June 2010.

[CMM10b] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting component behaviours using interface automata. In *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '10, pages 119–122, Washington, DC, USA, 2010. IEEE Computer Society.

[CMM10c] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting components using interface automata enriched by action semantics. In *Proceedings of 1st International Conference on Formal Verification of Object-Oriented Software*, pages 7–21, Paris, France, 2010.

[CMM10d] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Assembly of components based on interface automata and uml component model. In Khalil Drira, editor, *CAL'10, 4e Conf. Francophone sur les Architectures Logicielles*, volume RNTI-L-5 of *RNTI, Revue des Nouvelles Technologies de l'Information*, pages 73–85, Pau, France, mar 2010.

[CMM10e] Samir Chouali, Hassan Mountassir, and Sebti Mouelhi. An I/O automata-based approach to verify component compatibility : Application to the cycab car. *Electr. Notes Theor. Comput. Sci.*, 238(6) :3–13, 2010.

[CMM12a] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation sémantique des protocoles des composants par les automates d'interface. *TSI, Technique et Science Informatiques*, 31(6) :769–796, 2012.

[CMM12b] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adaptation sémantique des protocoles des composants par les automates d'interface. *Technique et Science Informatiques*, 31(6) :769–796, 2012.

[CMM18] Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Component design and adaptation based on behavioral contracts. In El Hassan Abdelwahed, Ladjel Bellatreche, Djamal Benslimane, Matteo Golfarelli, Stéphane Jean, Dominique Méry, Kazumi Nakamatsu, and Carlos Ordonez, editors, *New Trends in Model and Data Engineering - MEDI 2018 International Workshops, DETECT, MEDI4SG, IWCFS, REMEDY, Marrakesh, Morocco, October 24-26, 2018, Proceedings*, volume 929 of *Communications in Computer and Information Science*, pages 217–230. Springer, 2018.

[CMP06] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. *L'OBJET*, 12(1) :9–31, 2006.

[CPS06a] Carlos Canal, Pascal Poizat, and Gwen Salaün. Adaptation de composants logiciels une approche automatisée basée sur des expressions régulières de vecteurs de synchronisation. In *CAL*, pages 31–39, 2006.

[CPS06b] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *FMOODS 2006, Bologna, Italy*, pages 63–77, 2006.

[CPS06c] Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing behavioural mismatch in software composition. In *The Proc. of Inter. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 63–77. Springer-Verlag, 2006.

[CPS08a] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Software Eng.*, 34(4) :546–563, 2008.

[CPS08b] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 34 :546–563, July 2008.

[CS14] Carlos Canal and Gwen Salaün. Adaptation of asynchronously communicating software. In *Service-Oriented Computing - 12th International Conference, ICSOC 2014, Paris, France, November 3-6, 2014. Proceedings*, pages 437–444, 2014.

[CSVC11] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5) :593–615, 2011.

[DA18] K. Dokter and F. Arbab. Treo : Textual Syntax for Reo Connectors. *ArXiv e-prints*, June 2018.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5) :109–120, 2001.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, pages 108–122, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Dav10] Brett Davis. Mercedes-benz celebrates 30 years of using airbag technology, 2010.

[DBMC17] D. Dahmani, Mohand Cherif Boukala, Hassan Mountassir, and Samir Chouali. Compatibility control of asynchronous communicating systems with unbounded buffers. In Daniel Moldt, Lawrence Cabac, and Heiko Rölke, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'17)*, volume 1846 of *CEUR Workshop Proceedings*, pages 69–84. CEUR-WS.org, 2017.

[DBMN08] Marlon Dumas, Boualem Benatallah, and Hamid R. Motahari Nezhad. Web service protocols : Compatibility and adaptation. *IEEE Data Engineering Bulletin*, 31(3) :40–44, 2008.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, August 1975.

[DLL+10] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata : A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems : Computation and Control*, HSCC '10, page 91–100, New York, NY, USA, 2010. Association for Computing Machinery.

[dLVA04] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in atom3. *Software and System Modeling*, 3(3) :194–209, 2004.

[DOP17] Iulia Dragomir, Iulian Ober, and Christian Percebois. Contract-based modeling and verification of timed safety requirements within sysml. *Software and Systems Modeling*, 16(2) :587–624, 2017.

[Dro03] R. G. Dromey. From requirements to design : formalizing the key steps. In *First International Conference onSoftware Engineering and Formal Methods, 2003.Proceedings.*, pages 2–11, Sep. 2003.

[Dro07] R. G. Dromey. Engineering large-scale software-intensive systems. In *2007 Australian Software Engineering Conference (ASWEC'07)*, pages 4–6, April 2007.

[ES09] Sima Emadi and Fereidoon Shams. Mapping annotated use case and sequence diagrams to a petri net notation for performance evaluation. In *Computer and Electrical Engineering, 2009. ICCEE'09. Second International Conference on*, volume 2, pages 68–71. IEEE, 2009.

[Eur10] European Railway Agency. ERTMS/ETCS System Requirements Specification. Technical report, 2010.

[FLB+19] Steve Jeffrey Tueno Fotso, Régine Laleau, Héctor Ruíz Barradas, Marc Frappier, and Amel Mammar. A formal requirements modeling approach : Application to rail communication. In *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019.*, pages 170–177, 2019.

[FMCB19] Moustafa Fayad, Ahmed Mostefaoui, Samir Chouali, and Salima Benbernou. Fall detection application for the elderly in the family heroes system. In Frank Y. Li and Rodolfo W. L. Coutinho, editors, *Proceedings of the 17th ACM International Symposium on Mobility Management and Wireless Access, MobiWac 2019, Miami Beach, FL, USA, November 25-29, 2019*, pages 17–23. ACM, 2019.

[Han98] J. Han. A comprehensive interface definition framework for software components. In *The 1998 Asia Pacific software engineering conference*, pages 110–117. IEEE Computer Society, 1998.

[HC01] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering : Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[HJGD18] P. Huang, K. Jiang, C. Guan, and D. Du. Towards modeling cyber-physical systems with sysml/marte/pccsl. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 264–269, July 2018.

[HMC13a] Ahmed Hammad, Hassan Mountassir, and Samir Chouali. An approach combining sysml and modelica for modelling and validate wireless sensor networks. In Flávio Oquendo, Paris Avgeriou, Carlos E. Cuesta, José Carlos Maldonado, Elisa Yumi Nakagawa, Khalil Drira, and Andrea Zisman, editors, *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems, SESoS@ECOOP, Montpellier, France, July 2, 2013*, pages 5–12. ACM, 2013.

[HMC13b] Ahmed Hammad, Hassan Mountassir, and Samir Chouali. Combining sysml and modelica to verify the wireless sensor networks energy consumption. In Slimane Hammoudi, Luís Ferreira Pires, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, pages 198–201. SciTePress, 2013.

[HMN+08] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K. Lundback. The rubus component model for resource constrained real-time systems. In *2008 International Symposium on Industrial Embedded Systems*, pages 177–183, June 2008.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23 :279–295, May 1997.

[Hol03] Gerard Holzmann. *Spin Model Checker, the : Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[HSS02] Maritta Heisel, Thomas Santen, and Jeanine Souquières. Toward a formal model of software components. In *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, pages 57–68, 2002.

[ISO12] ISO/IEC 8652. Information Technology – Programming Language – ADA. Standard, International Organization for Standardization, 2012.

[IST11] Paola Inverardi, Romina Spalazzese, and Massimo Tivoli. *Application-Layer Connector Synthesis*, pages 148–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[IT03] Paola Inverardi and Massimo Tivoli. Software Architecture for Correct Components Assembly. In *SFM 2003, Bertinoro, Italy*, pages 92–121, 2003.

[JA12] Sung-Shik T. Q. Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Sci. Ann. Comp. Sci.*, 22 :201–251, 2012.

[JDB09] Yosr Jarraya, Mourad Debbabi, and Jamal Bentahar. On the Meaning of SysML Activity Diagrams. In *Proceedings of the 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '09, pages 95–105, Washington, DC, USA, 2009. IEEE Computer Society.

[KAdSS11] Daniel Knorreck, Ludovic Apvrille, and Pierre de Saqui-Sannes. TEPE : a SysML language for time-constrained property modeling and formal verification. *SIGSOFT Softw. Eng. Notes*, 36 :1–8, January 2011.

[KBSB10] Marouane Kessentini, Arbi Bouchoucha, Houari Sahraoui, and Mounir Boukadoum. Example-based sequence diagrams to colored petri nets transformation using heuristic search. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *Modelling Foundations and Applications*, pages 156–172, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Kem10] Stephanie Kemper. Compositional construction of real-time dataflow networks. In Dave Clarke and Gul Agha, editors, *Coordination Models and Languages*, pages 92–106, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Kem12] S. Kemper. Sat-based verification for timed component connectors. *Science of Computer Programming*, 77(7) :779 – 798, 2012. (1) FOCLASA'09 (2) FSEN'09.

[KEP07] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal Verification of Component-based Designs. *Design Autom. for Emb. Sys.*, 11(1) :49–90, 2007.

[KKdV10] Natallia Kokash, Christian Krause, and Erik P. de Vink. Verification of context-dependent channel-based service models. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects*, pages 21–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[KKdV12] Natallia Kokash, Christian Krause, and Erik de Vink. Reo + mcrl2 : A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 24(2) :187–216, Mar 2012.

[KLLS12] Wei Ke, Xiaoshan Li, Zhiming Liu, and Volker Stolz. rcos : a formal model-driven engineering method for component-based software. *Frontiers Comput. Sci. China*, 6(1) :17–39, 2012.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0 : Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Kon95] Dimitri Konstantas. *Interoperation of object-oriented applications*, pages 69–95. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[KSA+08] Ramtin Khosravi, Marjan Sirjani, Nesa Asoudeh, Shaghayegh Sahebi, and Hamed Iravanchi. Modeling and analysis of reo connectors using alloy. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, pages 169–183, Berlin, Heidelberg, 2008. Springer-Verlag.

[LCA18] Benjamin Lion, Samir Chouali, and Farhad Arbab. Compiling protocols to promela and verifying their LTL properties. In *Proceedings of MODELS 2018 Workshops : ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 31–39, 2018.

[LCA20] Benjamin Lion, Samir Chouali, and Farhad Arbab. Compiling synchronous protocols to asynchronous promela programs for ltl properties verification. Technical report, DISC department, FEMTO-ST Institute, UMR CNRS 6174, July 2020. to be submitted for publication.

[Lec09] Thierry Lecomte. Applying a formal method in industry : a 15-year trajectory. In *Procs. of 14th International Workshop, FMICS 2009*, LNCS, pages 26–34. Springer Berlin Heidelberg, 2009.

[LNRT10] K. Lau, A. Nordin, T. Rana, and F. Taweel. Constructing component-based systems directly from requirements using incremental composition. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 85–93, Sep. 2010.

[LNW07] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal i/o automata for interface and product line theories. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[LO01] Kung-Kiu Lau and Mario Ornaghi. A formal approach to software component specification. In G.T. Leavens D. Giannakopoulou and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, 2001. Tampa, USA, October 2001.

[Loc10] D. Locke. MQ telemetry transport (MQTT) V3.1 protocol specification. Technical report, 2010. IBM developerWorks Technical Library.

[LS08] Arnaud Lanoix and Jeanine Souquières. Trustworthy assembly of components using the B refinement. *e-Informatica*, 2(1) :9–28, 2008.

[LSM+10] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouët. A first attempt to combine sysml requirements diagrams and B. *ISSE*, 6(1-2) :47–54, 2010.

[LTM+09] Vitor Lima, Chamseddine Talhi, Djedjiga Mouheb, Mourad Debbabi, Lingyu Wang, and Makan Pourzandi. Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. *Electr. Notes Theor. Comput. Sci.*, 254 :143–160, 2009.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16 :1811–1841, November 1994.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Software Eng.*, 33(10) :709–724, 2007.

[LX04] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Form. Asp. Comput.*, 16(3) :210–237, 2004.

[MACM15a] S. Mouelhi, K. Agrou, S. Chouali, and H. Mountassir. Object-oriented component-based design using behavioral contracts (version with proofs). Research report, DISC department, FEMTO-ST Institute, UMR CNRS 6174, 2015. hal.archives-ouvertes.fr.

[MACM15b] Sebti Mouelhi, Khalid Agrou, Samir Chouali, and Hassan Mountassir. Object-oriented component-based design using behavioral contracts : Application to railway systems. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2015, Montreal, QC, Canada, May 4-8, 2015*, pages 49–58, 2015.

[MBMC19] Mohammed-Amine Merzoug, Azzedine Boukerche, Ahmed Mostefaoui, and Samir Chouali. Spreading aggregation : A distributed collision-free approach for data aggregation in large-scale wireless sensor networks. *Journal of Parallel and Distributed Computing*, 125 :121 – 134, mar 2019.

[MCM09] Sebti Mouelhi, Samir Chouali, and Hassan Mountassir. Refinement of interface automata strengthened by action semantics. *Electron. Notes Theor. Comput. Sci.*, 253 :111–126, October 2009.

[MCM11] Sebti Mouelhi, Samir Chouali, and Hassan Mountassir. Invariant preservation by component composition using semantical interface automata. In *Proceedings of the Sixth International Conference on Software Engineering*

Advances ICSEA 2011 (IARIA Conferences), to appear. IEEE Computer Society, 2011.

[Mer14] Elkamel Merah. Design of atl rules for transforming uml 2 sequence diagrams into petri nets. *International Journal of Computer Science and Business Informatics*, 8(1), 2014.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10) :40–51, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[MLS06] I. Mouakher, A. Lanoix, and J. Souquières. Component Adaptation : Specification and Verification. In *The proceedings of the 11th International Workshop on Component Oriented Programming (WCOP'06)*, pages 23–30, Nantes, France, July 2006.

[MMC17] Mohammed Amine Merzoug, Ahmed Mostefaoui, and Samir Chouali. Distributed collision-free data aggregation approach for wireless sensor networks. In *13th International Conference on Distributed Computing in Sensor Systems, DCOSS 2017, Ottawa, ON, Canada, June 5-7, 2017*, pages 175–182. IEEE, 2017.

[MMTL+16] Saad Mubeen, Jukka Mäki-Turja, John Lundbäck, Mattias Glnander, Kurt-Lennart Lundbäck, Mikael Sjödin, and Bud Lawson. Academic-industrial Collaboration in the Vehicle Software Domain : Experiences and End-user Perspective. In Matthieu Roy, editor, *CARS 2016 - Critical Automotive applications : Robustness & Safety*, Workshop CARS 2016 - Critical Automotive applications : Robustness & Safety, Göteborg, France, September 2016.

[Mou11] Sebti Mouelhi. *Contributions à la vérification de la sûreté de l'assemblage et à l'adaptation de composants réutilisables. (Contributions to the formal verification of the assembly and adaptation of reusable components)*. PhD thesis, University of Franche-Comté, Besançon, France, 2011.

[MPM08] Tarek Melliti, Pascal Poizat, and Sonia Ben Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 146–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[MPS12] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Trans. Software Eng.*, 38(4) :755–777, 2012.

[MS99] Eric Meyer and Jeanine Souquières. A systematic approach to transform omt diagrams to a b specification. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 875–895, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[OMD13] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. A probabilistic verification framework of sysml activity diagrams. In *IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques, SoMeT 2013, Budapest, Hungary, September 22-24, 2013*, pages 165–170. IEEE, 2013.

[OMD14] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, 41(6) :2713 – 2728, 2014.

[OSB13] Meriem Ouederni, Gwen Salaün, and Tevfik Bultan. Compatibility checking for asynchronously communicating software. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 310–328, 2013.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. volume 46 of *Advances in Computers*, pages 329 – 400. Elsevier, 1998.

[Pad09] Luca Padovani. Contract-based discovery and adaptation of web services. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, volume 5569 of *Lecture Notes in Computer Science*, pages 213–260. Springer, 2009.

[PdAHSV02] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis : two faces of the same coin. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 132–139, New York, NY, USA, 2002. ACM.

[PEML10] Jean-François Pétin, Dominique Evrot, Gérard Morel, and Pascal Lamy. Combining SysML and formal methods for safety requirements verification. In *22nd International Conference on Software & Systems Engineering and their Applications*, page CDROM, Paris, France, December 2010.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera : A hybrid theorem prover for hybrid systems. In *Procs. of IJCAR 2008*, volume 5195 of *LNCS*, pages 171–178. Springer Berlin Heidelberg, 2008.

[PQ09] André Platzer and Jan-David Quesel. European Train Control System : A case study in formal verification. In *Formal Methods and Software Engineering*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.

[PST07] Pascal Poizat, Gwen Salaün, and Massimo Tivoli. An adaptation-based approach to incrementally build component systems. *Electr. Notes Theor. Comput. Sci.*, 182 :155–170, 2007.

[RBB+09] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces : unifying interface automata and modal specifications. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 87–96, 2009.

[Reu03] Ralf H. Reussner. Automatic component protocol adaptation with the coconut/j tool suite. *Future Generation Computer Systems*, 19(5) :627 – 639, 2003. Tools for Program Development and Analysis. Best papers from two Technical Sessions, at ICCS2001, San Francisco, CA, USA, and ICCS2002, Amsterdam, The Netherlands.

[RF06] Oscar R Ribeiro and João M Fernandes. Some rules to transform sequence diagrams into coloured petri nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 237–56. Citeseer, 2006.

[RHI17] Messaoud Rahim, Ahmed Hammad, and Malika Ioualalen. A methodology for verifying sysml requirements using activity diagrams. *Innovations in Systems and Software Engineering*, 13(1) :19–33, 2017.

[RKR19] Alejandro Rodríguez, Lars Michael Kristensen, and Adrian Rutle. *Formal Modelling and Incremental Verification of the MQTT IoT Protocol*, pages 126–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[Roz15] Oscar Alberto Carrillo Rozo. *Formal and Incremental Verification of SysML Specifications for the Design of Component-Based Systems*. PhD thesis, Université de Franche-Comté, Besançon, France, 2015.

[SC05] Jeanine Souquières and Samir Chouali. Verifying the compatibility of component interfaces using the B formal method. In *Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2005, Las Vegas, Nevada, USA, June 27-29, 2005, Volume 2*, pages 850–856, 2005.

[SEG08] R. Seguel, R. Eshuis, and P. Grefen. An overview on protocol adaptors for service component integration. Technical report, BETA Working Paper Series WP 265, Eindhoven University of Technology, 2008.

[SEG09] R. Seguel, R. Eshuis, and P. Grefen. An overview on protocol adaptors for service component integration, 2009.

[SLMP13] Walter Schön, Guy Larraufie, Gilbert Moens, and Jacques Pore. *Railway Signalling and Automation*, volume 3. La Vie du Rail, 2013.

[Smi99] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.

[STE] STERIA. *Atelier B : Preuves et Exemples*.

[sys] OMG systems Modeling Language (OMG SysML). http://www.omgsysml.org/. Accessed 2020-02-26.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TI08] Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming*, 71(3) :181 – 212, 2008.

[uml] The Unified Modelling Language - UML. http://www.uml.org. Accessed : 2020-02-26.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems : Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, Advanced Lectures*, pages 25–43, 2003.

[vL09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[VNnT99] Antonio Vallecillo, Juan Hernández Núñez, and José M. Troya. Object interoperability. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 1–21, London, UK, UK, 1999. Springer-Verlag.

[VW01] W. Vanderperren and B. Wydaeghe. Towards a new component composition process. *IEEE ECBS 2001, Eighth Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 322–329, 2001.

[Wan08] Wang, Renzhong and Dagli, C. H. An Executable System Architecture Approach to Discrete Events System Modeling Using SysML in Conjunction with Colored Petri Net. In *Systems Conference, 2008, 2nd Annual IEEE*, 2008.

[Weg96] Peter Wegner. Interoperability. *ACM Comput. Surv.*, 28(1) :285–287, March 1996.

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2) :292–333, 1997.

[ZTL+11] Yan Zhang, Tao Tang, KePing Li, JoseManuel Mera, Li Zhu, Lin Zhao, and TianHua Xu. Formal verification of safety protocol in train control system. *Science China Technological Sciences*, 54(11) :3078–3090, 2011.

[ZW95] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching : a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2) :146–170, 1995.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4) :333–369, 1997.

TABLE DES FIGURES

1.1 Conception incrémentale d'un SBC guidée par SysML 8
1.2 Assemblage des composants basé sur les Automates d'interfaces 10

2.1 Le BDD du bloc composite *Station* 20
2.2 Le IBD du bloc *Station* 21
2.3 Le diagramme de séquence du bloc Sensors 21
2.4 Illustration de notre approche. 22
2.5 Le méta modèle d'un automate d'interface. 22
2.6 Les automates d'interface des sous-composants *Sensors* et *CU* 23
2.7 L'automate d'interface de la composition de *Sensors* et *CU* 27
2.8 Illustration de l'approche 28

3.1 Approche d'adaptation ascendante des blocs SysML 32
3.2 Illustration de l'approche d'adaptation des blocs SysML. 32
3.3 Le bloc composite Robot et son protocole d'interaction. 35
3.4 Les blocs *Controller* et *Motor* et leurs protocoles 37
3.5 Le protocole du bloc adaptateur $ADContrMot$ 42

4.1 Illustration des étapes de l'approche proposée 46
4.2 Un système de sécurité d'un véhicule 49
4.3 Le diagramme des exigences SysML pour le système de sécurité d'un véhicule 50
4.4 Le DS du composant Sensors 53
4.5 Le code Promela pour le composant Sensors 53
4.6 Le DS du composant ACU 53
4.7 Le code Promela pour le composant ACU 53
4.8 L'automate d'interface de Sensors 58
4.9 L'automate d'interface de ACU 58
4.10 L'automate d'interface de la composition des composants Sensors et ACU . 58
4.11 Le BDD du bloc *SensorsControl* 58

4.12 Le IBD du bloc *SensorsControl* 59

5.1 Illustration d'un automate d'interface 67
5.2 Les fonctions simplifiées de la protection des trains. 69
5.3 Modélisation UML de l'architecture du système. 70
5.4 Automates d'interface des composants OBD, DEU, and MCU : actions de méthodes (transitions doubles) ; action de retour (transitions simples) ; actions des exception (transitions en pointillés). 71
5.5 Automate d'interfacon $(B_d|B_m).\mathcal{A}$. 73
5.6 Automate d'interface $(B_d||B_m).\mathcal{A}$. 74
5.7 L' automate d'interface raffiné de MCU 76
5.8 Processus d'adaptation de deux automates d'interface composables A_1 et A_2. 79
5.9 Illustration de l'approche d'adaptation 85

6.1 Architecture à composants du système de réservation d'hôtel 91
6.2 Le modèle de donnée d'interface de *IHotelHandling* 92
6.3 La spécification B des classes dans *IHotelHandling* 92
6.4 La spécification B des associations entre les classes 93
6.5 La spécification B des opérations 93
6.6 Le modèle de données d'interface de *IHotelMgt* 94
6.7 La spécification B de la classe *RoomType* dans *IHotelMgt* 95
6.8 L'opération *makeReservation* dans *IHotelMgt* 95
6.9 Spécification B de *New_IHotelMgt* 97

7.1 Syntaxe graphique pour les primitives Reo. 103

LISTE DES TABLES

4.1 Implémentation des concepts basiques d'un diagramme de séquence avec Promela 52

5.1 Sémantique de l'action de méthode $covReq$. 71

7.1 Exemple d'un comportement de *fifo* avec le domaine $D = \{1, *\}$ 104

7.2 Formalisation en Promela des propriétés LTL des connecteurs Reo 110

Résumé :

Mes travaux portent sur la conception rigoureuse des systèmes à base de composants (SBC) critiques, en exploitant des notations semi-formelles et des approches formelles. Je m'intéresse particulièrement à l'exploitation d'un formalisme particulier des automates, appelé automates d'interface, pour proposer des approches de conception incrémentales par contrats. Ce qui permet de vérifier l'interopérabilité, la substituabilité, et l'adaptabilité des composants, lors de leur assemblage. Mes travaux se focalisent également sur l'exploitation des modèles semi-formels, particulièrement SysML, pour formaliser les exigences des SBC dans le but de les prendre en considération lors de la vérification de la cohérence des architectures des SBC. Plus récemment, j'ai travaillé, dans le cadre de collaborations, sur la spécification et la vérification des protocoles de coordination dans les SBC, ainsi que sur l'utilisation des approches formelles pour la validation des protocoles de communication dans les véhicules connectés.

Mots-clés : Systèmes à base de composants, SysML, automates d'interface, conception par contrats, composition, compatibilité, exigences, vérification, raffinement, adaptation.

Abstract:

My research focuses on the rigorous design of critical component-based systems (CBS) using semi-formal notations and formal approaches. I am particularly interested in the exploitation of a particular formalism of automata, called interface automata, to propose incremental design approaches by contracts. Which allows to verify interoperability, substitutability and adaptability of components during their assembly. My work also focuses on the use of semi-formal models, particularly SysML, to formalize the requirements of CBS in order to consider them when checking the consistency of CBS architectures. More recently, I have worked, in the context of collaborations, on the specification and verification of coordination protocols in CBS, as well as on the use of formal approaches for the validation of communication protocols in connected vehicles.

Keywords: Component-based systems, SysML, interface automata, design by contracts, composition, compatibility, requirements, verification, refinement, adaptation.

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr

UFC UNIVERSITÉ DE FRANCHE-COMTÉ